Long Paper: SQISign on ARM

Luca De Feo[1], Li-Jie Jian[2], Ting-Yuan Wang[2,3], Bo-Yin Yang[2]

IBM Research Europe, Zürich, Switzerland[1]
Academia Sinica, Taipei, Taiwan[2]
University of Southern California[3]
iacr-papers@defeo.lu,{jcuyo613,deanwang88528}@gmail.com,by@crypto.tw

Abstract. We present the first vectorized implementation of SQIsign for high-performance Arm architectures. SQIsign is a promising candidate in the NIST On-Ramp Digital Signatures Call Round 2 thanks to its most compact key and signature sizes. However, its performance remains a bottleneck, in particular in key-generation and signing. SQIsign requires a large number of operations on elliptic curves and Abelian varieties, which depend on finite field arithmetic. Despite recent algorithmic improvements, research on high-performance vectorized implementations of these components is still unexplored.

Our main contribution is the first demonstration of non-trivial vectorization speedups for SQIsign. By leveraging the NEON instruction set, we implement highly efficient finite field arithmetic and batched elliptic curve operations tailored for 2-dimensional isogeny chain computations. This accelerates signing by 1.52× over the official NIST implementation at security level I. Moreover, our improvements are completely orthogonal to the recent algorithmic improvement Qlapoti (Asiacrypt 2025): combining the two, we achieve a speedup of more than 2.24×. We expect our work to inspire further SQIsign optimization through vectorization, especially for quaternion computations.

Keywords: Vectorization · SQIsign · Ideal-to-isogeny Conversion · Finite Field Arithmetic · Arm Architectures

1 Introduction

Post-quantum cryptography (PQC) refers to cryptographic algorithms that remain secure against quantum computers. With PQC we can continue to protect information against adversaries equipped with cryptographically relevant quantum computers.

In recent years, NIST has initiated and published PQC standards, including Key Encapsulation Mechanisms and digital signature schemes. In particular, there are three digital signatures, ML-DSA (based on CRYSTALS-Dilithium)[DKL+18], SLH-DSA (SPHINCS+) [BDE+17], and FN-DSA (FALCON)[FHK+19], that have been chosen for standardization. Partly in order to diversify the NIST digital signature portfolio against overreliance on lesser-studied lattice-based security assumptions, NIST opened a new call for additional digital signature schemes [ABC+24].

SQIsign [DKL+20] is a promising candidate in the second round of the NIST on-ramp call for digital signatures. As an isogeny-based digital signature scheme, SQIsign relies on the conjectured hardness of computing endomorphism rings of supersingular curves, which is fundamentally different from the security assumptions used in the current NIST PQC standards. Additionally, SQIsign is noted for the most compact key and signature sizes. For example, at NIST security level I, the combined size of SQIsign's public key and signature is approximately **7** times smaller than that of FN-DSA, **17** times smaller than ML-DSA, and **37** times smaller than SLH-DSA. Notwithstanding these advantages, the main challenge in SQIsign lies in its expensive and complex operation.

While recent advances [BDD+24, BCE+25] have yielded substantial algorithmic speedups, there is still ample room for software optimization. Specifically, the current open-source codebases have focused on portable implementations and on the x86 architecture [AAA+25b], with a few efforts limited to verification on Arm Cortex-M4 [CEMR24, AAA+25a, RFA+25]. In particular, the Arm NEON SIMD instruction set remains unexplored. Closing this implementation gap is vital for realizing competitive performance and fostering architectural diversity across modern CPU architectures.

SQIsign leverages Deuring's correspondence between supersingular curves over finite fields and maximal orders of a quaternion algebra. Most of the effort in key-generation and signing (over **80%** of the total running time) is spent in so-called ideal-to-isogeny conversions. These routines take as input or generate an ideal of a maximal order, i.e. a full rank lattice in $\mathbb{Q}^4$, and output an efficient representation of a corresponding isogeny over a finite field. All the variants of ideal-to-isogeny [DKL+20, DLRW24, BDD+24, BCE+25] involve some amounts of "quaternion computations" followed by some amounts of "isogeny computations". The former are linear-algebraic computations on unbounded integers, typically implemented on top of a multi-precision integer library (GMP in the official implementation). The latter dominate the overall cost, involving a large number of operations on elliptic curves and Abelian varieties, ultimately reducing to arithmetic over a finite field $\mathbb{F}_{p^2}$ of fixed multi-word characteristic p. Verification, on the other hand, only features isogeny computations. Optimizing the "isogeny component" is thus essential for improving overall performance.

1.1 Our Contributions

This work presents the first vectorized implementation of SQIsign on bigger CPUs using the Armv8-A. In particular, we focus on Cortex-A76 due to its enduring relevance across the computing spectrum. According to the Arm White Paper [PSB+20], Neoverse N1 is based on Cortex-A76. Since 2019, Neoverse N1-based processors have been widely deployed in cloud services. For instance, Amazon EC2 instances [Ser26] powered by AWS Gravition2 processors continue to offer multiple services. Furthermore, other major cloud providers, such as Microsoft Azure with its Dpdsv5-series virtual machines [Mic26] and Oracle OCI with Ampere A1 Compute instances [Ora26], deploy Ampere Altra processors. Therefore, Cortex-A76-class microarchitectures remain a practical target for optimization.

As announced, we focus on enhancing the performance of IdealToIsogeny and verification by optimizing isogeny computations. Indeed, we identified through profiling that these computations constitute the dominant bottleneck of SQIsign. Rather than attempting global optimization across all routines, we concentrate on vectorizing the isogeny computation and show that it alone yields significant end-to-end speedups.

Our primary contributions include:

1. To the best of our knowledge, this is the first work to demonstrate non-trivial vectorization speedups for SQIsign, on any platform. Our implementation achieves performance improvements on par with the latest algorithmic advances [BCE+25]. In addition, our result further indicates that implementation-level vectorization remains a largely unexplored yet impactful optimization dimension for SQIsign.

2. We implement a fast computation of 2-dimensional 2-isogeny chains. By leveraging NEON instruction set, we introduce vectorized arithmetic over $\mathbb{F}_p$ and $\mathbb{F}_{p^2}$, together with batched elliptic curve operations. This approach provides greater flexibility in choosing efficient arithmetic strategies, without strictly relying on optimized formulas from the Explicit-Formulas Database [BL].

3. At NIST security level I, we achieve **1.54**× faster in key generation, **1.52**× faster in signing, and **1.44**× faster in verification for SQIsign on Cortex-A76. In addition, our

work achieves at least a **1.22**× speedup for all SQIsign even on the highly superscalar Apple M1 processor.

The code of this work is publicly available at https://github.com/LeeJ-art/the-sqisign-vectorization

1.2 Related work

1.2.1 Verification-focused work

Prior work [CEMR24, AAA+25a, RFA+25] has focused on optimizing the verification of SQIsign. Specifically, [AAA+25a] proposed an optimized one-dimensional (1D) SQIsign verification based on the ApresSQI approach [CEMR24]. A key technique introduced by [AAA+25a] uses the property that evaluating two quadratic characters suffices to determine whether a point $P = (x, y) \in E(\mathbb{F}_{p^2})$ has an order divisible by 2^f. By efficiently identifying the specific point of order 2 over which P lies, a basis $\{P, Q\}$ can be computed and verified through a few quadratic character evaluations, avoiding the costly chain of point doublings. Therefore, this improvement nearly eliminates the non-isogeny overhead, reducing the total verification cost to essentially that of the core isogeny computations.

