Compressed verification for post-quantum signatures with long-term public keys

Gustavo Banegas, Anaëlle Le Dévéhat, and Benjamin Smith

Inria and Laboratoire d'Informatique de l'Ecole polytechnique,
Institut Polytechnique de Paris, Palaiseau, France
gustavo@cryptme.in
anaelle.le-devehat@inria.fr
smith@lix.polytechnique.fr

Abstract. Many signature applications—such as root certificates, secure software updates, and authentication protocols—involve long-lived public keys that are transferred or installed once and then used for many verifications. This key longevity makes post-quantum signature schemes with conservative assumptions (e.g., structure-free lattices) attractive for long-term security. But many such schemes, especially those with short signatures, suffer from extremely large public keys. Even in scenarios where bandwidth is not a major concern, large keys increase storage costs and slow down verification. We address this with a method to replace large public keys in GPV-style signatures with smaller, private verification keys. This significantly reduces verifier storage and runtime while preserving security. Applied to the conservative, short-signature schemes Wave and Squirrels, our method compresses Squirrels-I keys from 665 kB to 20.7 kB and Wave822 keys from 3.5 MB to 207.97 kB.

Keywords: Post-quantum cryptography · Digital Signatures · Lattice-based cryptography · Code-based cryptography · Compressed GPV.

1 Introduction

Post-quantum signatures are a primary requirement for the transition towards quantum-resistant cryptography. Post-quantum lattice- and code-based signatures can be roughly classified as *conservative* or *structured*, according to whether their underlying hard problems involve general codes and lattices, or involve special algebraic structure. These structures facilitate important practical improvements (often, much smaller public keys); but they also allow the possibility of specialized attacks, so structured schemes generally have weaker security arguments (i.e., their assumptions are stronger).

For example: compare the structured lattice scheme Falcon [13] with the conservative lattice scheme Squirrels [11]. Both are based on the same GPV

* Author list in alphabetical order; see https://ams.org/profession/leaders/CultureStatement04.pdf. This work was supported by the HYPERFORM consortium, funded by France through Bpifrance, and by the France 2030 program under grant agreement ANR-22-PETQ-0008 PQ-TLS. Date of this document: 2025-09-02.

design [14]. At NIST post-quantum security level 1, Falcon and Squirrels have comparable signature sizes: 666 and 1019 bytes, respectively. But while the structured lattices of Falcon give 897-byte public keys, the unstructured lattices of Squirrels push public-key sizes up to 665 *kilo*bytes.

Signature schemes with large public keys are unsuitable for applications where public keys are regularly transmitted, such as TLS certificates. They are better-suited to applications where

- the public key is pre-installed on the verifier's device (for verifying signed software updates, for example, or root certificates), or
- the public key is transmitted, but the cost of transmission is amortised over many subsequent verifications (in `ssh` authentication, for example).

These applications often involve public keys with *very* long lifetimes: 20-30 years for root certificates like ISRG Root X1 and GlobalSign Root R1, for example, and a decade or more for IoT code-signing certificates and government-issued digital IDs.

The long-term nature of keys in these applications makes conservative security assumptions reassuring, but working with very large public keys remains expensive and inconvenient. This makes hash-based signatures like SLH-DSA (SPHINCS+) [3, 21] an interesting choice: they offer conservative security assumptions *and* very small public keys—but at the cost of large signatures and computationally intensive verification. Squirrels offers shorter signatures and faster verification, but its 665 kB public keys make long-term storage impractical.

1.1 Compressed verification

We want to reduce public key storage for conservative GPV-style signatures. The core idea is that the verifier can (pre-)process the public key PK once to derive a much smaller verification key VK (private to the verifier), which they can then use in place of PK for confident—and often much faster—signature verification.

More formally: suppose we are given a signature scheme defined by three algorithms, with an implicit security parameter λ:

- `KeyGen`: returns a private key SK and a public key PK.
- `Sign`: given a private key SK and a message m, returns a signature σ.
- `Verify`: given a putative signature σ on m under a public key PK, returns Accept or Reject.

We will define three additional algorithms to be used by the verifier:

- `CKeyGen`: returns a (private) compression key CK.
- `VKeyGen`: given a public key PK and a (private) compression key CK, return a private verification key VK.
- `CVerify`: given a putative signature σ on m and a verification key VK, returns Accept or Reject.

The goal is to define these functions such that if $\mathsf{VK} = \texttt{VKeyGen}(\mathsf{CK}, \mathsf{PK})$ for some public key PK and some CK output by `CKeyGen`, then

1. if $\texttt{Verify}(\sigma,\ m,\ \mathsf{PK}) = \mathsf{Accept}$ then $\texttt{CVerify}(\sigma,\ m,\ \mathsf{VK}) = \mathsf{Accept}$;
2. if $\texttt{Verify}(\sigma,\ m,\ \mathsf{PK}) = \mathsf{Reject}$ then $\texttt{CVerify}(\sigma,\ m,\ \mathsf{VK}) = \mathsf{Reject}$ with probability $\geq 1 - 1/2^{\mu}$ for a second security parameter μ; and
3. the size of VK is much smaller than the size of PK.

In terms of storage, CK is generated randomly and—once it has been used in `VKeyGen`—need not be stored. Therefore, the verifier only needs to retain VK.

The verifier is free to choose μ; our goal is that the probability that `CVerify` accepts one forgery after Q attempts is on the order of $Q/\#\mathcal{S}$, where $\mathcal{S}$ is the verifier's compression-keyspace. We will generally take $\#\mathcal{S} \approx 2^{\mu}$ with $\mu \approx \lambda$, assuming Q is relatively small (in any case, $Q \leq 2^{64}$). The verifier may force a limit on Q by refreshing VK after a given number of rejections.

Figure 1 illustrates the signature protocol with compressed verification. CK and VK are private *to the verifier* and are derived only from the signer's public key PK, and not the signer's private key SK. Additionally, the verification algorithm `CVerify` does not require PK or CK. From the signer's point of view, the original signature scheme is unchanged.

Signer		**(Compressed) Verifier**
$\mathsf{SK}, \mathsf{PK} \leftarrow \texttt{KeyGen()}$		$\mathsf{CK} \leftarrow \texttt{CKeyGen()}$
	$\xrightarrow{\mathsf{PK}}$	
$\sigma \leftarrow \texttt{Sign}(\mathsf{SK}, m)$		$\mathsf{VK} \leftarrow \texttt{VKeyGen}(\mathsf{PK},\ \mathsf{CK})$
	$\xrightarrow{(m,\sigma)}$	
		$\mathsf{Accept}\vert\mathsf{Reject} \leftarrow \texttt{CVerify}(m, \sigma, \mathsf{VK})$

Fig. 1. Compressed verification as a protocol.

1.2 Results

We introduce a framework for *compressed verification* in GPV-style signatures. The verifier chooses a private homomorphism ϕ and uses it to compress incoming public keys to compact *private verification keys*. We give security arguments for the general construction, and consider two conservative instantiations: Squirrels [11], a lattice-based signature, and Wave [1, 9], a code-based signature.

Squirrels and Wave both have little special algebraic structure. This builds confidence in their security, but it comes at an important practical cost: as we see in Table 1, their public keys are *very* big. Indeed, among the submissions to the NIST Postquantum Signatures on-ramp with no vulnerabilities found during Round 1, Squirrels and Wave had the largest public keys at most security levels.[1] These oversized public keys would be a major factor in the non-selection

[1] The *classic* parameters of UOV, a conservative multivariate scheme, had larger public keys than Squirrels at NIST PQ Security Level 5, but other UOV variants had

of these schemes for Round 2 of the standardization process. They also make Squirrels and Wave prime candidates to demonstrate the effectiveness of our technique.

We developed two implementations of both Squirrels and Wave:

1. A comprehensive implementation in Python, and
2. A C implementation, based on the reference implementations of Squirrels [27] and Wave, used to measure performance improvements.

Table 1 shows the significant reduction in verifier storage achieved for both Squirrels and Wave while maintaining security levels (and backward compatibility with the original schemes). Specifically, we achieve a **compression ratio of up to 34x** for Wave at higher security levels, and **over 26x** for Squirrels at comparable levels. Compressed verification can also reduce verification time: verification is **up to 9.26% faster** than ordinary verification for Squirrels and **up to 30% faster** for Wave.

Table 1. Signature scheme parameters and verification times. All sizes are in bytes. Compressed verification parameters are chosen such that $\mu \approx \lambda$ (see §4 and §5 for details). Verification times are cycle counts on an Intel Core i7-1365U processor running Arch Linux Kernel 6.11.5-arch1. "Reference" refers to C reference implementations, and "Compressed" to our own C code. Wave signatures are variable-length; sizes here are upper bounds, and Wave signature length roughly doubles for compressed verification.

	Reference		Compressed verification			Verification time (kCycles)		
	$\lvert\sigma\rvert$	$\lvert PK\rvert$	$\lvert VK\rvert$	$(\lvert CK\rvert)$	$\lvert PK\rvert/\lvert VK\rvert$	Reference	Compressed	*Speedup*
NIST PQ Security Level 1, classical $\lambda = 128$								
Squirrels-I	1 019	**681 780**	**20 700**	(3 360)	*32.9×*	280	254	*9.3%*
Wave822	822	**3 677 390**	**171 594**	(83 822)	*21.4×*	1 101	762	*30.8%*
SPHINCS+-SHAKE-128s	7 856	32	—	—	—	3 285	—	—
XMSS^{MT}-20-2	4 963	64	—	—	—	2 868	—	—
NIST PQ Security Level 3, classical $\lambda = 192$								
Squirrels-III	1 554	**1 629 640**	**49 824**	(8 480)	*32.7×*	551	520	*5.69%*
Wave1249	1 249	**7 867 598**	**266 188**	(183 134)	*29.6×*	2 330	1 865	*19.9%*
SPHINCS+-SHAKE-192s	16 224	48	—	—	—	5 374	—	—
XMSS^{MT}-40-4	9 893	64	—	—	—	5 729	—	—
NIST PQ Security Level 5, classical $\lambda = 256$								
Squirrels-V	2 025	**2 786 580**	**90 598**	(15 048)	*30.8×*	916	898	*1.9%*
Wave1644	1 644	**13 632 308**	**370 436**	(321 507)	*36.8×*	3 911	3 198	*18.2%*
SPHINCS+-SHAKE-256s	29 792	64	—	—	—	7 567	—	—
XMSS^{MT}-60-6	14 824	64	—	—	—	8 743	—	—

As a baseline for conservative signatures, Table 1 includes the hash-based schemes SPHINCS+ and XMSS^{MT} [15]. For SPHINCS+, the authors provide a benchmark script, which we ran in our environment; in the table we report the fastest result produced by that script [26]. For XMSS^{MT}, we used the reference code [16] with the smallest parameters for each security level.

smaller keys. In any case, Wave is the undisputed super-heavyweight champion: its Level I public keys were larger than even the Level V keys of any other scheme.

Remark 1. Compressed verification may also benefit schemes like Ajtai-based hash-and-sign [7,19], which offer strong SIS-based security. The benefit is limited for structured GPV-style schemes such as Falcon [13]: public keys are already compact, and compressed verification requires the full (s_1, s_2) signature rather than just s_2, thus doubling signature size.

Remark 2. Our method is compatible with the PS-3 [24] and BUFF [8] transforms, which strengthen security in certain attack models. However, both require hashing PK with the message—a costly step when PK is large. We suggest storing the much smaller Hash(PK) and hashing that with the message instead.