Additionally, the optimized 1D isogeny computation exploits its relatively long but simple structure, which can be split into smaller blocks for parallelization. The verification procedure of SQIsign fundamentally differs between its 1D and 2D variants. In the 1D case, verification requires evaluating a single but longer isogeny of degree 2^e with $e = 7.5\lambda$ (where λ is the security parameter), which is decomposed into $n = \lceil \frac{e}{f} \rceil$ blocks of 2^f-isogenies. Although this results in a longer isogeny chain, the uniform block structure presents a natural parallelization opportunity. In contrast, 2D verification computes an isogeny of degree approximately $(2^\lambda, 2^\lambda)$ within a single block. Since partitioning the higher-dimensional isogeny into independent submodules without introducing prohibitive intermediate overhead remains an open problem, parallelization of 2D verification is difficult to implement in practice.

The compact size of SQIsign's signature and public-key has motivated its deployment on resource-constrained devices such as the ARM Cortex-M4, especially for verification [RFA+25]. While Cortex-M4 has powerful multiple-accumulate instructions such as `UMAAL` that is executed in a single clock cycle, it does not support NEON instruction set, making our vectorization techniques inapplicable. However, the unsaturated field representation, finite field arithmetic, and lazy reduction techniques discussed in Section 3 remain effective for isogeny chain evaluations involving heavy consecutive arithmetic. As this work primarily targets ARM architectures with NEON support, we present verification improvements benefiting from NEON intrinsics in Table 5 while deferring Cortex-M4 optimizations to future work.

1.2.2 Arithmetic for variable-size big integers

Both key-generation and signing of SQIsign require variable-size arithmetic for quaternion computations. Although it may not lead to as large speed-ups as our work, optimizing the non-finite-field computations is equally important. At the moment, the unbounded quaternion integer coefficients force the official SQIsign implementation to use GMP, a dynamic multi-precision library. The recent work [KLKL26] tries to bound the integers appearing in the quaternion computation and replaces GMP with a fixed-precision integer implementation, without claiming any speed-up. This is a preliminary step towards optimizing the quaternion part of SQIsign. Once exact bounds are determined, we expect that SQIsign could further enhance efficiency with specialized vectorization in the quaternion part.

2 Preliminaries

2.1 The Deuring Correspondence

Let p be a prime and $\mathbb{F}_{p^2}$ a finite field with p^2 elements. Every *supersingular curve* in characteristic p is isomorphic to one defined over $\mathbb{F}_{p^2}$ and having $(p+1)^2$ points. There are $\lfloor p/12 \rfloor + c$ distinct isomorphism classes of such curves, with $c \in \{0, 1, 2\}$. Every two such curves are isogenous, and even connected by *short*[*] ℓ-isogeny walks for any prime $\ell \neq p$.

For any p, a supersingular elliptic curve can be constructed in time polynomial in $\log(p)$. For example, if and only if $p = 3 \pmod 4$, the curve of equation $y^2 = x^3 + x$ is supersingular. Starting from this one, any other supersingular curve can, in principle, be constructed efficiently by taking a short ℓ-isogeny walk for some small ℓ. However, given a pair of such curves, it is difficult in general to find any isogeny between them: even for a quantum computer, the best algorithm known has complexity polynomial in p.

A *quaternion algebra* over $\mathbb{Q}$ is a 4-dimensional $\mathbb{Q}$-algebra with basis $1, i, j, k$ and multiplication table given by $i^2 = a$, $j^2 = b$, $ij = -ji = k$, for some $a, b \in \mathbb{Q}$. A *(fractional) ideal* of a quaternion algebra B is a sub-lattice of rank 4. Two ideals I, J of B are said to be *left-equivalent* if there is an $\alpha \in B$ such that $I\alpha = J$. An ideal that is also a ring is called an *order*, it is said to be *maximal* if it is not contained in any other order. The *left order* of an ideal I is the largest order stabilizing it to the left: $O_L(I) = \{\alpha \in B \mid \alpha I \in I\}$, the *right order* being the one stabilizing I to the right. An ideal that is contained in its left (equivalently, right) order is said to be *integral*.

The *Deuring correspondence* is a remarkable equivalence relating supersingular curves and quaternion algebras. The endomorphisms of a supersingular curve form a ring isomorphic to a maximal order in a quaternion algebra. For example, the curve $y^2 = x^3 + x$ has endomorphism ring isomorphic to the order generated by $(1, i, \frac{j+1}{2}, \frac{k+i}{2})$, with $i^2 = -1$ and $j^2 = -p$. Fix a curve E_0 and a maximal order O_0 isomorphic to $\mathrm{End}(E_0)$. The Deuring correspondence states that the isogenies with domain E_0 are in one-to-one correspondence with integral left O_0-ideals (ideals whose left order is O_0), and the curves isogenous to E_0 are in one-to-one correspondence with left ideal classes of O_0. Additionally, if I_ϕ is the ideal corresponding to an isogeny ϕ, then the right order of I_ϕ is isomorphic to the endomorphism ring of the codomain of ϕ.

Ideals provide an efficient representation for isogenies: an ideal-to-isogeny algorithm is one that takes as input E_0, O_0, an integral ideal I_ϕ corresponding to an isogeny ϕ and a point $P \in E_0$, and outputs $\phi(P)$ in time polynomial in $\deg \phi$ and $\log(p)$. Such an algorithm is easily extended to evaluate isogenies of any supersingular curve, not just E_0.

But the quaternion analogue of the isogeny finding problem turns out to be easy too: given a left ideal class $[I]$, or equivalently the isomorphism class of its right order[†], an ideal representative of I can be computed in time polynomial in $\log(p)$. It follows that computing endomorphism rings of supersingular curves must be generically hard, otherwise we could combine this fact with an ideal-to-isogeny algorithm and solve the isogeny problem. And indeed computing endomorphism rings and finding isogenies of supersingular curves are proven equivalent, assuming the Generalized Riemann Hypothesis [Wes22].

2.2 Higher-dimensional Abelian Varieties

Abelian varieties are smooth projective varieties that are also algebraic groups. Elliptic curves are the only example of one-dimensional varieties. Two-dimensional Abelian varieties come in two kinds: Cartesian products of elliptic curves or Jacobians of genus 2 hyperelliptic curves. As the dimension grows, more and more kinds of Abelian varieties appear.

[*] By *short* walk, we mean one of length $O(\log(p))$.

[†] also known as its right order *type*.

Higher-dimensional Abelian varieties started being centrally featured in isogeny-based cryptography after the SIDH attacks [CD23, MMP+23, Rob23]. Their main use is *Kani's lemma*, a technique that embeds two isogenies of degrees a, b between Abelian varieties into an isogeny of degree $a + b$ between Abelian varieties of double the dimension. Despite the increase in dimension, an exponential complexity gain is achieved by setting up the isogenies so that $a + b$ is smooth, for example, $a + b = 2^n$. Kani's lemma was first used constructively in SQIsignHD [DLRW24], an improvement over the original SQIsign using 4-dimensional isogenies. Further improvements brought down to 2 the dimension of the Abelian varieties [NOC+24, BDD+24], ultimately informing NIST's round-two version of SQIsign [AAA+25b].

2.3 SQIsign

SQIsign is a Fiat-Shamir-style signature, based on a sigma-protocol proving knowledge of the endomorphism ring of a supersingular curve. In a nutshell, an origin supersingular curve E_0 is fixed, together with its endomorphism ring O_0. The secret key is a random ideal I_ϕ corresponding to an isogeny ϕ from E_0, the public key is the codomain of ϕ, computed via IdealToIsogeny.

When signing, the prover generates a random commitment curve, in the same way as they generated the public key. The verifier offers a challenge in the form of a 2-isogeny walk from the public key. The prover responds with an isogeny from the commitment curve to the public key, passing through the challenge isogeny. The intuition behind the scheme is that only a prover who knows the endomorphism rings of both the public key and the commitment could use the Deuring correspondence to compute an isogeny between these two curves.