1.3 Related work: flexible signatures and progressive verification

Fischlin's *progressive verification* for MACs [12] can probabilistically Reject early or Accept with reduced confidence; [12, §4] suggests an extension to signatures. Le, Kelkar, and Kate's *flexible signatures* [17], verified up to a real "confidence level" $0 \leq \alpha \leq 1$, quantify "partial" verification when expensive verification operations are interrupted voluntarily by the user or forcibly by the OS. This can improve fault-tolerance and reduce the cost of verification in embedded applications, but it does not reduce public-key or signature sizes. Indeed, the main targets in [17] are hash-based signatures, where public keys are already extremely compact; but extensions to GPV signatures are projected in [17, §5.3].

Taleb and Vergnaud revisit progressive verification in [28], analyzing Bernstein's RSA trick (see §2) and GPV signatures (including Wave). They propose verifying GPV signatures using a small set of linear combinations of columns based on a random linear code, achieving exponential confidence growth with runtime, unlike the linear growth in [12] and [17]. However, their approach significantly increases public key size, which is the opposite of our goal.

Boschini, Fiore, Pagnin, Torresetti, and Visconti [6] propose an *efficient verification* for signatures which verify using a matrix-vector product $\mathbf{M}\mathbf{v}^\top$. In an "offline" phase they compute a matrix $\mathbf{M}'$ formed by k random linear combinations of the n rows of $\mathbf{M}$; then, in an "online" phase, they verify using $\mathbf{M}'\mathbf{v}^\top$, with reduced confidence but with a speedup of n/k. This is very similar to what we do with Wave in §5, but they repeat the offline phase for every verification rather than maintaining the same $\mathbf{M}'$; indeed, their focus is minimising online verification latency, rather than reducing overall verification time or key sizes.

2 Warmup: Bernstein's trick for Rabin–Williams

As a warmup, we recall Bernstein's fast Rabin–Williams signature verification [2]. We write Primes(μ) for the set of (exactly) μ-bit primes: that is,

$$\text{Primes}(\mu) := \{2^{\mu-1} < p < 2^{\mu} \mid p \text{ is prime}\}\,.$$

2.1 Verifying Rabin–Williams signatures

A Rabin–Williams signature [25,30] on a message m under a public key $N = pq$ is a tuple $\sigma = (e, f, \mathsf{salt}, s)$ such that

$$efs^2 \equiv \mathsf{Hash}(\mathsf{salt} \parallel m) \pmod{N} \tag{1}$$

where $1 < s < N$, salt is a salt value, and $e \in \{-1, 1\}$ and $f \in \{1, 2\}$ are chosen such that s exists for the given salt, m, and N. That is: given $\sigma = (e, f, \mathsf{salt}, s)$, m, and $\mathsf{PK} = N$, `Verify(`σ`, `m`, PK)` returns Accept if and only if (1) holds.

If (1) is satisfied, then there is a unique integer $-2N < t < 2N$ such that

$$efs^2 - tN = \mathsf{Hash}(\mathsf{salt} \parallel m)\,. \tag{2}$$

(The sign of t is equal to e.) Note that (2) holds over $\mathbb{Z}$, not just over $\mathbb{Z}/N\mathbb{Z}$; and any solution to (2) yields a solution to (1) and vice versa, so verifying (1) or (2) is mathematically (though not algorithmically) equivalent.

Bernstein suggested speeding up verification by including t in σ and verifying (2) modulo a random λ-bit prime ℓ (with λ the security parameter). The verifier picks $\ell \in \textsc{Primes}(\lambda)$, computes $N_\ell := N \bmod \ell$, and upon receiving $\sigma = (e, f, \mathsf{salt}, s, t)$, checks

$$efs_\ell^2 - t_\ell N_\ell \equiv h_\ell \pmod{\ell}, \tag{3}$$

where $s_\ell := s \bmod \ell$, $t_\ell := t \bmod \ell$, and $h_\ell := \mathsf{Hash}(\mathsf{salt} \parallel m) \bmod \ell$. Since $\ell \ll N$, this is faster than computing $s^2 \bmod N$; the speedup increases with λ. The trade-off: including t doubles the size of σ, and generating ℓ is relatively costly. However, as Bernstein notes, ℓ can be reused across multiple verifications—even for different public keys—if kept secret, amortizing the cost.

This trick was proposed to speed up verification. We observe that it also saves space if many signatures are verified under the same N, since N_ℓ can be stored instead of N (and this saves even more time, since N_ℓ need not be recomputed).

In terms of our framework above,

- `CKeyGen` samples a random λ-bit prime ℓ;
- `VKeyGen(CK = `ℓ`, PK = `N`)` returns $\mathsf{VK} = (\ell, N_\ell := N \bmod \ell)$;
- `CVerify(`m`, `$\sigma = (e, f, \mathsf{salt}, s, t)$`, VK = `$(\ell, N_\ell)$`)` returns Accept if and only if $efs^2 - tN_\ell \equiv \mathsf{Hash}(\mathsf{salt} \parallel m) \pmod{\ell}$.

As Bernstein observes, the same technique applies to RSA signatures. However, if e is the public exponent, the resulting integer t is roughly N^{e-1}, making the signature e times longer than a standard RSA signature.

2.2 Security argument for Bernstein's trick

Let's say that an ℓ-forgery on a message m for a public key $\mathsf{PK} = N$ is a vector $(e, f, \mathsf{salt}, s, t)$ such that (2) (and hence (1)) fails, but (3) holds. That is: an ℓ-forgery is a putative expanded Rabin–Williams signature that `Verify` with

$\mathsf{PK} = N$ would safely reject, but `CVerify` with $\mathsf{VK} = (\ell, N \bmod \ell)$ would accept. (We assume that forging a signature for (1) or (2) is infeasible.)

An adversary that knows ℓ can easily construct ℓ-forgeries for any m and N. Take a random salt, and find a t such that $x := \mathsf{Hash}(\mathsf{salt} \parallel m) + tN$ is a square modulo ℓ; then, compute $s := x^{1/2} \pmod{\ell}$ (which is easy because ℓ is prime); finally, set $e := 1$ and $f := 1$.

Conversely, if we can find an ℓ-forgery $\sigma = (e, f, \mathsf{salt}, s, t)$ for (m, N) then

$$\ell \mid \Xi(\sigma, m, N) \qquad \text{where} \qquad \Xi(\sigma, m, N) := efs^2 - tN - \mathsf{Hash}(\mathsf{salt} \parallel m)\,.$$

At just λ bits, the prime ℓ is sufficiently small (for cryptographic values of λ) to be recovered from $\Xi(\sigma, m, N)$ with ECM [18, 20, 31].[2]

An adversary $\mathcal{A}$ who can compute an ℓ-forgery can therefore find ℓ, and vice versa. But $\mathcal{A}$'s interaction with the verifier is limited to submitting tuples (σ, m, N), and observing whether they are accepted or rejected: that is, whether the unknown ℓ divides $\Xi(\sigma, m, N)$ or not. Let

$$\mu := \log_2 \ell \qquad \text{and} \qquad \kappa := \lfloor (\log_2 N)/\mu \rfloor\,.$$

Observe that $\mathcal{A}$ learns nothing from an ℓ-forgery attempt (σ, m, N) such that $\Xi(\sigma, m, N)$ is not divisible by at least one μ-bit prime, and that if $\Xi(\sigma, m, N) \neq 0$, then it is divisible by at most 2κ μ-bit primes (because $|\Xi(\sigma, m, N)| < 2N^2$). Therefore, if an adversary makes at most Q forgery attempts against a verifier using a μ-bit prime ℓ for VK, then their success probability is at most

$$P(N, \mu, Q) := \frac{2\kappa Q}{\#\textsc{Primes}(\mu)}\,. \tag{4}$$

It follows from [10, Corollary 5.3] that

$$0.975 \frac{2^{\mu-1}}{(\mu-1)\log 2} < \#\textsc{Primes}(\mu) < \frac{2^{\mu-1}}{(\mu-1)\log 2}. \tag{5}$$

Thus, $P(N, \mu, Q) \approx Q \cdot (\log N)/2^{\mu-2}$. For $\mu \approx \lambda$, this bound remains negligible even across more verifications than could feasibly be generated, without needing to refresh ℓ or track rejections.

3 The general approach

3.1 GPV signatures

Consider a general GPV-style signature. Let $\mathcal{M}$ be a finitely generated module over an integral domain $\mathcal{R}$: in practice, $\mathcal{M}$ is either a lattice (with $\mathcal{R} = \mathbb{Z}$) or a code (with $\mathcal{R} = \mathbb{F}_q$). Fix a cryptographic hash function $\mathsf{Hash} : \{0,1\}^* \to \mathcal{M}$. A public key is a random-looking $\mathbf{M} = (\mathbf{M}_0, \ldots, \mathbf{M}_{n-1}) \in \mathcal{M}^n$ for some system

[2] Further: if we can find two ℓ-forgeries σ_1 and σ_2—*not* necessarily for the same m and N—then $\ell \mid g := \gcd(\Xi(\sigma_1, m_1, N_1), \Xi(\sigma_2, m_2, N_2))$, and in fact probably $\ell = g$.

parameter n. A signature on a message m under $\mathbf{M}$ is a tuple $\sigma = (\mathsf{salt}, \mathbf{s})$ with $\mathsf{salt} \in \{0,1\}^\lambda$ (a random salt) and $\mathbf{s} \in \mathcal{R}^n$ such that

$$\textsc{Constraint}(\mathbf{s}) \qquad \text{and} \qquad \mathbf{sM} := \sum_{i=0}^{n-1} s_i \mathbf{M}_i = \mathsf{Hash}(\mathsf{salt} \parallel m) \tag{6}$$

where $\textsc{Constraint}(\mathbf{s})$ is a predicate on $\mathbf{s}$ such as having small norm (in Squirrels) or a fixed number of nonzero entries (in Wave). An important variant has Hash mapping into (a subset of) $\mathcal{R}^n$ instead of $\mathcal{M}$, and (6) is replaced by

$$\textsc{Constraint}(\mathbf{s}) \qquad \text{and} \qquad \big(\mathsf{Hash}(\mathsf{salt} \parallel m) + \mathbf{s}\big)\mathbf{M} = \mathbf{0}\,. \tag{7}$$

Example 1. In Squirrels, $\mathcal{R} = \mathbb{Z}$ and $\mathcal{M} = \mathbb{Z}/\Delta$ for some large Δ (though later we lift to $\mathcal{M} = \mathbb{Z}$); verification uses (7) where $\textsc{Constraint}(\mathbf{s})$ is $\|\mathbf{s}\|_2^2 \leq \lfloor \beta^2 \rfloor$ for a small system parameter β. For example, Squirrels-I has $n = 1034$, Δ a 5048-bit modulus formed as the product of 165 31-bit primes, and $\lfloor \beta^2 \rfloor = 2026590$.

Example 2. In Wave, $\mathcal{R} = \mathbb{F}_3$ and $\mathcal{M} = \mathbb{F}_3^{n-k}$ for system parameters n and k; verification uses (6) where $\textsc{Constraint}(\mathbf{s})$ is $\#\{i \mid s_i \neq 0\} = w$ for some (large) $w < n$. For example: Wave822 has $n = 8576$, $k = 4288$, and $w = 7668 \approx 0.9n$.