The round-two version of SQIsign makes heavy use of Kani's lemma and 2-dimensional Abelian varieties in IdealToIsogeny, essentially following [BDD+24]. A recently published improvement over this algorithm, named Qlapoti [BCE+25], also uses 2-dimensional Abelian varieties.

The response too is transmitted to the verifier as a 2-dimensional isogeny, embedding the 1-dimensional isogeny from the public key to the commitment; Indeed the endomorphism rings of these curves are secret information, thus using ideals to represent the response isogeny is not an option. Because of this difference between prover and verifier, SQIsign's code is peculiar in that verification has much smaller code coverage than key-generation or signing. Figure 1 presents the dependency diagram of SQIsign, where we clearly see how key-generation and signing depend on both $\mathbb{F}_p$ arithmetic for the isogeny part and multi-precision integer arithmetic for the quaternions, whereas verification does not use quaternions at all. In this work, we focus on optimizing the 2-dimensional isogeny part, as it is the one that has the greatest impact by far on the overall running time.

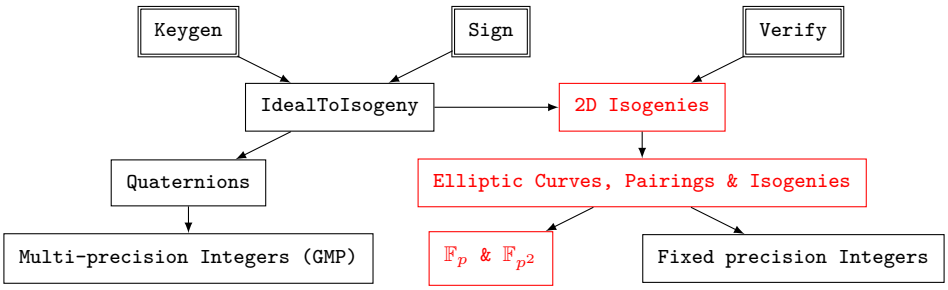


Figure 1: SQIsign's dependency graph, inspired by https://github.com/SQIsign/the-sqisign#project-structure. In red the modules to which our NEON optimizations are applied.

2.4 Ideal-to-isogeny conversion

Converting ideals to isogenies is the main computational cost of SQIsign's key generation and signing. There are a few variants of ideal-to-isogeny in SQIsign, but they can all be understood as converting some left $\mathcal{O}_0$-ideal I into its corresponding isogeny ϕ_I from E_0 via the Deuring correspondence. Below we describe the most general of these variants, IdealToIsogeny, as introduced in [BDD+24] and adopted in the NIST standard.

In practice, IdealToIsogeny evaluates the target isogeny ϕ_I by computing its codomain E_I and the images of the pre-determined basis $(P_0, Q_0) \in E_0[2^e]$. The general outline is as follows:

1. Find I_1, I_2 equivalent to I of coprime norms d_1, d_2 and u, v such that $ud_1 + vd_2 = 2^e$, with ud_1 coprime to vd_2.
2. Evaluate isogenies $\phi_u : E_0 \to E_u$ and $\phi_v : E_0 \to E_v$ of degrees u and v respectively on the 2^e-torsion group $E_0[2^e]$.
3. Use Kani's Lemma on $\phi_u \circ \widehat{\phi_{I_1}} : E_I \to E_u$ and $\phi_v \circ \widehat{\phi_{I_2}} : E_I \to E_v$, where $\phi_{I_1}, \phi_{I_2} : E_0 \to E_1$ are the isogenies corresponding to I_1 and I_2 respectively, to compute $\Phi : E_u \times E_v \to E_I \times E'$ that embeds the isogenies $\phi_{I_1} \circ \widehat{\phi_u}$ and $\phi_{I_2} \circ \widehat{\phi_v}$.
4. Use Φ to compute $(\phi_I(P_0), \phi_I(Q_0))$ by evaluating $\phi_I = [1/d_1]\phi_{I_1} \circ \beta_1$ on $(P_0, Q_0) \in E_0[2^e]$.

These procedures are summarized in Algorithm 1. We refer to [BDD+24, AAA+25b] for detailed proofs. IdealToIsogeny can be informally decomposed into two stages. In the first stage (Steps 1–2), one must sample two equivalent ideals with small coprime norms satisfying the relation, $u \cdot nrd(I_1) + v \cdot nrd(I_2) = 2^e$. Note that this stage incorporates a randomization process and may fail with non-negligible probability. The second stage then computes some kernel basis of a 2-dimensional abelian variety (Steps 4–7) and applies Kani's lemma to construct a (2,2)-isogeny chain (Steps 8–10), which involves several 2-dimensional isogeny evaluations and intensive finite-field arithmetic. This work focuses on accelerating the latter stage through vectorization.

Algorithm 1 IdealToIsogeny

Input: A left $\mathcal{O}_0$-ideal $I \subseteq \mathcal{O}_0$ and a basis $(P_0, Q_0) \in E_0[2^e]$, where $\mathcal{O}_0$ is a maximal order and isomorphic to $\mathrm{End}(E_0)$.

Output: The codomain E_I of ρ_I and the images of (P_0, Q_0) under ϕ_I.

1: Find I_1, I_2 equivalent to I of coprime norms d_1, d_2 and u, v such that $ud_1 + vd_2 = 2^e$, with ud_1 coprime to vd_2.
2: Compute $I_i \leftarrow I\overline{\beta_i}/nrd(I)$ for $i \in \{1, 2\}$.
3: Compute $\widehat{\phi_2} \circ \phi_1 \leftarrow \beta_2\overline{\beta_1}/nrd(I) \in End(E_0)$.
4: Compute $(\phi_u(P_0), \phi_u(Q_0))$ for the u-isogeny $\phi_u : E_0 \to E_u$.
5: Compute $(\phi_v(P_0), \phi_v(Q_0))$ for the v-isogeny $\phi_v : E_0 \to E_v$.
6: Set $K_P \leftarrow ([d_1u]\phi_u(P_0), \phi_v \circ \widehat{\phi_2} \circ \phi_1(P_0))$.
7: Set $K_Q \leftarrow ([d_1u]\phi_u(Q_0), \phi_v \circ \widehat{\phi_2} \circ \phi_1(Q_0))$.
8: Compute $\Phi : E_u \times E_v \to E_I \times E'$ of kernel $\langle K_P, K_Q \rangle$.
9: Evaluate $\Phi(\phi_u(P_0), 0)$ and $\Phi(\phi_u(Q_0), 0)$ to obtain $\phi_{I_1}(P_0)$ and $\phi_{I_1}(Q_0)$.
10: Use $(\phi_{I_1}(P_0), \phi_{I_1}(Q_0))$ to evaluate $\phi_I = [1/d_1]\phi_{I_1} \circ \beta_1$ on $(P_0, Q_0) \in E_0[2^e]$.

2.4.1 Qlapoti

Recently, a more efficient replacement for IdealToIsogeny, named Qlapoti, was published at Asiacrypt [BCE+25]. The key to the improvement is a technique for solving $nrd(I_1) +$

$nrd(I_2) = 2^e$, i.e., setting $u = v = 1$ in IdealToIsogeny. Thanks to this, Qlapoti dispenses completely with Steps 4–5, and saves many isogeny evaluations in Steps 6–7. See [BCE+25] for a detailed account.

Overall, Qlapoti speeds up both the quaternion computations and the isogeny computations, roughly by the same amount. Thus our vectorization techniques result in roughly the same overall speedup in key generation and signing, orthogonally to the use of Qlapoti, as we shall demonstrate in Section 5.

2.5 Barrett reduction

[Bar87] introduced an efficient technique to compute modular reduction $a \; mod \; q$. Assuming q is an odd number such that $q < R = 2^k$, where $a \in \mathbb{Z}$, the standard reduction can be expressed as $x \equiv a \mod q = a - (\text{round}\left(\frac{a}{q}\right) \cdot q)$. Barrett reduction provides an approximation, $a \mod q \approx a - \lfloor (a \cdot \lfloor R/q \rceil)/R \rceil \cdot q$. This formula replaces the division with a more efficient way using a single multiplication and a bit shift. The precomputed part, $\lfloor R/q \rceil$, represents an approximate quotient. This quotient is particularly valuable for effectively implementing the modular reduction step in large-integer multiplication. More details on its vectorized implementation will be presented in Section 3.

2.6 Montgomery representation

Given a modulus q and a Montgomery radix R, usually chosen as a power of 2 such that $gcd(q, R) = 1$ and $R > q$, an integer $a \in \mathbb{Z}_q$ is represented in Montgomery representation as:

$$A = aR \bmod q.$$

Montgomery multiplication [Mon85] enables efficient modular multiplication on elements with Montgomery representation. Specifically, under the representation, Montgomery multiplication is defined as:

$$\text{MontMul}(A, B) = ABR^{-1} \bmod q,$$

which computes modular multiplication without costly division by the modulus. The result preserves Montgomery representation. Consequently, sequences of modular operations can be performed entirely within Montgomery representation. In the end, conversion back to the standard representation requires only a single Montgomery multiplication with B = 1.

In our isogeny chain computation, field elements are reorganized to implement vectorized arithmetic. This change needs an adjustment the Montgomery radix. At NIST security level I, for example, we switch the Montgomery radix from $R_1 = 2^{255}$ to $R_2 = 2^{261}$ for subsequent modular operations, ensuring correctness while enabling efficient SIMD implementations.

2.7 ARMv8-A instruction

Armv8-A [ARM21] is the first 64-bit ARM architecture. Beyond standard 32-bit and 64-bit scalar operations, ARMv8-A offers a set of instructions for 32 128-bit vector registers, known as NEON. Specifically, NEON operates on multiple data elements within each register, supporting two 64-bit, four 32-bit, eight 16-bit, or sixteen 8-bit integers. However, a notable limitation is that NEON does not support 64-bit vectorized long multiplication. In contrast, the 8-bit, 16-bit, or 32-bit integer type has more powerful instructions.

3 Vectorized field arithmetic

We now present the vectorized finite field arithmetic operations essential for SQIsign. To reach NIST security levels, SQIsign [AAA+25b] selected three distinct parameter sets corresponding to security level I, III, and V. The prime p chosen for each level determines the size of the base field. Table 1 summarizes the concrete values associated with each of the three NIST security levels, along with the respective representations of a field element in both scalar and vector registers. The following vectorization techniques are suitable for all three security levels. For conciseness, we focus on NIST security level I as our primary reference, as these methods extend straightforwardly to levels III and V.

Following NIST security level I, SQIsign utilizes the prime $p = 5 \cdot 2^{248} - 1$ as the characteristic of the base field. We analyze four unsaturated representations for large integer multiplication in Montgomery representation in Subsection 3.2. Considering lazy reduction, Subsection 3.3 displays a tailored modular reduction for p. In Subsection 3.4, we introduce our vectorized arithmetic for $\mathbb{F}_p$ and show how these methods are extended to the extension field $\mathbb{F}_{p^2}$ in Subsection 3.5.

Table 1: Chosen parameters and corresponding limb configurations for SQIsign.

	NIST I	NIST III	NIST V
Prime p	$5 \cdot 2^{248} - 1$	$65 \cdot 2^{376} - 1$	$27 \cdot 2^{500} - 1$
Scalar (counts · bits)	$5 \cdot 51$	$7 \cdot 55$	$9 \cdot 57$
Vector (counts · bits)	$9 \cdot 29$	$14 \cdot 28$	$19 \cdot 27$

3.1 Lazy reduction

Reducing the number of modular reduction operations often yields significant savings. Specifically, in our finite field arithmetic, a modular reduction costs approximately two-thirds of a full multiplication. Therefore, we aim to delay reductions as much as possible at the register level.

A key technique is **lazy reduction**. It prevents the need for reduction and carry chain after every arithmetic operation by using unsaturated representation, which reserves space in the most significant bits of each integer. Given an m-bit integer that contains only numbers smaller than 2^n, where $m > n$, the integer can tolerate up to $(2^{m-n} - 1)$ consecutive additions/subtractions before a carry is required. We refer to this tolerance value as **bound**, which is equal to 2^{m-n}, where n is referred to as the limb size. That is, a modular reduction is only executed before the number of consecutive operations would exceed this bound.

In our implementation, the lazy reduction is static rather than dynamic. Specifically, we do not trigger reductions based on runtime values checks. Instead, we conduct bound analysis during the design phase to predetermine fixed reduction points. By incorporating function-specific safety margins, we ensure that overflow edge cases are never reached. Consequently, the reduction process does not introduce additional timing side-channel risks.

3.2 Configuration

There are four potential limb sizes: 27-bit, 28-bit, 29-bit, and 51-bit for NIST security level I, each with its own pros and cons. We analyze the overhead of $\mathbb{F}_p$ arithmetic and choose the most suitable unsaturated representation for our implementation.

- The 51-bit limb configuration is adopted from [AAA+25b]. The limbs have the bound of $2^{64-51} = 8192$. However, this configuration is inefficient for the ARM-v8-

A architecture because the NEON SIMD instruction set does not support 64-bit multiplication.

- The 29-bit limb configuration actually uses 8 limbs of 29 bits and a single 19-bit most significant limb. In particular, the lower 8 limbs have a tight bound: $2^{32-29} = 8$, and the most significant limb has a larger bound $2^{32-19} = 8192$. We employ an incomplete reduction on the lower limbs when their bound is reached. The resulting carry is accumulated into the most significant limb until its larger bound is also reached. Then, we apply a reduction on the most significant limb.
- The 28-bit limb configuration consists of 8 limbs of 28 bits and one 27-bit most significant limb. The lower 8 limbs have the bound of $2^{32-28} = 16$, and the most significant limb has a bound of $2^{32-27} = 32$. The same incomplete reduction technique can be used here, with the lower limbs having twice the bound. However, the smaller bound of the most significant limb causes more limitations.
- The 27-bit limb requires 9 limbs of 27 bits and one 8-bit most significant limb. The lower 9 limbs have the bound of $2^{32-27} = 32$, and the most significant limb has a bound of $2^{32-8} = 16777216$. While this approach increases the per-limb bound for the lower limbs, it requires an extra limb. This results in a substantial increase in computational cost, necessitating at least 19 additional `umlal` instructions per finite field multiplication.

Among the configurations discussed above, the 51-bit limb size is excluded because it does not effectively utilize SIMD instructions. The 27-bit limb size is also omitted, as the extra limb incurs additional computational overhead. As a result, we focus on the performance comparison between 28-bit and 29-bit limbs in $\mathbb{F}_p$ arithmetic, especially analyzing the required carry chains and the number of reductions.