3.2 Compressed verification

Let Σ be a general GPV-style signature on an $\mathcal{R}$-module $\mathcal{M}$ as described above, and fix a set $\mathcal{S}$ of $\mathcal{R}$-submodules of $\mathcal{M}$. We define a compressed-verification signature scheme Σ_{comp} with the same `KeyGen` and `Sign` as in Σ, but with `Verify` replaced by the following `CKeyGen`, `VKeyGen`, and `CVerify`:

- `CKeyGen`: samples a random $\mathcal{K}$ from $\mathcal{S}$, and returns the quotient homomorphism $\mathsf{CK} := \phi : \mathcal{M} \to \overline{\mathcal{M}} := \mathcal{M}/\mathcal{K}$ (which is unique up to isomorphism, and has kernel $\ker\phi = \mathcal{K}$).
- `VKeyGen`: takes $\mathsf{PK} = \mathbf{M}$ and returns $\mathsf{VK} := (\phi, \overline{\mathbf{M}} := (\phi(\mathbf{M}_1), \ldots, \phi(\mathbf{M}_n)))$.
- `CVerify`: given m, $\sigma = (\mathsf{salt}, \mathbf{s})$, and VK, let $\mathbf{h} = \mathsf{Hash}(\mathsf{salt} \parallel m)$. If σ would normally be verified with (6), then `CVerify` returns Accept if and only if $\textsc{Constraint}(\mathbf{s})$ and $\phi(\mathbf{sM}) = \phi(\mathbf{h})$; and since ϕ is a homomorphism of $\mathcal{R}$-modules, this means

$$\textsc{Constraint}(\mathbf{s}) \qquad \text{and} \qquad \sum_{i=0}^{n-1} s_i \overline{\mathbf{M}}_i = \phi(\mathbf{h}) \quad \text{in } \overline{\mathcal{M}}\,. \tag{8}$$

  If verification normally uses (7), then `CVerify` returns Accept if and only if

$$\textsc{Constraint}(\mathbf{s}) \qquad \text{and} \qquad \sum_{i=0}^{n-1} (s_i + h_i) \overline{\mathbf{M}}_i = \mathbf{0} \quad \text{in } \overline{\mathcal{M}}\,. \tag{9}$$

If (9) is used for verification, then $\mathsf{CK} = \phi$ need not be included in VK.

If the verifier verifies many signatures from the same signer, then the cost of `CKeyGen` and `VKeyGen` is amortised over the many subsequent `CVerify` calls. A good choice of ϕ can also reduce the time required for each verification.

3.3 Correctness and security

The correctness of the signature scheme Σ_{comp} described above follows from ϕ being a homomorphism: (6) implies (8) and (7) implies (9). Our security goal is EUF-CMA (Existential unforgeability under adaptive chosen-message attacks). Recall that a signature scheme $\Sigma = (\mathsf{Gen}, \mathtt{Sign}, \mathsf{Verify})$ is said to be EUF-CMA if for all probabilistic polynomial-time (PPT) adversaries $\mathcal{A}$, the probability that $\mathcal{A}$ wins the following game is negligible in the security parameter λ:

1. The challenger gets $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{Gen}(1^\lambda)$ and gives pk to the adversary $\mathcal{A}$.
2. The adversary $\mathcal{A}$ has access to a signing oracle $\mathcal{O}_{\mathtt{Sign}}$ and a verification oracle $\mathcal{O}_{\mathtt{Verify}}$. It may adaptively query $\mathcal{O}_{\mathtt{Sign}}$ on messages $m_1, \ldots, m_q$ and receive valid signatures σ_i for each. It also may query $\mathcal{O}_{\mathtt{Verify}}$ on signatures σ_i and know if it is valid signature or not.
3. $\mathcal{A}$ outputs a pair (m^*, σ^*).
4. $\mathcal{A}$ wins the game if:
   (a) $\mathsf{Verify}_{\mathsf{pk}}(m^*, \sigma^*) = 1$ (i.e., the signature is valid), and
   (b) $m^* \notin \{m_1, \ldots, m_q\}$ (i.e., m^* was not queried to the signing oracle).

We want to relate the EUF-CMA security of a GPV signature with compressed verification to the assumed EUF-CMA security of the original scheme. We model forgery attempts as attempts at solving a hidden-structure problem:

Definition 1 (Submodule Element Guessing Problem $\mathsf{SEGP}(\mathcal{S}, \mathcal{T})$). *Let $\mathcal{M}$ be an $\mathcal{R}$-module, $\mathcal{T}$ a finite subset of $\mathcal{M}$, and $\mathcal{S}$ a set of $\mathcal{R}$-submodules of $\mathcal{M}$. The **Submodule Element Guessing Problem** $\mathsf{SEGP}(\mathcal{S}, \mathcal{T})$ is: given a membership oracle $\mathcal{O}_\mathcal{K}$ for an unknown submodule $\mathcal{K} \in \mathcal{S}$ taking input in $\mathcal{T}$ (i.e.: $\mathcal{O}_\mathcal{K}$ takes $\mathbf{t} \in \mathcal{T}$ and returns* True *if $\mathbf{t} \in \mathcal{K}$ and* False *otherwise), find an element $\mathbf{t}^* \neq \mathbf{0} \in \mathcal{T}$ such that $\mathcal{O}_\mathcal{K}(t^*) =$* True*, i.e., $\mathbf{t}^* \in \mathcal{K}$.*

The set $\mathcal{S}$ in Definition 1 represents the set of possible kernels of the secret homomorphism ϕ, and $\mathcal{T}$ represents the vectors constructed by forgery attempts before input to ϕ.

Definition 2. *Let Σ be a general GPV signature scheme as above. We define*

$$\mathcal{T}(\Sigma) := \left\{ \mathbf{sM} - \mathbf{h} : \mathbf{h} \in \mathrm{Im}(\mathsf{Hash}), \mathbf{s} \in \mathcal{R}^n \mid \textsc{Constraint}(\mathbf{s}) \right\}$$

if Σ uses (6) for verification, or

$$\mathcal{T}(\Sigma) := \left\{ (\mathbf{h} + \mathbf{s})\mathbf{M} : \mathbf{h} \in \mathrm{Im}(\mathsf{Hash}), \mathbf{s} \in \mathcal{R}^n \mid \textsc{Constraint}(\mathbf{s}) \right\}$$

if Σ uses (7) for verification.

Theorem 1. *Let Σ be a general GPV-style signature scheme on an $\mathcal{R}$-module $\mathcal{M}$, and fix a set $\mathcal{S}$ of $\mathcal{R}$-submodules of $\mathcal{M}$. For each $\mathcal{K}$ in $\mathcal{S}$, let $\Sigma_{comp}^{\mathcal{K}}$ be the instance of Σ_{comp} where* `CKeyGen` *samples $\mathcal{K}$ from $\mathcal{S}$. If $\mathcal{A}$ is an algorithm running in time T that wins the EUF-CMA game for $\Sigma_{comp}^{\mathcal{K}}$ with probability P, then there exists an algorithm $\mathcal{B}$ running in time $T + O(1)$ that succeeds with probability P in winning the EUF-CMA game for Σ, or solving the $\mathsf{SEGP}(\mathcal{S}, \mathcal{T}(\Sigma))$ instance corresponding to $\mathcal{K}$.*

Proof. The challenger for EUF-CMA game of Σ gives $\mathsf{PK} = \mathbf{M}$ to the algorithm $\mathcal{B}$. $\mathcal{B}$ has access to a signing oracle $\mathcal{O}_{\mathtt{Sign}}$, a verification oracle $\mathcal{O}_{\mathsf{Verify}}$ both associated to Σ and a submodule membership oracle $\mathcal{O}_K$ associated to $\mathsf{SEGP}(\mathcal{S}, \mathcal{T}(\Sigma))$. $\mathcal{B}$ calls $\mathcal{A}$ on $\mathsf{PK} = \mathbf{M}$. $\mathcal{B}$ answers a signing oracle query m from $\mathcal{A}$ by $\mathcal{O}_{\mathtt{Sign}}(m)$ as the signing process is the same for Σ and $\Sigma^{\mathcal{K}}_{\mathrm{comp}}$. If $\mathcal{A}$ makes a query to the verification oracle for $\Sigma^{\mathcal{K}}_{\mathrm{comp}}$ on a signature $\sigma = (\mathsf{salt}, \mathbf{s})$, $\mathcal{B}$ queries $\mathcal{O}_{\mathsf{Verify}}(\sigma)$. If $\mathcal{O}_{\mathsf{Verify}}(\sigma) = 1$ then $\mathcal{B}$ answers 1 to $\mathcal{A}$. Otherwise, If Σ verifies with (6) then $\mathcal{B}$ sets $\mathbf{t} := \mathbf{s}\mathbf{M} - \mathsf{Hash}(\mathsf{salt} \parallel m)$; if Σ verifies with (7) instead, then $\mathcal{B}$ sets $\mathbf{t} := (\mathsf{Hash}(\mathsf{salt} \parallel m) + \mathbf{s})\mathbf{M}$. $\mathcal{B}$ answers the verification oracle query by $\mathcal{O}_K(\mathbf{t})$.

If $\mathcal{A}$ fails, then $\mathcal{B}$ fails. Otherwise, it receives a $(m^*, \sigma^* = (\mathsf{salt}^*, \mathbf{s}^*))$ such that $\textsc{Constraint}(\mathbf{s}^*)$ holds and $\mathtt{CVerify}(m^*, \sigma^*, \mathsf{VK}) = \mathsf{Accept}$. $\mathcal{B}$ computes $\mathbf{t}^*$, if $\mathbf{t}^* = \mathbf{0}$, then $\mathtt{Verify}(m^*, \sigma^*, \mathsf{PK})$ would return Accept, so $\mathcal{B}$ returns (m^*, σ^*) and wins the EUF-CMA game associated to Σ. Otherwise, $\mathbf{t}^*$ is a nonzero element of $\mathcal{K}$, so $\mathcal{B}$ returns $\mathbf{t}^*$ and finds a solution to $\mathsf{SEGP}(\mathcal{S}, \mathcal{T}(\Sigma))$. □

Theorem 1 tells us that if Σ is EUF-CMA secure, then forging a signature for $\Sigma^{\mathcal{K}}_{\mathrm{comp}}$ is *at least as hard* as solving the $\mathsf{SEGP}(\mathcal{S}, \mathcal{T}(\Sigma))$ instance corresponding to $\mathcal{K}$. We can therefore choose secure parameters for Σ_{comp} by choosing the set $\mathcal{S}$ of compression keys such that random instances of $\mathsf{SEGP}(\mathcal{S}, \mathcal{T}(\Sigma))$ are hard.

3.4 The hardness of SEGP

The hardness of $\mathsf{SEGP}(\mathcal{S}, \mathcal{T})$ depends on the choice of $\mathcal{S}$ and $\mathcal{T}$, and also on the properties of $\mathcal{R}$ and $\mathcal{M}$ (and $\mathcal{M}/\mathcal{K}$ for $\mathcal{K}$ in $\mathcal{S}$). There are a few general things that we can say before returning to the problem in the concrete cases of Squirrels and Wave later. Let $\mathcal{O}$ be a membership oracle for a secret $\mathcal{K}$ sampled uniformly random from $\mathcal{S}$, and accepting only queries from $\mathcal{T}$; and let $\phi : \mathcal{M} \to \mathcal{M}/\mathcal{K}$ be the quotient homomorphism. We can assume $\phi(\mathcal{T}) = \mathcal{M}/\mathcal{K}$. The adversary's goal is to find some $\mathbf{t} \neq \mathbf{0}$ in $\mathcal{T}$ such that $\mathcal{O}(\mathbf{t}) = \mathsf{True}$: implicitly, $\mathbf{t} \in (\mathcal{K} \setminus \{\mathbf{0}\}) \cap \mathcal{T}$.