In the 29-bit limb configuration, fpMulVec multiplies two 29-bit limbs, adds the corresponding congruence term obtained from a 29-bit by 19-bit multiplication, and performs a 29-bit right shift for the most significant limb of the output. Consequently, for any input smaller than $8\times$`(0x1FFFFFFF)` in the lower 8 limbs and $32\times$`(0x4FFFF)` in the most significant limb, the resulting limb still fits within $8\cdot29+19$ bits. In contrast, the 28-bit limb configuration involves a 28-bit by 28-bit multiplication, a congruence term computed from a product of 28-bit and 27-bit, and a 28-bit right shift. As a result, this configuration tolerates up to $16\times$`(0xFFFFFFF)` in the lower 8 limbs but only $1\times$`(0x4FFFFFF)` in the most significant limb.

The wider bound in the most significant limb allows the 29-bit limb configuration to better support lazy reduction during addition and subtraction with modular multiplication. Moreover, it enables the use of incomplete reduction, effectively exploiting the generous bound in the most significant limb. Conversely, the narrow tolerance of the 28-bit configuration in its most significant limb requires a complete reduction before each fpMulVec. Based on this analysis, we conclude that the 29-bit limb size offers a more efficient overall configuration.

As mentioned, the unsaturated representation can be extended to NIST security levels III and V. For level III, the 28-bit configuration is preferred. Although both 28-bit and 29-bit configurations require 14 limbs to represent field elements, the 28-bit configuration offers better flexibility for lower limbs and sufficient for IncompleteReduction in the most significant limb. In contrast, the 29-bit configuration appears unnecessarily redundant. For level V, we select the 27-bit configuration. While a 29-bit configuration requires only 18 limbs, it lacks the space to support additions/subtractions prior to multiplication. With the 19 limbs setting, the 27-bit configuration is not only more flexible than the 28-bit but also maintains compatibility with 64-bit scalar arithmetic.

Establishing a baseline is essential for analyzing corresponding carry and reduction operations in field arithmetic in Subsection 3.4 and Subsection 3.5. Throughout the remainder of this paper, unless otherwise specified, we use NIST security level I as an example, denoting values smaller than `(0x1FFFFFFF)` in the lower limbs and `(0x4FFFF)` in the most significant limb as the **base**. Noted that the analysis also applies equally to NIST security levels III and V, with `(0xFFFFFFF)` and `(0x7FFFFF)` as the bases for the lower limbs and `(0x40FFF)` and `(0x6BFFF)` as the bases for the most significant limb, respectively.

3.3 Modular reduction

For values approaching the bound, we employ IncompleteReduction to propagate the excess values from the reserved bits of each limb sequentially into the next, continuing until the most significant limb. Since this initial propagation step does not guarantee the most significant limb itself will not overflow, this process is termed an incomplete reduction.

If the most significant limb also requires reduction, we apply Barrett reduction. For subsequent vectorization, we implement this reduction for vector registers as BarrettReduced. Specifically, BarrettReduced computes both the remainder and the quotient of the most significant limb divided by 5. The remainder is retained in the most significant limb, whereas the quotient is added back to the least significant limb. Subsequently, a second incomplete reduction is performed to clear all reserved spaces within the limbs, preventing potential overflow in subsequent calculations.

In the 29-bit limb configuration, the most significant limb can tolerate at most $32\times$ base, ensuring the quotient after BarrettReduced is less than `(0x1FFFFFFF)`. Therefore, a complete reduction in the configuration requires only two IncompleteReduction and one BarrettReduced application.

Algorithm 2 IncompleteReduction

Input: $a < 8p$.
Output: Incomplete reduced a

$t_0[0] \leftarrow a_0[0]$
$t_0[1] \leftarrow a_0[1]$
$t_1[0] \leftarrow a_0[2]$
$t_1[1] \leftarrow a_0[3]$
$t_0 \leftarrow \texttt{shr}(t_0, 29)$
$t_1 \leftarrow \texttt{shr}(t_1, 29)$
$a_0 \leftarrow \texttt{and}(a_0, mask)$
for $i \leftarrow 1$ to 7 **do**
    $t_0 \leftarrow \texttt{addw}(t_0, \left[a_i[0], a_i[1]\right])$
    $t_1 \leftarrow \texttt{addw}(t_1, \left[a_i[2], a_i[3]\right])$
    $a_i[0], a_i[1] \leftarrow \texttt{and}(t_0, mask)$
    $a_i[2], a_i[3] \leftarrow \texttt{and}(t_1, mask)$
    $t_0 \leftarrow \texttt{shr}(t_0, 29)$
    $t_1 \leftarrow \texttt{shr}(t_1, 29)$
end for
$a_8[0], a_8[1] \leftarrow \texttt{add}(\left[a_8[0], a_8[1]\right], t_0)$
$a_8[2], a_8[3] \leftarrow \texttt{add}(\left[a_8[2], a_8[3]\right], t_1)$
return a

Algorithm 3 BarrettReduced

Input: a_8 is the most significant limb of a.
Output: Complete reduced a_8

1: $t_0 \leftarrow \texttt{shr}(a_8, 16)$
2: $t_1 \leftarrow \texttt{qrdmulh}(t_0, const)$ ▷ $const = \lfloor \frac{R}{q} \rceil, R = 2^{16}$, and $q = 5$
3: $t_2 \leftarrow \texttt{mls}(t_0, t_1, q)$
4: $quo \leftarrow t_1$
5: $t_0 \leftarrow \texttt{shl}(t_2, 16)$
6: $t_1 \leftarrow \texttt{and}(a_8, mask_{16})$ ▷ $mask_{16} = 2^{16} - 1$
7: $a_8 \leftarrow \texttt{add}(t_0, t_1)$
8: return a_8, quo

3.4 Vectorized $\mathbb{F}_p$ arithmetic

We implement vectorized 32-bit operations with the NEON instruction set. Since NEON vector registers are represented as 4-lane, 32-bit vectors, we process four field elements simultaneously and transpose them so that each limb of the field elements is stored across multiple vector registers. In Subsection 3.2, we determine the unsaturated representation of field elements for all NIST security levels. As described in Table 1, field elements are stored across 9, 14, and 19 vector registers, respectively. Figure 2 also offers a visualization of this process at NIST security level I.

Another trick is the replacement of multiplication with bit-shifting operations for modular multiplication in Montgomery representation. In particular, leveraging the property that $(p + 1) = 5 \cdot 2^{248}$, it needs to multiply the modulus to maintain the result within the target $(8 \cdot 29 + 19)$-bit format. Since the overall process has a true dependency, relying on the SIMD multiplication unit forces the pipeline to stall; it cannot be parallelized. However, using bit-shifting not only reduces this data dependency but also optimizes pipeline utilization for better performance. We call this vectorized, unsaturated, large integer multiplication `fpMulVec`. The complete algorithm is presented in fpMulVec.

Specifically, fpMulVec utilizes two vectors, t_0 and t_1, to store the unsigned multiplication result. The assembly implementation takes advantage of the properties of `umlal` to perform multiplication and accumulation with the first element, thereby saving an extra addition. Moreover, the `umull` is used initially to avoid having to initialize registers.

For carry handling, the low part is retained in the register $t64$ using a constant mask `(0x1FFFFFFF)`. The corresponding carry bits are then efficiently extracted via a shift right and temporarily stored in registers t_0, t_1 to be added to the next iteration. The mask is pre-defined and reused, eliminating the overhead of reloading. Overall, we have 36-$t64$ like variables to store the low parts. The return value, c, is obtained from the last 18 of $t64$ by extracting narrow elements and converting the data form from 2×64-bit to 2×32-bit.

Moreover, all instances of the for-loop and if statements present in fpMulVec are inlined in our asm implementation. This means that no actual loop control, conditional branching, or data reloading overhead occurs. The high efficiency is sustained by utilizing the NEON register structure (e.g., v0.2d or v0.4s) for direct data manipulation and ensuring data remains cached to minimize register reloading.

The multiplication for NIST security levels III and V follows the same approach as for level I. They adopt $(13 \cdot 28 + 19)$-bit and $(18 \cdot 27 + 19)$-bit formats for the respective field elements. Although these larger fields inherently require additional operations such as `mlal`, our use of unsaturated representations and widening multiplication ensures that no overflow occurs, even at the maximum column height of schoolbook multiplication. Thus our vectorization techniques remain consistent and effective across all NIST security levels.

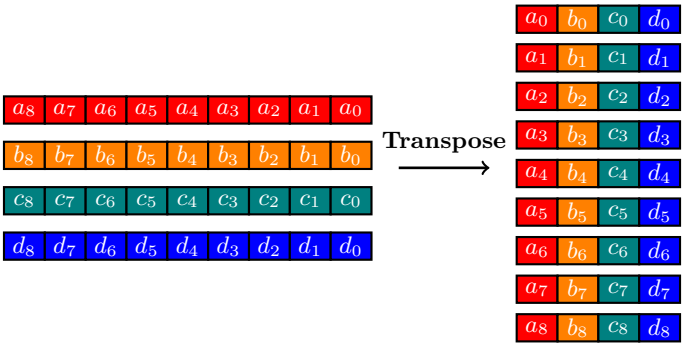


Figure 2: **Transpose**: Packing four $(8 \cdot 29 + 19)$-bit unsaturated limb elements into nine vector registers at NIST security level I.

Algorithm 4 fpMulVec

Input: a, b
Output: $c = abR^{-1} \bmod q$
for $i \leftarrow 0$ to 16 **do** ▷ Replace umlal(2) with umull(2) for the i=0 case.
  if $i \leq 8$ **then**
    for $j \leftarrow 0$ to i **do**
      $t_0 \leftarrow \texttt{umlal}(t_0, a_j, b_{i-j}),\ t_1 \leftarrow \texttt{umlal2}(t_1, a_j, b_{i-j})$
    end for
  else
    for $j \leftarrow 0$ to $(16 - i)$ **do**
      $t_0 \leftarrow \texttt{umlal}(t_0, a_{i+j-8}, b_{8-j}),\ t_1 \leftarrow \texttt{umlal2}(t_1, a_{i+j-8}, b_{8-j})$
    end for
  end if
  if $i \geq 8$ **then**
    for $k \leftarrow 0$ to 1 **do**
      $tmp_0 \leftarrow \texttt{shl}(t64_{i\%8+k}, 18),\ tmp_1 \leftarrow \texttt{shl}(t64_{i\%8+k}, 16)$
      $tmp_0 \leftarrow \texttt{add}(tmp_0, tmp_1)$
      $t_k \leftarrow \texttt{add}(t_k, tmp_0)$
    end for
  end if
  $t64_{(2i)} \leftarrow \texttt{and}(t_0, mask),\ t64_{(2i+1)} \leftarrow \texttt{and}\ (t_1, mask)$
  $t_0 \leftarrow \texttt{ushr}(t_0, 29),\ t_1 \leftarrow \texttt{ushr}(t_1, 29)$
end for
$t64_{34} \leftarrow \texttt{shrn}(\texttt{t}_0, 29)$
$t64_{35} \leftarrow \texttt{shrn}(\texttt{t}_1, 29)$ ▷ The latter 18 in $t64$ are stored in the output, c.

3.5 Vectorized $\mathbb{F}_{p^2}$ arithmetic

Each $\mathbb{F}_{p^2}$ element is processed using a cross-vector view, divided into a pair of 9×32-bit components. We use 18 vector registers to store an element of $\mathbb{F}_{p^2}$.

For the arithmetic in $\mathbb{F}_{p^2}$, fp2MulVec uses a Karatsuba-based method to reduce the required number of fpMulVec from 4 to 3. Similarly, in fp2SqrVec, the square is reduced to 2-fpMulVec, while adding only a few additional additions and subtractions.

Furthermore, to prevent overflow during subtraction in the $\mathbb{F}_{p^2}$ arithmetic, we must add an auxiliary modulus q before the operation. Specifically, q is defined as $q = kp$ for the smallest integer k such that kp is greater than the subtrahend. While the exact value of q depends on the corresponding subtrahend, it can still be precomputed and loaded without introducing any overhead.

Algorithm 5 fp2MulVec

Input: $a = a_0 + a_1 i, b = b_0 + b_1 i \in \mathbb{F}_{p^2}, q$
Output: $c = c_0 + c_1 i = (a_0 b_0 - a_1 b_1) + (a_0 b_1 + a_1 b_0) i$

for $i \leftarrow 0$ to 9 **do**
  $t_0[i] \leftarrow a_0[i] + a_1[i]$
  $t_1[i] \leftarrow b_0[i] + b_1[i]$
end for
$t_0 \leftarrow \mathrm{fpMulVec}(t_0, t_1)$
$t_1 \leftarrow \mathrm{fpMulVec}(a_1, b_1)$
$c_0 \leftarrow \mathrm{fpMulVec}(a_0, b_0)$
for $i \leftarrow 0$ to 9 **do**
  $c_1[i] \leftarrow t_0[i] + q[i]$
  $t_0[i] \leftarrow t_1[i] + c_0[i]$
  $c_1[i] \leftarrow c_1[i] - t_0[i]$
  $c_0[i] \leftarrow c_0[i] + q[i]$
  $c_0[i] \leftarrow c_0[i] - t_1[i]$
end for
return $c \in \mathbb{F}_{p^2}$

Since we are using the 29-bit limb configuration, we must consider the input bounds for fp2MulVec. The input $t \in \mathbb{F}_p$ for fpMulVec within fp2MulVec operations is derived by adding the inputs of fp2MulVec. The input bound for fp2MulVec becomes $4\times$ base for the lower limbs and $16\times$ base for the most significant limb. Moreover, fp2MulVec implements a Karatsuba-like method, which results in the real and imaginary parts of the output having different bounds. Specifically, the limbs in the real part are bounded by $2\times$ base, while the limbs in the imaginary part are bounded by $3\times$ base. On the other hand, fp2SqrVec shares the same input bound as fp2MulVec, but its output is only $1\times$ base.

Algorithm 6 fp2SqrVec

Input: $a = a_0 + a_1 i \in \mathbb{F}_{p^2}, q$
Output: $b = b_0 + b_1 i = (a_0^2 - a_1^2) + (2 a_0 a_1) i \in \mathbb{F}_{p^2}$

for $i \leftarrow 0$ to 9 **do**
  $t_0[i] \leftarrow a_0[i] + a_1[i]$
  $t_1[i] \leftarrow q[i] + a_0[i]$
  $t_2[i] \leftarrow a_1[i] + a_1[i]$
  $t_1[i] \leftarrow t_1[i] - a_1[i]$
end for
$b_1 \leftarrow \mathrm{fpMulVec}(a_0, t_2)$
$b_0 \leftarrow \mathrm{fpMulVec}(t_0, t_1)$
return $b \in \mathbb{F}_{p^2}$

Algorithm 7 fp2AddVec

Input: $a = a_0 + a_1 i, b = b_0 + b_1 i \in \mathbb{F}_{p^2}$
Output: $c = c_0 + c_1 i,$
$= (a_0 + b_0) + (a_1 + b_1) i$

for $i \leftarrow 0$ to 9 **do**
  $c_0[i] \leftarrow a_0[i] + b_0[i]$
  $c_1[i] \leftarrow a_1[i] + b_1[i]$
end for
return $c \in \mathbb{F}_{p^2}$

Algorithm 8 fp2SubVec

Input: $a = a_0 + a_1 i, b = b_0 + b_1 i \in \mathbb{F}_{p^2}$
Output: $c = c_0 + c_1 i,$
$= (a_0 - b_0) + (a_1 - b_1) i$

for $i \leftarrow 0$ to 9 **do**
  $t_0[i] \leftarrow a_0[i] + q[i]$
  $c_0[i] \leftarrow t_0[i] - b_0[i]$
  $t_0[i] \leftarrow a_1[i] + q[i]$
  $c_1[i] \leftarrow t_0[i] - b_1[i]$
end for
return $c \in \mathbb{F}_{p^2}$

4 Vectorizing SQIsign

In this section, we focus on evaluating isogenies from ideals for the NIST security level I of SQIsign signature scheme with vectorization perspectives. Following this, in Subsection 4.1, we utilize our vectorized large integer multiplication and the unsaturated representation in the supersingular elliptic curve point computations. Although our vectorized methods have higher algorithmic complexity, they are precisely designed for vector-friendly operations, achieving better performance. Next, Subsection 4.2 presents our vectorized approaches for theta coordinate computation, which is the most resource-demanding algorithm when computing a 2-dimensional 2-isogeny chain.