The adversary makes a series of adaptive queries $\mathbf{t}^{(1)}, \mathbf{t}^{(2)}, \ldots$ to $\mathcal{O}$. Suppose $\mathcal{O}(\mathbf{t}^{(i)}) = \mathsf{False}$ for $1 \leq i \leq Q$. The adversary wants $\phi(\mathbf{t}^{(Q+1)}) \neq \phi(\mathbf{t}^{(i)})$ for $1 \leq i \leq Q$, because none of the $\phi(\mathbf{t}^{(i)}$ were $\mathbf{0}$. In the best case for the adversary all these values are distinct,[3] so if the adversary chooses $\mathbf{t}^{(Q+1)}$ arbitrarily then

$$P\big[\mathcal{O}(\mathbf{t}^{(Q+1)})\big] \leq \frac{1}{\#(\mathcal{M}/\mathcal{K}) - Q}\,.$$

(If the adversary can choose $\mathbf{t}^{(Q+1)}$ such that it maps into a proper submodule $\mathcal{N} \subset \mathcal{M}/\mathcal{K}$ then we can replace $\#(\mathcal{M}/\mathcal{K})$ with $\#\mathcal{N}$ and Q with the number of prior queries landing in $\mathcal{N}$, but this is not significant in our applications.)

In the meantime, the adversary has also learned that

$$\mathcal{K} \notin \bigcup_{i=1}^{Q} \mathcal{S}_{\mathbf{t}^{(i)}} \quad \text{where} \quad \mathcal{S}_{\mathbf{t}} := \{\mathcal{K} \in \mathcal{S} \mid \mathbf{t} \in \mathcal{K}\} \subset \mathcal{S}\,.$$

[3] Note that collisions $\phi(\mathbf{t}^{(i)}) = \phi(\mathbf{t}^{(j)})$ are not useful to the adversary, because they cannot detect them without querying $\mathcal{O}$ on all the $\mathbf{t}^{(i)} - \mathbf{t}^{(j)}$.

In our applications $\mathcal{S}$ is finite, so there exists an integer

$$\kappa_{\mathcal{T}} := \max\{\#\mathcal{S}_{\mathbf{t}} : \mathbf{t} \in \mathcal{T}\}\,;$$

each unsuccessful query eliminates up to $\kappa_{\mathcal{T}}$ candidate kernels from consideration. The adversary must choose $\mathbf{t}^{(Q+1)}$ such that $\mathcal{S}_{\mathbf{t}^{(Q+1)}} \not\subset \bigcup_{i=1}^{Q} \mathcal{S}_{\mathbf{t}^{(i)}}$ to have any chance of success. If $\mathbf{t}^{(Q+1)}$ is chosen arbitrarily among the elements of $\mathcal{T}$ such that $\mathcal{S}_{\mathbf{t}^{(Q+1)}} \setminus \bigcup_{i=1}^{Q} \mathcal{S}_{\mathbf{t}^{(i)}}$ is maximal, then the probability of success is

$$P\big[\mathcal{O}(\mathbf{t}^{(Q+1)})\big] \le \frac{\#\big(\mathcal{S}_{\mathbf{t}^{(Q+1)}} \setminus \bigcup_{i=1}^{Q} \mathcal{S}_{\mathbf{t}^{(i)}}\big)}{\#\mathcal{S} - \#\bigcup_{i=1}^{Q} \mathcal{S}_{\mathbf{t}^{(i)}}} \le \frac{\kappa_{\mathcal{T}}}{\#\mathcal{S} - \kappa_{\mathcal{T}} Q}\,. \tag{10}$$

For EUF-CMA security with parameter μ against this adversary, we need to ensure the success probability after Q queries is $\le 2^{-\mu}$, so

$$\min(\#\mathcal{S}/\kappa_{\mathcal{T}}, \#(\mathcal{M}/\mathcal{K})) \ge 2^{\mu} + Q\,. \tag{11}$$

We should therefore sample $\mathcal{K}$ from an $\mathcal{S}$ chosen such that (at least)

$$\#\mathcal{S}/\kappa_{\mathcal{T}} \ge 2^{\mu} \qquad \text{and} \qquad \#(\mathcal{M}/\mathcal{K}) \ge 2^{\mu} \text{ for each } \mathcal{K} \in \mathcal{S}\,, \tag{12}$$

and we should replace the verification key when the number of (failed) compressed verifications approaches 2^{μ}.

Heuristically, we will suppose that the adversary cannot improve on any strategy that simply enumerates queries $\mathbf{t}^{(i+1)}$ while maximising $\#\mathcal{S}_{\mathbf{t}^{(i+1)}} \setminus \bigcup_{j=1}^{i} \mathcal{S}_{\mathbf{t}^{(j)}}$. Indeed, to do so they would need more information on $\phi(\mathbf{x})$ for unqueried $\mathbf{x}$. Our heuristic is that this information has to come from exploiting the $\mathcal{R}$-module structures of $\mathcal{M}$ and $\mathcal{M}/\mathcal{K}$, but these structures generally do not help.

First, $(\phi(\mathbf{x}) \ne \mathbf{0}) \wedge (\phi(\mathbf{y}) \ne \mathbf{0}) \;\not\Longrightarrow\; \phi(\mathbf{x}+\mathbf{y}) \ne \mathbf{0}$ unless $\mathbf{y} \in \mathcal{R}\mathbf{x}$ or $\mathbf{x} \in \mathcal{R}\mathbf{y}$. This tells us that given the results of $\mathcal{O}(\mathbf{t}^{(i)})$ for $1 \le i \le Q$, we cannot predict the result of $\mathcal{O}(\mathbf{t}^{(Q+1)})$ for general linear combinations $\mathbf{t}^{(Q+1)} = \sum_{i=1}^{Q} \alpha_i \mathbf{t}^{(i)}$.

Looking at scalar multiplication, there are two cases.

1. If $\alpha \ne 0 \in \mathcal{R}$ is invertible on $\mathcal{M}/\mathcal{K}$, then $\phi(\mathbf{x}) = \mathbf{0} \iff \phi(\alpha\mathbf{x}) = \mathbf{0}$ for all $\mathbf{x}$. In this case, if we know $\mathcal{O}(\mathbf{x}) = \mathsf{False}$, then we can predict that $\mathcal{O}(\alpha\mathbf{x}) = \mathsf{False}$ *without* querying $\mathcal{O}$ on $\alpha\mathbf{x}$ (and vice versa). But these "virtual" queries cannot not help the adversary, because $\mathcal{S}_{\alpha\mathbf{x}} = \mathcal{S}_{\mathbf{x}}$ for all such α.
2. If $\alpha \ne 0 \in \mathcal{R}$ is *not* invertible on $\mathcal{M}/\mathcal{K}$, then $\phi(\mathbf{x}) = \mathbf{0} \implies \phi(\alpha\mathbf{x}) = \mathbf{0}$ for all $\mathbf{x}$, but the converse does not hold; likewise, $\mathcal{S}_{\mathbf{x}} \subset \mathcal{S}_{\alpha\mathbf{x}}$ but the inclusion may be strict. If such elements α are known, then the adversary should query on $\alpha\mathbf{x}$ instead of $\mathbf{x}$ to maximise $\#\mathcal{S}_{\alpha\mathbf{x}}$ (and hence $\#\mathcal{S}_{\mathbf{t}^{(i+1)}} \setminus \bigcup_{j=1}^{i} \mathcal{S}_{\mathbf{t}^{(j)}}$). If these α exist but are *not* known to the adversary, then they should try to guess them in order to approach the ideal bound of (10).

In our application to Wave, $\mathcal{R} = \mathbb{F}_3$ is a field, so we are always in the first situation. For Rabin–Williams and Squirrels, $\mathcal{R} = \mathbb{Z}$ and $\mathcal{M}/\mathcal{K} = \mathbb{Z}/d\mathbb{Z}$ for some d unknown to the adversary. Every query is $\alpha \cdot 1$ for some α, and the adversary's goal is precisely to find $\alpha \ne 0$ divisible by the unknown d: that is, they are (or are trying to be) in the second situation.

Remark 3. The bounds in (12) may be overly pessimistic: even if $\mathsf{SEGP}(\mathcal{S}, \mathcal{T}(\Sigma))$ is hard, forging signatures in $\Sigma_{\text{comp}}^{\mathcal{K}}$ for random $\mathcal{K}$ in $\mathcal{S}$ may be significantly harder. With GPV-style signatures, a solution $\mathbf{t}$ to the $\mathsf{SEGP}(\mathcal{S}, \mathcal{T}(\Sigma))$ instance for $\mathcal{K}$ gives a forgery against $\Sigma_{\text{comp}}^{\mathcal{K}}$ only if we can construct $(m, \sigma = (\mathsf{salt}, \mathbf{s}))$ mapping to $\mathbf{t}$; and this is made difficult by the need to satisfy $\textsc{Constraint}(\mathbf{s})$. After all, if we could find (m, σ) for arbitrary $\mathbf{t}$ then we could find them for $\mathbf{0}$, and thus construct forgeries in the original scheme Σ. For a computationally bounded adversary, it may be infeasible to construct (m, σ) yielding implicit queries $\mathbf{t}$ with large $\#\mathcal{S}_{\mathbf{t}}$, which means the success probability is actually much lower—and then we can make $\mathcal{S}$ (and thus, potentially, $|\mathsf{VK}|$) much smaller. Hence, while setting parameters to make SEGP hard will guarantee unforgeability, these parameters may also be much larger than what is required for EUF-CMA in practice.

4 Compressed verification for Squirrels

Now we turn our attention to Squirrels. The challenge here is to define compressed verification algorithms that, like Squirrels, avoid multiprecision arithmetic. To simplify presentation, we use the following notation:

Definition 3. *Given a list of primes* $\mathbf{m} = (m_1, \ldots, m_s)$*, we write*

$$[[x]]_{\mathbf{m}} := \big(x \bmod m_1, \ldots, x \bmod m_s\big) \quad \textit{for all } x \in \mathbb{Z}\,.$$

4.1 The Squirrels signature scheme

Squirrels is a GPV signature on unstructured lattices. More precisely, it uses co-cyclic lattices: n-dimensional lattices $\mathcal{L}$ such that $\mathbb{Z}^n/\mathcal{L} = \mathbb{Z}/\Delta\mathbb{Z}$ for some Δ. Co-cyclic lattices are dominant among full-rank integer lattices [22]: their natural density is $\approx 85\%$. Squirrels works with co-cyclic lattices $\mathcal{L}$ of determinant

$$\Delta := p_1 \cdots p_s$$

where $\mathbf{p} = (p_1, \ldots, p_s)$ is a fixed tuple of 31-bit primes (the length s depends on the security parameter). Table 2 gives the Squirrels parameter sets.

Squirrels is built on the one-way function

$$\begin{aligned} f : D_n &\longrightarrow \mathbb{Z}/\Delta\mathbb{Z} \\ \mathbf{x} &\longmapsto \mathbf{x}\mathbf{A}^T \pmod{\Delta}, \end{aligned} \tag{13}$$

where $\mathbf{A}$ is the matrix defining $\mathcal{L}$ and $D_n = \{\mathbf{e} \in \mathbb{Z}^n \mid \|\mathbf{e}\| \leq \beta\}$. One-wayness depends on the hardness of $GSIS_{n,\Delta,\beta}$, that is, finding a vector $\mathbf{x}$ such that $\mathbf{x}\mathbf{A}^T \equiv 0 \pmod{\Delta}$ and $\|\mathbf{x}\| \leq \beta$ for some small β.