4.1 Efficiently computing elliptic points on Jacobian coordinates

The SQIsign signature scheme relies on point arithmetic computed in Jacobian coordinates. To improve performance, we focus on the use of NEON instruction sets by analyzing and optimizing vectorization strategies for three core algorithms: `DBL`, `DBLW`, and `GluingEval`. These algorithms and their calling functions are defined and elaborated on in [AAA+25b].

The `DBL` (Point Doubling) function for a Montgomery curve in Jacobian coordinates originally required 6 multiplications (M) and 6 squarings (S) for each call. The cost is suboptimal for direct single-instruction vectorization. Since the computation often requires simultaneous processing of two Jacobian points, these two DBL operations can be batched and implemented efficiently using just 3 vectorized field multiplications and 3 vectorized field squarings. The algorithm is presented as VectorizedBatchedDBL.

In particular, VectorizedBatchedDBL utilizes four vector registers, t_0-t_4, to record the intermediate results of $[2]P_1$ and $[2]P_2$. As detailed in 3.4, we operate with the 4×32-bit instruction set, where each vector register can directly access its four 32-bit elements. These elements are indexed from 0 to 3, specifically as $t[0]$, $t[1]$, $t[2]$, and $t[3]$. For clarity and ease of reading, we have omitted the data loading process and movement. In fact, we mitigate its impact on the efficiency of the algorithm by taking advantage of cached data and the `ext` instruction.

As detailed in Section 3, we implemented lazy reduction techniques in VectorizedBatchedDBL to eliminate redundant overhead. Our vectorized version requires only six complete reductions with reduced input, ensuring the output is restricted to $1\times$ base. Compared to the previous `DBL` approach, this optimization resulted in a computational saving of over twelve reductions.

`DBLW` is another point doubling function for a Weierstrass curve in Jacobian coordinates, typically requiring 3M and 5S. This operation count is also not good for direct vectorization. Furthermore, even when processing two points, the batched `DBLW` is not amenable to perfect instruction alignment as the batched `DBL`. Therefore, we employ a strategy that uses a slightly computationally redundant but vectorization-friendly algorithm: 4M and 4S per `DBLW` call. This strategy better utilizes NEON instruction sets, thereby resulting in better performance in practice. The adjusted batched `DBLW` is in VectorizedBatchedDBLW.

[CFA+12] states that the points on a short Weierstrass curve can be expressed using modified Jacobian coordinates. This representation uses the curve coefficient a to define a new, precomputed variable $T = aZ^4$. Hence, VectorizedBatchedDBLW takes the Jacobian points P_1 and P_2 as its input, represented by (X_P, Y_P, Z_P, T_P). Again, vector registers t_0 to t_5 are used to save intermediate results. In our specific implementation, we enhance efficiency by caching data and reusing these vector registers instead of wasting space and incurring data reloading overhead. Moreover, by using lazy reduction, we perform only eight complete reductions. This yields better performance than the reference `DBLW` that reduces after every operation. Precisely, VectorizedBatchedDBLW saves at least six reductions.

Algorithm 9 VectorizedBatchedDBL

Input: Jacobian points P_1, P_2 and their corresponding Montgomery curves E_1, E_2.
Output: The result of doubled Jacobian points $[2]P_1$ and $[2]P_2$

$t_0 \leftarrow \text{fp2SqrVec}(\left[X_{P_1}, X_{P_2}, Z_{P_1}, Z_{P_2}\right])$
$t_1 \leftarrow \text{fp2SqrVec}(\left[t_0[2], t_0[3], Y_{P_1}, Y_{P_2}\right])$
$t_1 \leftarrow \text{fp2AddVec}(\left[t_0[0], t_0[1], t_1[2], t_1[3]\right], \left[t_1[0], t_1[1], t_1[2], t_1[3]\right])$
$t_2 \leftarrow \text{fp2AddVec}(\left[t_0[0], t_0[1], X_{P_1}, X_{P_2}\right], \left[t_0[0], t_0[1], X_{P_1}, X_{P_2}\right])$
$t_0 \leftarrow \text{fp2MulVec}(\left[t_0[2], t_0[3], t_1[2], t_1[3]\right], \left[A_{E_1}, A_{E_2}, X_{P_1}, X_{P_2}\right])$
$t_3 \leftarrow \text{fp2AddVec}(\left[t_0[2], t_0[3], t_2[0], t_2[1]\right], \left[t_0[2], t_0[3], t_1[0], t_1[1]\right])$
$t_2 \leftarrow \text{fp2AddVec}(\left[t_0[0], t_0[1], t_3[0], t_3[1]\right], \left[t_2[2], t_2[3], t_3[0], t_3[1]\right])$
$t_4 \leftarrow \text{fp2MulVec}(\left[t_1[2], t_1[3], 2X_{P_1}, 2X_{P_2}\right], \left[t_2[0], t_2[1], t_0[0], t_0[1]\right])$
$t_4 \leftarrow \text{fp2AddVec}(\left[t_4[0], t_4[1], t_4[2], t_4[3]\right], \left[t_4[0], t_4[1], t_3[2], t_3[3]\right])$
$t_0 \leftarrow \text{fp2SqrVec}(\left[t_4[2], t_4[3], t_1[2], t_1[3]\right])$
$t_0 \leftarrow \text{fp2SubVec}(\left[t_0[0], t_0[1], 0, 0\right], \left[t_4[0], t_4[1], t_0[2], t_0[3]\right])$
$t_3 \leftarrow \text{fp2SubVec}(\left[t_3[0], t_3[1], t_0[2], t_0[3]\right], \left[t_0[0], t_0[1], t_0[2], t_0[3]\right])$
$t_1 \leftarrow \text{fp2MulVec}(\left[t_4[2], t_4[3], Y_{P_1}, Y_{P_2}\right], \left[t_3[0], t_3[1], Z_{P_1}, Z_{P_2}\right])$
$t_1 \leftarrow \text{fp2AddVec}(\left[t_1[0], t_1[1], t_1[2], t_1[3]\right], \left[t_3[2], t_3[3], t_1[2], t_1[3]\right])$
$X_{[2]P_1}, X_{[2]P_2}, Y_{[2]P_1}, Y_{[2]P_2}, Z_{[2]P_1}, Z_{[2]P_2} \leftarrow t_0[0], t_0[1], t_1[0], t_1[1], t_1[2], t_1[3]$
return $[2]P_1, [2]P_2$

Algorithm 10 VectorizedBatchedDBLW

Input: Modified Jacobian points P_1, P_2 for a Weierstrass curve
Output: The result of doubled modified Jacobian points $[2]P_1$ and $[2]P_2$