Suppose $\mathcal{L}$ is co-cyclic of dimension n and determinant Δ. We can specify $\mathcal{L}$ with the row-HNF of its generating matrix, which is determined by a vector

$$\mathbf{v}_{\text{check}} = (\mathbf{v}_{\text{check},1}, \ldots, \mathbf{v}_{\text{check},n}) \in (\mathbb{Z}/\Delta\mathbb{Z})^n \quad \text{with} \quad \mathbf{v}_{\text{check},n} = -1\,.$$

Table 2. Parameters for SQUIRRELS instances.

NIST Security Level	1	2	3	4	5
Lattice dimension n	1034	1164	1556	1718	2056
Hash space size q	4096	4096	4096	4096	4096
Max. signature square norm $\lfloor\beta^2\rfloor$	2 026 590	2 442 439	4 512 242	3 659 372	5 370 115
Number s of small primes	165	188	262	275	339
Bitlength of Δ	5048	5738	8017	8402	10347
Signature Size (B)	1019	1147	1554	1676	2025
Public Key Size (B)	681 780	874 576	1 629 640	1 888 700	2 786 580

The public key encodes $\mathbf{v}_{\text{check}}$ as a list of lists of residues mod the small primes:

$$\mathsf{PK} = \left((v_{i,j} := \mathbf{v}_{\text{check},i} \bmod p_j)_{i=1}^{n-1}\right)_{j=1}^{s}$$

(since $\mathbf{v}_{\text{check},n} = -1$ by convention, there is no need to store it or any of the $v_{n,j}$). The $v_{i,j}$ are encoded as *signed* twos-complement 32-bit integers, but they are all non-negative (except the $v_{n,j}$, which are all -1 and not stored anyway); in particular, $0 \le v_{i,j} < 2^{31}$ for all $1 \le i < n$ and $1 \le j \le s$.

The private key encodes a "good" basis for $\mathcal{L}$, which allows sampling short vectors in $\mathcal{L}$ following a Gaussian distribution using Klein's trapdoor sampler. `KeyGen` ensures that each $\mathbf{v}_{\text{check},i}$ looks like a uniform random integer modulo Δ.

We now focus on SQUIRRELS verification (`KeyGen` and `Sign` are detailed in [11]). The verifier accepts $\sigma = (\mathbf{s}, \mathsf{salt})$ if two conditions are met:

1. The vector $\mathbf{s}$ is short: $\|\mathbf{s}\| \le \beta$, which is more easily checked as
$$\textsc{Constraint}(\mathbf{s}) : \|\mathbf{s}\|^2 \le \lfloor\beta^2\rfloor\,.$$
2. The vector $\mathbf{c} := \mathbf{s} + \mathbf{h}$ (where $\mathbf{h} := \mathsf{Hash}(\mathsf{salt} \parallel \mathbf{m})$) is in $\mathcal{L}$. That is,
$$\sum_{i=1}^{n} c_i \mathbf{v}_{\text{check},i} \equiv 0 \pmod{\Delta}\,, \quad (14)$$
or equivalently (by the CRT)
$$\sum_{i=1}^{n} c_i v_{i,j} \equiv 0 \pmod{p_j} \qquad \text{for all } 1 \le j \le s\,. \quad (15)$$

We can thus verify by checking (15) for each of the p_j in turn, as in Algorithm 1.

Algorithm 1: Verification algorithm for SQUIRRELS.

Parameters : q, n, $\lfloor\beta^2\rfloor$, and $P_\Delta = (p_1, \ldots, p_m)$
Input: Signature $\sigma = (\mathsf{salt}, \underline{\mathbf{s}})$, message m, public key $\mathsf{PK} = ((v_{i,j})_{i=1}^{n-1})_{j=1}^{m}$ with $v_{i,j} = \mathbf{v}_{\text{check},i} \bmod p_j$
Output: Accept if σ is a valid signature on m under PK, otherwise Reject.

$\mathbf{s} \leftarrow \mathsf{Decompress}(\underline{\mathbf{s}})$
if $\mathbf{s} = \perp$ ***or*** $\|\mathbf{s}\|_2^2 > \lfloor\beta^2\rfloor$ **then**
  return Reject
$\mathbf{c} \leftarrow \mathbf{s} + \mathsf{HashToPoint}(m \parallel \mathsf{salt}, q, n)$
foreach $1 \le j \le m$ **do** `// Trivially parallelizable`
  $S \leftarrow \sum_{i=0}^{n-1} c_i v_{i,j} \bmod p_j$
  if $S - c_n \ne 0 \bmod p_j$ **then** `// Uses` $\mathbf{v}_{\text{check},n} = -1$
    return Reject
return Accept

4.2 Homomorphisms for Squirrels verification

Squirrels is an instance of our general framework with $\mathcal{R} = \mathbb{Z}$ and $\mathcal{M} = \mathbb{Z}/\Delta\mathbb{Z}$. But the only homomorphisms from $\mathbb{Z}/\Delta\mathbb{Z}$ map through $\mathbb{Z}/\Delta'\mathbb{Z}$ for $\Delta'|\Delta$, and while a verifier could choose a secret ϕ by choosing a secret subset of the p_j, an adversary who could forge a signature for a large subset of the p_j would fool many verifiers. There are many more homomorphisms from $\mathbb{Z}$, and we can lift Squirrels trivially to $\mathcal{M} = \mathbb{Z}$ if we replace (14) with the equivalent condition

$$\sum\nolimits_{i=1}^{n} c_i \cdot \mathbf{v}_{\mathrm{check},i} = k\Delta \qquad \text{for some integer } k\,. \tag{16}$$

Let the verifier choose a secret list of secret 31-bit primes $\mathbf{r} = (r_1, \ldots, r_t)$, each prime to Δ. The parameter t is a function of the desired verification security level, to be determined later in §4.5. Now the verifier could check

$$\sum\nolimits_{i=1}^{n} c_i \cdot (\mathbf{v}_{\mathrm{check},i} \bmod r_j) \equiv k(\Delta \bmod r_j) \pmod{r_j} \quad \text{for } 1 \le j \le t \tag{17}$$

—but k is not included in the signature, and re-computing it is the same computation as a full Squirrels verification. Instead, we will verify by implicitly recovering k modulo $\prod_j r_j$, and checking that it is in the appropriate range.

First, we need to compute each of the $[[\mathbf{v}_{\mathrm{check},i}]]_{\mathbf{r}}$ from the $[[\mathbf{v}_{\mathrm{check},i}]]_{\mathbf{p}} = (v_{i,j} = \mathbf{v}_{\mathrm{check},i} \bmod p_j)_{j=1}^{s}$. In the spirit of Squirrels, we want to avoid multiprecision integer arithmetic, so we need to compute the $[[\mathbf{v}_{\mathrm{check},i}]]_{\mathbf{r}}$ *without reconstructing* any of the integers $\mathbf{v}_{\mathrm{check},i}$. Our main tool is the *explicit* CRT.[4]

Definition 4. *With the notation above: for each $1 \le i \le s$, we write Δ_i for Δ/p_i, and let q_i be the the unique integer in $(1, p_i)$ such that $q_i\Delta_i \equiv 1 \pmod{p_i}$.*

We will never explicitly compute with the Δ_i (they are a notational convenience). We *will* need the q_i, and these can be precomputed in advance (using e.g. Algorithm 7 in Appendix A). Each is a positive 31-bit integer.

Lemma 1 (Explicit CRT). *If $0 \le x < \Delta$ and $(x_1, \ldots, x_s) = [[x]]_{\mathbf{p}}$, then*

$$x = \alpha\Delta - \lfloor\alpha\rfloor\Delta \quad \textit{where} \quad \alpha = \sum\nolimits_{i=1}^{s} x_i q_i / p_i\,. \tag{18}$$

Proof. Observe that α is a rational number, but $\alpha\Delta$ is an integer. The CRT says $x \equiv \alpha\Delta \pmod{\Delta}$, so obviously $\alpha\Delta - \lfloor\alpha\rfloor\Delta \equiv x \pmod{\Delta}$. But $0 \le \alpha - \lfloor\alpha\rfloor < 1$ by construction, so $0 \le \alpha\Delta - \lfloor\alpha\rfloor\Delta < \Delta$, so $\alpha\Delta - \lfloor\alpha\rfloor\Delta = x$. □

Lemma 1 gives an exact expression for $0 \le x < \Delta$ in terms of $[[x]]_{\mathbf{p}}$ that we can use to compute $[[x]]_{\mathbf{r}}$ by computing the integers $\alpha\Delta$ and $\lfloor\alpha\rfloor\Delta$ modulo each r_j. We precompute the q_i, $[[\Delta_i]]_{\mathbf{r}}$, and $[[\Delta]]_{\mathbf{r}}$. Computing $\alpha\Delta = \sum_i x_i q_i \Delta_i$ modulo r_j is straightforward. The interesting part is determining $\lfloor\alpha\rfloor$, and thus computing $\lfloor\alpha\rfloor \bmod r_j$, without computing α. We will do this using fixed-point approximations, as in [4]. Lemma 2, an adaptation of [4, Lemma 3.1], shows that with a relatively low precision we can determine $\lfloor\alpha\rfloor$ up to a possible error of 1.

[4] This is similar to the modular reduction in RNS arithmetic in [4], but there, $\mathbf{p}$ can be freely chosen to optimise computations on the operands; here, $\mathbf{p}$ is fixed.

Lemma 2. *Let* $\alpha_1, \ldots, \alpha_s$ *be non-negative real numbers, and set* $\alpha := \sum_{j=1}^{s} \alpha_j$. *Fix some integer* $a \geq \log_2 s + 1$. *Then*

$$f := \left\lfloor \frac{s}{2^a} + \frac{1}{2^a} \sum\nolimits_{j=1}^{s} \lfloor 2^a \alpha_j \rfloor \right\rfloor \quad \textit{is either } \lfloor \alpha \rfloor \textit{ or } \lfloor \alpha \rfloor + 1\,.$$

Further, if $\alpha - \lfloor \alpha \rfloor < 1 - s/2^a$ *then* f *is exactly* $\lfloor \alpha \rfloor$.

Proof. Note that $2^a \geq 2s$, so $s/2^a \leq 1/2$. Let $q := (1/2^a)\sum_j \lfloor 2^a \alpha_j \rfloor$. By construction, $0 \leq 2^a \alpha_j - \lfloor 2^a \alpha_j \rfloor < 1$ for each $1 \leq j \leq s$. Summing over j gives $0 \leq 2^a \alpha - 2^a q < s$, so $0 \leq \alpha - q < s/2^a$; so $\lfloor \alpha \rfloor < s/2^a + q \leq s/2^a + \alpha$. Taking floors gives $\lfloor \alpha \rfloor \leq f \leq \lfloor s/2^a + \alpha \rfloor$. But $\lfloor s/2^a + \alpha \rfloor$ is either $\lfloor \alpha \rfloor$ or $\lfloor \alpha \rfloor + 1$, because $0 < s/2^a \leq 1/2$, proving the first statement. For the second, if $\alpha - \lfloor \alpha \rfloor < 1 - s/2^a$ then $\alpha + s/2^a < \lfloor \alpha \rfloor + 1$, so $\lfloor \alpha + s/2^a \rfloor = \lfloor \alpha \rfloor$, and thus $f = \lfloor \alpha \rfloor$. □

Theorem 2. *Fix an integer* $a \geq \log_2(s) + 1$. *Given* $\mathbf{r}$, $[[\Delta]]_{\mathbf{r}}$, $([[\Delta_i]]_{\mathbf{r}})_{i=1}^{s}$, *and* $[[x]]_{\mathbf{p}}$ *for some* $0 \leq x < \Delta$, *Algorithm 2 returns* $[[z]]_{\mathbf{r}}$ *where* z *is either* x *or* $x - \Delta$. *Further: if* $x < (1 - s/2^a)\Delta$, *then* $z = x$.

Proof. Algorithm 2 evaluates (18) modulo r_j for $1 \leq j \leq t$, computing the floor using Lemma 2 with $\alpha_i = x_i q_i / p_i$ for $1 \leq i \leq s$. □

Algorithm 2: Explicit CRT: computing $[[x]]_{\mathbf{r}}$ or $[[x - \Delta]]_{\mathbf{r}}$ from $[[x]]_{\mathbf{p}}$

Parameters : Squirrels prime vector $\mathbf{p} = (p_1, \ldots, p_s)$ and CRT coefficients $(q_1, \ldots, q_s)$; an integer $a \geq \log_2 s + 1$ (fixed-point precision)
Input: Secret prime list $\mathbf{r}$; $[[\Delta]]_{\mathbf{r}}$, $([[\Delta_i]]_{\mathbf{r}})_{i=1}^{s}$, and $[[x]]_{\mathbf{p}}$ for $0 \leq x < \Delta$
Output: $[[z]]_{\mathbf{r}}$, where $z = x$ or $x - \Delta$ (if $x < (1 - s/2^a)\Delta$, then $z = x$)

1 **function** ModECRT($\mathbf{r}$, $[[\Delta]]_{\mathbf{r}}$, $([[\Delta_i]]_{\mathbf{r}})_{i=1}^{s}$, $[[x]]_{\mathbf{p}}$)
2     $(z_1, \ldots, z_t) \leftarrow (0, \ldots, 0)$
3     $f \leftarrow s$
4   **foreach** $1 \leq j \leq s$ **do**
5         $y_j \leftarrow x_j \cdot q_j$
6         $f \leftarrow f + \lfloor 2^a y_j / p_j \rfloor$ // See Remark 5
7   **foreach** $1 \leq k \leq t$ **do**
8             $y_{j,k} \leftarrow y_j \bmod r_k$
9             $z_k \leftarrow (z_k + y_{j,k} \cdot \Delta_{j,k}) \bmod r_k$
10     $f \leftarrow \lfloor f / 2^a \rfloor$ // Right-shift f by a
11   **foreach** $1 \leq k \leq t$ **do**
12         $z_k \leftarrow (z_k - f \cdot \Delta_k) \bmod r_k$
13   **return** $(z_1, \ldots, z_t)$

Remark 4. Theorem 2 recovers $[[x]]_{\mathbf{r}}$ or $[[x - \Delta]]_{\mathbf{r}}$ from $[[x]]_{\mathbf{p}}$. To guarantee a result of $[[x]]_{\mathbf{r}}$ requires increasing the precision to $a \sim \log_2 \Delta$, which means working with integers the size of Δ; but then we may as well reconstruct x.

Remark 5. As noted in [4], the floors in Line 6 of Algorithm 2 can be computed by repeatedly doubling y_j modulo p_j and counting overflows.

4.3 Compressed verification for Squirrels

Recall that our goal is to verify (16) by evaluating (17), namely

$$\sum\nolimits_{i=1}^{n} c_i \cdot \mathbf{v}_{\mathrm{check},i} \equiv k\Delta \pmod{r_j} \quad \text{for each } 1 \leq j \leq t\,,$$

but *without* knowing k. Given a public key PK and a compression key $\mathsf{CK} := \left(\mathbf{r}, ([[\Delta_i]]_{\mathbf{r}})_{i=1}^{s}, [[\Delta]]_{\mathbf{r}}, (I_1, \ldots, I_t)\right)$, for each $1 \leq i \leq n$ we use ModECRT to compute

$$\begin{aligned}[[\bar{v}_i]]_{\mathbf{r}} &:= \mathtt{ModECRT}(\mathbf{r}, [[\Delta]]_{\mathbf{r}}, ([[\Delta_j]]_{\mathbf{r}})_{j=1}^{s}, [[\mathbf{v}_{\mathrm{check},i}]]_{\mathbf{p}} = (v_{i,j})_{j=1}^{s}) \\ &= [[\mathbf{v}_{\mathrm{check},i} - \epsilon_i \Delta]]_{\mathbf{r}} \quad \text{where } \epsilon_i \in \{0,1\} \text{ is unknown.}\end{aligned}$$

The r_j are chosen such that $r_j \nmid \Delta$, so if we let

$$k'_j := \left(\sum\nolimits_{i=1}^{n} c_i \bar{v}_{i,j}\right) I_j \bmod r_j \quad \text{where} \quad I_j := \Delta^{-1} \bmod r_j \quad \text{for } 1 \leq j \leq t\,,$$

then the system of verification equations (17) becomes

$$k'_j \equiv k' \pmod{r_j} \quad \text{where} \quad k' := k + \sum\nolimits_{i=1}^{n-1} \epsilon_i c_i\,. \tag{19}$$

Lemma 3 below shows that

- If (16) holds, then k' is a small integer (Table 3 gives bounds on k' for each SQUIRRELS instance), small enough that $k'_j = k'$ *as an integer* for each j.
- If (16) does not hold, then k' does not exist, and the k'_j look like random (and generally large) values modulo each of the r_j.

This distinction is the basis of our CVerify for SQUIRRELS: we compute $(k'_1, \ldots, k'_t)$, and Accept if $k'_1 = \cdots = k'_t$ and k'_1 is within the bounds on k'; otherwise, we Reject. Note that even if the adversary has full control over the c_i, they cannot control the k'_j, because $\mathbf{r}$ and the I_j are unknown.

Lemma 3. *With SQUIRRELS parameters $q \ll \Delta$ and $\lfloor \beta^2 \rfloor \ll \Delta$: if $\|\mathbf{s}\|_2^2 \leq \lfloor \beta^2 \rfloor$, then the integer k' of (19) satisfies*

$$k'_{\min} \leq k' \leq k'_{\max} \quad \textit{where} \begin{cases} k'_{\min} := -\left\lfloor 2\sqrt{n\lfloor \beta^2 \rfloor} \right\rfloor - 1\,, \\ k'_{\max} := 2(n-1)(q-1) + \left\lfloor 2\sqrt{n\lfloor \beta^2 \rfloor} \right\rfloor + 1\,. \end{cases}$$

Proof. By definition,

$$k' = \left(\sum_{i=1}^{n-1} c_i \beta_i\right) - \frac{c_n}{\Delta} \quad \text{with} \quad \beta_i = \frac{\mathbf{v}_{\mathrm{check},i}}{\Delta} + \epsilon_i$$

for $1 \leq i < n$, so

$$k' + E = H + S$$

where

$$H = \sum_{i=1}^{n-1} h_i \beta_i, \quad S = \sum_{i=1}^{n-1} s_i \beta_i, \quad E = \frac{c_n}{\Delta}.$$

For $1 \leq i < n$, we have $0 \leq \mathbf{v}_{\mathrm{check},i} < \Delta$, so $0 \leq \beta_i < 2$.
Thus,

$$-S' - E < k' < H' + S' + E$$

where

$$H' = 2\sum_{i=1}^{n-1} h_i \quad \text{and} \quad S' = 2\sum_{i=1}^{n} |s_i|$$

(note: we include $|s_n|$ in S').

But $0 \leq |E| \leq \frac{h_n + |s_n|}{\Delta} < 1$ because

$$h_n < q \ll \Delta \quad \text{and} \quad |s_n| \leq \lfloor \beta^2 \rfloor \ll \Delta,$$

so

$$-S' - 1 < k' < H' + S' + 1.$$

Now,

$$0 \leq S' \leq 2\sqrt{n}\|\mathbf{s}\|_2 \leq 2\sqrt{n\lfloor\beta^2\rfloor}, \quad 0 \leq H' \leq 2(n-1)(q-1)$$

and k' is an integer, so

$$\left\lceil -2\sqrt{n\lfloor\beta^2\rfloor} \right\rceil - 1 \leq k' \leq 2(n-1)(q-1) + \left\lfloor 2\sqrt{n\lfloor\beta^2\rfloor} \right\rfloor + 1,$$

and the result follows. □

Table 3. Values of $k'_{\min}$ and $k'_{\max}$ from Lemma 3 for the Squirrels instances in [11]. Note that $k'_{\max} - k'_{\min}$ is a 24-bit integer except for Squirrels-V, where it is 25 bits.

Instance	Squirrels-I	Squirrels-II	Squirrels-III	Squirrels-IV	Squirrels-V
$k'_{\min}$	-91554	-106640	-144446	-15879	-210152
$k'_{\max}$	8551824	9631610	9603896	14220809	17040602

The verification key VK is $\left(\mathbf{r}, (I_1, \ldots, I_t), ([[\bar{v}_i]]_{\mathbf{r}})_{i=1}^{n-1}\right)$, so

$$|\mathsf{VK}| = 4(n+1)t \text{ bytes} \quad \text{and} \quad |\mathsf{PK}| = 4(n-1)s \text{ bytes}.$$

The key-compression ratio is $|\mathsf{PK}| : |\mathsf{VK}| \approx s : t$. See Table 4 for sample values.

The compression key CK requires $4(s+3)t$ bytes. The values of $([[\Delta_i]]_{\mathbf{r}})_{i=1}^{s}$, $[[\Delta]]_{\mathbf{r}}$, and $(I_1, \ldots, I_t)$ may be left out of CK, thus reducing $|\mathsf{CK}|$ to $4s$ bytes, but this implies recomputing them from $\mathbf{r}$ and $\mathbf{p}$ for every verification key generation.

4.4 The algorithms

Algorithms 3, 4, and 5 formalise `CKeyGen`, `VKeyGen`, and `CVerify` for Squirrels.

Algorithm 3: `CKeyGen` (compression key generation) for Squirrels.

Parameters : n, s, Δ, $\mathbf{p}$, $(\Delta_1, \ldots, \Delta_s)$, $[[\Delta_i]]_{\mathbf{p}}$, t
Output: CK
Sample a list $\mathbf{r} = (r_1, \ldots, r_t)$ of random 31-bit primes // Use Lemma 4
Compute $([[\Delta_i]]_{\mathbf{r}})_{i=1}^{s}$ and $[[\Delta]]_{\mathbf{r}}$ from $\mathbf{p}$ // Using e.g. Algorithm 8 in App. A.
for $1 \leq j \leq t$ **do**
  $I_j \leftarrow \Delta^{-1} \bmod r_j$ // Use $I_j = (\Delta \bmod r_j)^{-1} \bmod r_j$
return $\mathsf{CK} := \left(\mathbf{r}, ([[\Delta_i]]_{\mathbf{r}})_{i=1}^{s}, [[\Delta]]_{\mathbf{r}}, (I_1, \ldots, I_t)\right)$

Algorithm 3 requires randomly sampling 31-bit primes, which is easy using the criteria of [23]. Recall that an odd integer $r = d2^u + 1$ is a *strong pseudoprime to the base* a if $a^d \equiv 1 \pmod{r}$ or $a^{d2^v} \equiv -1 \pmod{r}$ for some $0 \leq v < u$.

Lemma 4. *Let $2^{30} < r < 2^{31}$ be odd. If r is a strong pseudoprime to the bases 2, 3, and 5, then either r is prime or $r = 1157839381 = 24061 \cdot 48121$.*

Proof. See [23, Page 1022]. □

Algorithm 4: `VKeyGen` (verification key generation) for Squirrels.

Parameters : n, s, Δ, $\mathbf{p}$, $(\Delta_1, \ldots, \Delta_s)$, $[[\Delta_i]]_{\mathbf{p}}$, t
Input: $\mathsf{CK} = \left(\mathbf{r}, ([[\Delta_i]]_{\mathbf{r}})_{i=1}^{s}, [[\Delta]]_{\mathbf{r}}, (I_1, \ldots, I_t)\right)$, $\mathsf{PK} = (\mathbf{v}_{\mathrm{check},i})_{i=1}^{n-1}$
Output: VK

foreach $1 \leq i < n$ **do** // May be parallelized
  $[[\bar{v}_i]]_{\mathbf{r}} \leftarrow$ `ModECRT`$(\mathbf{r}, [[\Delta]]_{\mathbf{r}}, ([[\Delta_i]]_{\mathbf{r}})_{i=1}^{t}, \mathbf{v}_{check,i})$ // $=[[\mathbf{v}_{\mathrm{check},i} + \epsilon_i \Delta]]_{\mathbf{r}}$
return $(\mathbf{r}, (I_1, \ldots, I_t), (\bar{v}_{i,1}, \ldots, \bar{v}_{i,t})_{i=1}^{n-1})$

Algorithm 5 defines `CVerify`. Lines 6-13 must be implemented with constant-time techniques to ensure no information on $\mathbf{r}$ is leaked in the event of rejection[5].

Algorithm 5: `CVerify` (compressed verification) for Squirrels.

Parameters : n, s, Δ, $\mathbf{p}$, $(\Delta_1, \ldots, \Delta_s)$, $[[\Delta_i]]_{\mathbf{p}}$, t
Input: $\sigma = (\mathsf{salt}, \underline{\mathbf{s}})$, m, $\mathsf{VK} = \left(\mathbf{r}, (I_1, \ldots, I_t), (\bar{v}_{i,1}, \ldots, \bar{v}_{i,t})_{i=1}^{n}\right)$
Output: Accept or Reject

1 $\mathbf{s} \leftarrow \mathsf{Decompress}(\underline{\mathbf{s}})$
2 **if** $\mathbf{s} = \perp$ ***or*** $\|\mathbf{s}\|_2^2 > \lfloor \beta^2 \rfloor$ **then**
3   **return** Reject
4 $\mathbf{c} \leftarrow \mathbf{s} + \mathsf{HashToPoint}(m \parallel \mathsf{salt}, q, n)$
5 $(a_1, \ldots, a_t) \leftarrow (\mathsf{True}, \ldots, \mathsf{True})$
6 **foreach** $1 \leq j \leq t$ **do** // May be parallelized
7   $k'_j \leftarrow \left(\left(\sum_{i=1}^{n} c_i \bar{v}_{i,j}\right) I_j - k'_{\min}\right) \bmod r_j$ // $k'_{\min}$: see Lemma 3
8   **if** $k'_j > k'_{\max} - k'_{\min}$ **then** // $k'_{\max} - k'_{\min}$: see Lemma 3
9     $a_j \leftarrow \mathsf{False}$
10 **if** $(a_1 \wedge \cdots \wedge a_t) \wedge (k'_1 = \cdots = k'_t)$ **then**
11   **return** Accept
12 **else**
13   **return** Reject

4.5 Security argument

If Σ is Squirrels, then $\mathcal{R} = \mathcal{M} = \mathbb{Z}$. Compressed Squirrels samples $\mathcal{K}$ from

$$\mathcal{S} = \{\Delta' \mathbb{Z} : \Delta' \text{ is a product of } t \text{ 31-bit primes}\}.$$

[5] The implementation can be made constant-time by using constant-time arithmetic routines (e.g. multiplication/modular reduction [5]) and replacing any conditional branches with bit-masking operations.

Theorem 1 ensures EUF-CMA security for compressed SQUIRRELS provided $\mathsf{SEGP}(\mathcal{S}, \mathcal{T}(\Sigma))$ is hard. For SQUIRRELS parameters, $\mathcal{T}(\Sigma) \subset [0, 2q\sqrt{n}\lfloor\beta^2\Delta\rfloor] \subset [0, 2^{30}\Delta)$: at most s 31-bit primes can divide any $\mathbf{t} \in \mathcal{T}(\Sigma)$, so $\kappa_{\mathcal{T}} \approx s!/(t!(s-t)!)$. Now $\#\mathcal{S} = P_{31}!/(t!(P_{31}-t)!)$ where $P_{31} := \#\text{PRIMES}(31)$, and $\#(\mathcal{M}/\mathcal{K}) \approx 2^{31t}$. When $t \ll s$, we have $s!/(s-t)! \sim s^t$ and $P_{31}!/(P_{31}-t)! \sim P_{31}^t$. Looking at [29, Table 3], we find that

$$P_{31} := \#\text{PRIMES}(31) = 105097565 - 54400028 \approx 2^{25.6}\,.$$

The heuristic of §3.4 therefore suggests taking $t \approx \mu/(25.6 - \log_2(s))$, where μ is the targeted security level.

In reality, it is computationally infeasible to construct forgery attempts (m, σ) that yield $\mathbf{t}$ with $\#\mathcal{S}_{\mathbf{t}} \approx \kappa_{\mathcal{T}}$: this would mean constructing (m, σ) such that $\mathbf{t}$ is divisible by another product Δ' of s 31-bit primes, and this is essentially as hard as forging a full SQUIRRELS signature: that is, solving $\text{GSIS}_{n,\Delta,\beta}$. If we want an (m, σ) mapping to a $\mathbf{t}$ with k 31-bit prime factors, and thus $\#\mathcal{S}_{\mathbf{t}} = \binom{k}{t}/\#S$, we must solve a single GSIS instance with modulus $\Delta^\star \ll \Delta$. Indeed, it requires a nontrivial computational effort to even construct a forgery attempt (m, σ) yielding $\mathbf{t}$ with $\#\mathcal{S}_{\mathbf{t}} > 0$. In our security estimates, we therefore model the expected value of $\#\mathcal{S}_{\mathbf{t}}$ as a small constant, and hence we choose t such that $P_{31}!/(t!(P_{31}-t)!$ is on the order of 2^λ, which suggests taking t as in Table 4. While the resulting security level μ is slightly smaller than λ in some cases, it should be remembered that each forgery attempt requires an interaction with the verifier, and is therefore substantially more expensive than (e.g.) the AES circuit evaluations used to model post-quantum cryptographic attack costs.

Table 4. Suggested values for the number t of secret primes in $\mathbf{r}$.

Instance	Original scheme				Compressed verification				
	λ	s	$\lvert\mathsf{PK}\rvert$	$\lvert\sigma\rvert$	t	μ	$\lvert\mathsf{CK}\rvert$	$\lvert\mathsf{VK}\rvert$	$\lvert\mathsf{PK}\rvert : \lvert\mathsf{VK}\rvert$
SQUIRRELS-I	128	165	681780	1019	5	121.1	3360	20700	32.94
SQUIRRELS-II	128	188	874576	1147	5	121.1	3820	23300	37.54
SQUIRRELS-III	192	262	1629640	1554	8	189.5	8480	49824	32.71
SQUIRRELS-IV	192	275	1888700	1676	8	189.5	8896	55008	34.34
SQUIRRELS-V	256	339	2786580	2025	11	256.3	15048	90508	30.79

5 Compressed verification for WAVE

WAVE is a GPV-style signature based on hard problems in ternary linear codes. Very briefly: a WAVE public key is a matrix $\mathsf{PK} = \mathbf{R} \in \mathbb{F}_3^{k\times(n-k)}$ such that $\mathbf{M} = (\mathbf{I}_{n-k}|\mathbf{R})^\top$ in $\mathbb{F}_3^{n\times(n-k)}$ is a parity-check matrix for a permuted generalized $(U|U+V)$-code C; knowledge of the relation between C and the component codes U and V is a trapdoor allowing the signer to generate vectors $\mathbf{s}$ in $\mathbb{F}_3^n$ such that

$$\text{CONSTRAINT}(\mathbf{s}) \qquad \text{and} \qquad \mathbf{sM} = \mathsf{Hash}(\mathsf{salt}, m)\,, \tag{20}$$

where CONSTRAINT($\mathbf{s}$) is that $\mathbf{s}$ have a fixed, high weight w. An "original" WAVE signature (as in [9]) is $\sigma = (\mathsf{salt}, \mathbf{s})$. In the WAVE NIST submission [1], $\mathbf{s}$ is truncated to its last k entries (the other $n-k$ entries are implicitly recovered in verification). The special form of $\mathbf{M} = (\mathbf{I}_{n-k}|\mathbf{R})^\top$ allows us to rewrite (20) as

$$\textsc{Constraint}(\mathbf{s}) \quad \text{and} \quad \mathbf{tM} = \mathbf{0} \quad \text{where} \quad \mathbf{t} := \mathbf{s} - (\mathsf{Hash}(\mathsf{salt}, \mathbf{s})|\mathbf{0}_k)\,. \tag{21}$$

WAVE has exceptionally large public keys: for example, WAVE822, WAVE1249, and WAVE1644 require 3.5 MB, 7.5 MB, and 13 MB, respectively. This motivates the use of *compressed verification*, where the verifier privately replaces the large matrix $\mathbf{M}$ with a much smaller secret key VK derived via a compression matrix.

`VKeyGen` computes $\mathsf{VK} := (\mathbf{I}_{n-k} \mid \mathbf{R})^\top \mathbf{C}$, where $\mathbf{C}$ is the verifier's compression matrix. Only the bottom $n-k-c$ rows of VK need to be stored, yielding a total size of $c(n-c)/4$ bytes. `CVerify` then replaces the full-rank check $\mathbf{tM} = \mathbf{0}$ with a lower-dimensional test $\mathbf{t}\mathsf{VK} = \mathbf{0}$ over $\mathbb{F}_3^c$, as detailed in Algorithm 6. Compressed verification requires "original" WAVE signatures (as in [9]) and is incompatible with the truncated versions from [1]. Although full signatures can be recovered from truncated ones using the public key, doing so during verification defeats compression by reintroducing computational and storage overhead.

Algorithm 6: `CVerify` (compressed verification) for WAVE

Input: message m, signature $\sigma = (\mathsf{salt}, \mathbf{s})$, and verification key VK
Output: Accept or Reject

if $\mathsf{Weight}(\mathbf{s}) \neq W$ **then**
  return Reject
$\mathbf{t} \leftarrow \mathbf{s} - (\mathsf{Hash}(\mathsf{salt} \parallel m) \parallel \mathbf{0}_k)$
$\mathbf{r} \leftarrow \mathbf{t}\mathsf{VK}$
if $\mathbf{r} \neq 0$ **then**
  return Reject
return Accept

The security argument for compressed WAVE is particularly simple. Theorem 1 ensures that $\Sigma_{\text{comp}}^{\mathcal{K}}$ is EUF-CMA secure if $\mathsf{SEGP}(\mathcal{S}, \mathcal{T}(\Sigma))$ is hard, where $\mathcal{S}$ is the set of codimension-c subspaces of $\mathcal{M} = \mathbb{F}_3^{n-k}$ and $\mathcal{T}(\Sigma) = \mathcal{M}$. We have $\#\mathcal{S} = \begin{bmatrix} n-k \\ n-k-c \end{bmatrix}_3$, where $\begin{bmatrix} \cdot \\ \cdot \end{bmatrix}_3$ is the 3-binomial coefficient, $\#(\mathcal{M}/\mathcal{K}) = 3^{n-k-c}$, and $\#\mathcal{S}_{\mathbf{t}} = \kappa_{\mathcal{T}(\Sigma)} = \begin{bmatrix} n-k-1 \\ n-k-c-1 \end{bmatrix}_3$ for *every* $\mathbf{t} \neq \mathbf{0} \in \mathcal{M}$. This tells us that a naive adversary that simply tries random forgery attempts (m, σ) is already close to optimal. Intuitively: since $\mathbf{C}$ is a random projection, if $\mathbf{t} \neq \mathbf{0}$ then $\mathbf{tC}$ is a random element of $\mathbb{F}_3^c$, which is $\mathbf{0}$ (leading to an Accept) with probability $1/3^c$.

Indeed, if we admit the heuristic of §3.4 then we can take $2^\mu \approx 3^c$: that is, $c \approx \log_3(2)\mu$. Taking c to be a multiple of 8 simplifies implementation. Table 5 lists suggested values of c and corresponding sizes of CK and VK for WAVE822, WAVE1249, and WAVE1644.

Table 5. Parameters for (compressed) Wave. Sizes are in bytes. Wave signatures are variable-length: the values of $|\sigma|$ here are upper bounds, and $|\sigma|$ roughly doubles for the "original" (non-truncated) signatures required for compressed verification.

Instance	Security level NIST PQ	λ	Original scheme $\|\sigma\|$	$\|\mathsf{PK}\|$	Compressed verification c	μ	$\|\mathsf{CK}\|$	$\|\mathsf{VK}\|$	$\|\mathsf{PK}\| : \|\mathsf{VK}\|$
Wave822	1	128	822	3 677 390	80	126.8	83 822	171 594	21.4
Wave1249	3	192	1249	7 867 598	120	190.2	183 134	266 188	29.6
Wave1644	5	256	1644	13 632 308	160	253.6	321 507	370 436	36.8

References

1. G. Banegas, K. Carrier, A. Chailloux, A. Couvreur, T. Debris-Alazard, P. Gaborit, P. Karpman, J. Loyer, R. Niederhagen, N. Sendrier, B. Smith, and J.-P. Tillich. WAVE – submission to NIST post-quantum signatures project, 2023. https://wave-sign.org/.
2. D. J. Bernstein. RSA signatures and Rabin–Williams signatures: the state of the art. 2008.
3. D. J. Bernstein, A. Hülsing, S. Kölbl, R. Niederhagen, J. Rijneveld, and P. Schwabe. The SPHINCS+ signature framework. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 2129–2146. Association for Computing Machinery, 2019.
4. D. J. Bernstein and J. P. Sorenson. Modular exponentiation via the explicit chinese remainder theorem. *Math. Comput.*, 76(257):443–454, 2007.
5. J. W. Bos, T. Kleinjung, and D. Page. Efficient modular multiplication. Cryptology ePrint Archive, Paper 2021/1151, 2021.
6. C. Boschini, D. Fiore, E. Pagnin, L. Torresetti, and A. Visconti. Progressive and efficient verification for digital signatures: extensions and experimental results. *J. Cryptogr. Eng.*, 14(3):551–575, 2024.
7. Y. Chen, N. Genise, and P. Mukherjee. Approximate trapdoors for lattices and smaller hash-and-sign signatures. In S. D. Galbraith and S. Moriai, editors, *ASIACRYPT 2019, Part III*, volume 11923 of *Lecture Notes in Computer Science*, pages 3–32. Springer, 2019.
8. C. Cremers, S. Düzlü, R. Fiedler, M. Fischlin, and C. Janson. Buffing signature schemes beyond unforgeability and the case of post-quantum signatures. In *IEEE Symposium on Security and Privacy*, pages 1696–1714, 2021.
9. T. Debris-Alazard, N. Sendrier, and J. Tillich. Wave: A new family of trapdoor one-way preimage sampleable functions based on codes. In *ASIACRYPT 2019*, volume 11921, pages 21–51. Springer, 2019.
10. P. Dusart. Explicit estimates of some functions over primes. *Ramanujan Journal*, 45:227–251, 2018.
11. T. Espitau, G. Niot, C. Sun, and M. Tibouchi. SQUIRRELS - Square Unstructured Integer Euclidean Lattice Signature, 2023.
12. M. Fischlin. Progressive verification: The case of message authentication. In *INDOCRYPT 2003*, pages 416–429. Springer Berlin Heidelberg, 2003.
13. P.-A. Fouque, J. Hoffstein, P. Kirchner, V. Lyubashevsky, T. Pornin, T. Prest, T. Ricosset, G. Seiler, W. Whyte, and Z. Zhang. Falcon: Fast-fourier lattice-based compact signatures over NTRU – submission to NIST post-quantum cryptography project, 2018. https://falcon-sign.info/.

14. C. Gentry, C. Peikert, and V. Vaikuntanathan. Trapdoors for hard lattices and new cryptographic constructions. In *Proceedings of the 40th Annual ACM Symposium on Theory of Computing, 2008*, pages 197–206. ACM, 2008.
15. A. Hülsing, L. Rausch, and J. Buchmann. Optimal parameters for XMSS-MT. In *Security Engineering and Intelligence Informatics*, pages 194–208, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.
16. A. Hülsing and J. Rijneveld. XMSS reference implementation, 2024. https://github.com/XMSS/xmss-reference.
17. D. V. Le, M. Kelkar, and A. Kate. Flexible signatures: Making authentication suitable for real-time environments. In *ESORICS 2019*, pages 173–193, Cham, 2019. Springer International Publishing.
18. H. W. Lenstra Jr. Factoring integers with elliptic curves. *Annals of mathematics*, pages 649–673, 1987.
19. D. Micciancio and C. Peikert. Trapdoors for lattices: Simpler, tighter, faster, smaller. In D. Pointcheval and T. Johansson, editors, *EUROCRYPT 2012*, volume 7237 of *Lecture Notes in Computer Science*, pages 700–718. Springer, 2012.
20. P. L. Montgomery. Speeding the Pollard and elliptic curve methods of factorization. *Mathematics of computation*, 48(177):243–264, 1987.
21. National Institute of Standards and Technology. Stateless hash-based digital signature standard. Technical Report Federal Information Processing Standards Publications (FIPS PUBS) 205, NIST, 8 2004.
22. P. Q. Nguyen and I. E. Shparlinski. Counting co-cyclic lattices. *SIAM Journal on Discrete Mathematics*, 30(3):1358–1370, 2016.
23. C. Pomerance, J. Selfridge, and S. S. Wagstaff, Jr. The pseudoprimes to $25 \cdot 10^9$. *Mathematics of Computation*, 35(151):1003–1026, 7 1980.
24. T. Pornin and J. P. Stern. Digital signatures do not guarantee exclusive ownership. In *ACNS 2005*, volume 3531 of *LNCS*, pages 138–150, 2005.
25. M. O. Rabin. Digitalized signatures and public-key functions as intractable as factorization. Technical Report 212, MIT Laboratory for Computer Science, 1979.
26. SPHINCS+ Team. SPHINCS+ reference implementation, 2024. https://github.com/sphincs/sphincsplus.
27. Squirrels team. Squirrels reference implementation, 2024. Package from https://squirrels-pqc.org/ with sha256sum `c4fd2a961f177a2f49970ca1758bde1a676a2fd7b3e62ea0e0c0593955902ab7`.
28. A. R. Taleb and D. Vergnaud. Speeding-up verification of digital signatures. *Journal of Computer and System Sciences*, 116:22–39, 2021.
29. S. S. Wagstaff. Is there a shortage of primes for cryptography? *International Journal of Network Security*, 3:296–299, 2006.
30. H. C. Williams. A modification of the RSA public-key encryption procedure. *IEEE Transactions on Information Theory*, 26:726–729, 1980.
31. P. Zimmermann and B. Dodson. 20 years of ECM. In *Algorithmic Number Theory, ANTS-VII*, pages 525–542, 2006.

A Subroutines for the explicit CRT

We maintain the notation of §4: $\mathbf{p} = (p_1, \ldots, p_s)$ is a list of distinct primes, $\Delta := \prod_{i=1}^{s} p_i$ is their product, $\Delta_i := \Delta/p_i$ and $q_i := \Delta_i^{-1} \pmod{p_i}$ for $1 \leq i \leq s$. Algorithm 7 computes $(q_1, \ldots, q_s)$. Algorithm 8 computes $([[\Delta_i]]_{\mathbf{r}})_{i=1}^{s}$ and $[[\Delta]]_{\mathbf{r}}$. given another list of primes $\mathbf{r} = (r_1, \ldots, r_t)$ (all prime to Δ). These algorithms are not optimal, but they avoid multiprecision arithmetic.

Algorithm 7: Explicit Modular CRT setup: q-coefficients.

Input: $\mathbf{p} = (p_1, \ldots, p_s)$
Output: $(q_1, \ldots, q_s)$ s.t. $0 < q_i < p_i$ and $q_i(\Delta/p_i) \equiv 1 \pmod{p_i}$ for $1 \le i \le s$

function `qCoefficients`$((p_1, \ldots, p_s))$
  $(q_1, \ldots, q_s) \leftarrow (1, \ldots, 1)$
  foreach $1 \le i \le s$ **do**
    foreach $1 \le j \le s,\ j \ne i$ **do**
      $q_i \leftarrow q_i \cdot p_j \bmod p_i$
    $q_i \leftarrow q_i^{-1} \bmod p_i$
  return $(q_1, \ldots, q_s)$

Algorithm 8: Explicit Modular CRT setup.

Input: $\mathbf{p} = (p_i)_{i=1}^{s}$ and $\mathbf{r} = (r_j)_{j=1}^{t}$.
Output: $([[\Delta_i]]_{\mathbf{r}})_{i=1}^{s}$ and $[[\Delta]]_{\mathbf{r}}$.

function `ModECRTSetup(`$\mathbf{p}$, $\mathbf{r}$`)`
  $\mathbf{m} \leftarrow [[1]]_{\mathbf{r}}$
  $(\mathbf{c}^{(1)}, \ldots, \mathbf{c}^{(s)}) \leftarrow ([[1]]_{\mathbf{r}}, \ldots, [[1]]_{\mathbf{r}})$
  foreach $1 \le i \le s$ **do**
    $\mathbf{u} \leftarrow [[p_i]]_{\mathbf{r}}$ // $u_j = p_j$ `or` $p_j - r_j$
    $\mathbf{m} \leftarrow (m_1 u_1 \bmod r_1, \ldots, m_t u_t \bmod r_t)$
    foreach $1 \le j < i$ ***and*** $i < j \le s$ **do**
      $\mathbf{c}^{(j)} \leftarrow (c_1^{(j)} u_1 \bmod r_1, \ldots, c_t^{(j)} u_t \bmod r_t)$
  return $(\mathbf{c}^{(1)}, \ldots, \mathbf{c}^{(s)})$, $\mathbf{m}$