$t_0 \leftarrow \text{fp2SqrVec}(\left[X_{P_1}, Y_{P_1}, X_{P_2}, Y_{P_2}\right])$
$t_1 \leftarrow \text{fp2AddVec}(\left[t_0[0], t_0[1], t_0[2], t_0[3]\right], \left[t_0[0], t_0[1], t_0[2], t_0[3]\right])$
$t_2 \leftarrow \text{fp2AddVec}(\left[t_1[0], t_1[1], t_1[2], t_1[3]\right], \left[T_{P_1}, t_1[1], T_{P_2}, t_1[3]\right])$
$t_3 \leftarrow \text{fp2MulVec}(\left[t_2[1], Z_{P_1}, t_2[3], Z_{P_2}\right], \left[X_{P_1}, Y_{P_1}, X_{P_2}, Y_{P_2}\right])$
$t_4 \leftarrow \text{fp2AddVec}(\left[t_0[0], t_3[1], t_0[2], t_3[3]\right], \left[t_2[0], t_3[1], t_2[2], t_3[3]\right])$
$t_1 \leftarrow \text{fp2SqrVec}(\left[t_4[0], t_1[1], t_4[2], t_1[3]\right])$
$t_3 \leftarrow \text{fp2AddVec}(\left[t_3[0], t_1[1], t_3[2], t_1[3]\right], \left[t_3[0], t_1[1], t_3[2], t_1[3]\right])$
$t_2 \leftarrow \text{fp2SubVec}(\left[t_3[0], t_1[0], t_3[2], t_1[2]\right], \left[t_1[0], 4Y_{P_1}^2 X_{P_1}, t_1[2], 4Y_{P_2}^2 X_{P_2}\right])$
$t_5 \leftarrow \text{fp2AddVec}(\left[T_{P_1}, 4Y_{P_1}^2 X_{P_1}, T_{P_2}, 4Y_{P_2}^2 X_{P_2}\right], \left[T_{P_1}, t_2[0], T_{P_2}, t_2[1]\right])$
$t_5 \leftarrow \text{fp2MulVec}(\left[t_3[1], t_4[0], t_3[3], t_4[2]\right], \left[t_5[0], t_5[1], t_5[2], t_5[3]\right])$
$t_0 \leftarrow \text{fp2SubVec}(\left[t_2[1], t_5[1], t_2[3], t_5[3]\right], \left[4Y_{P_1}^2 X_{P_1}, t_3[1], 4Y_{P_1}^2 X_{P_1}, t_3[3]\right])$
$t_1 \leftarrow \left[t_5[0], 2Y_{P_1} Z_{P_1}, t_5[2], 2Y_{P_2} Z_{P_2}\right]$
$\left[X_{[2]P_1}, Y_{[2]P_1}, X_{[2]P_2}, Y_{[2]P_2}\right], \left[T_{[2]P_1}, Z_{[2]P_1}, T_{[2]P_2}, Z_{[2]P_2}\right] \leftarrow t_0, t_1$
return $[2]P_1, [2]P_2$

In addition to doubling operations, `GluingEval` is crucial in SQIsign. The techniques of gluing isogenies $\Phi : A_1 \to A_2$ and its subsequent (2,2)-isogeny chain evaluation significantly improve the efficiency of the scheme. We note A_1 is the Cartesian product of E_1 and E_2, and A_2 is the Jacobian of a hyperelliptic curve C, where C is a genus-2 hyperelliptic curve. A single gluing evaluation takes a gluing isogeny and a point P on A_1 as input, and computes the corresponding point $\Phi(P)$ on A_2 using 62M and 18S. As the basis evaluation requires the isogeny chain for two points simultaneously, the operation count doubles to 124M and 36S. While the total cost remains the same, we optimize for vectorization by adjusting the computation order in xz-coordinates to cache intermediate results, thereby achieving a better performance.

`GluingEval` initially relies on `AddComponents` for the cross-addition components. Specifically, the algorithm evaluates the gluing isogeny originating from the basis of $E_1 \times E_2$ and requires simultaneous computation for two distinct pairs of Jacobian points. The whole process costs four calls to `AddComponents` for their corresponding projective representations. However, this baseline approach is inefficient since the sequential instructions within `AddComponents` suffer from data dependency and cause redundant data reloading. To solve these issues, we combine and execute all four operations at once for vectorization optimization. A similar approach is applied in `ApplyIsomorphism`. Since `ApplyIsomorphism` is essentially a 4×4 matrix-by-4-vector multiplication, we adjust the instruction sequence to perform column-wise matrix-vector multiplication, further leveraging vectorization capabilities. Thanks to our reduction techniques, we only need to perform one additional reduction to ensure the output remains within $1\times$ base. This approach prevents a significant reduction overhead in `AddComponents` and `ApplyIsomorphism`. The complete and batched implementations are presented in VectorizedBatchedAddComponents and ApplyIsomorphismVec.

Algorithm 11 VectorizedBatchedAddComponents

Input: Four pairs of distinct Jacobian points, (P_1, Q_1), (P_3, Q_1) on E_1 and (P_2, Q_2), (P_4, Q_2) on E_2.
Output: (u, v, w) represents the projective x-only coordinates of $P + Q = (u - v : w)$ and $P - Q = (u + v : w)$.

$t_0 \leftarrow \text{fp2SqrVec}([Z_{P_1}, Z_{P_2}, Z_{P_3}, Z_{P_4}])$
$t_1 \leftarrow \text{fp2SqrVec}([Z_{Q_1}, Z_{Q_2}, Z_{Q_1}, Z_{Q_2}])$
$t_2 \leftarrow \text{fp2MulVec}([X_{P_1}, X_{P_2}, X_{P_3}, X_{P_4}], [t_1[0], t_1[1], t_1[2], t_1[3]])$
$t_3 \leftarrow \text{fp2MulVec}([X_{Q_1}, X_{Q_2}, X_{Q_1}, X_{Q_2}], [t_0[0], t_0[1], t_0[2], t_0[3]])$
$t_4 \leftarrow \text{fp2MulVec}([Y_{P_1}, Y_{P_2}, Y_{P_3}, Y_{P_4}], [Z_{Q_1}, Z_{Q_2}, Z_{Q_1}, Z_{Q_2}])$
$t_5 \leftarrow \text{fp2MulVec}([Z_{P_1}, Z_{P_2}, Z_{P_3}, Z_{P_4}], [Y_{Q_1}, Y_{Q_2}, Y_{Q_1}, Y_{Q_2}])$
$t_5 \leftarrow \text{fp2MulVec}([t_5[0], t_5[1], t_5[2], t_5[3]], [t_0[0], t_0[1], t_0[2], t_0[3]])$
$t_4 \leftarrow \text{fp2MulVec}([t_4[0], t_4[1], t_4[2], t_4[3]], [t_1[0], t_1[1], t_1[2], t_1[3]])$
$t_0 \leftarrow \text{fp2MulVec}([t_0[0], t_0[1], t_0[2], t_0[3]], [t_1[0], t_1[1], t_1[2], t_1[3]])$
$t_6 \leftarrow \text{fp2MulVec}([t_4[0], t_4[1], t_4[2], t_4[3]], [t_5[0], t_5[1], t_5[2], t_5[3]])$
$t_4 \leftarrow \text{fp2SqrVec}([t_4[0], t_4[1], t_4[2], t_4[3]])$
$t_5 \leftarrow \text{fp2SqrVec}([t_5[0], t_5[1], t_5[2], t_5[3]])$
$t_4 \leftarrow \text{fp2AddVec}([t_4[0], t_4[1], t_4[2], t_4[3]], [t_5[0], t_5[1], t_5[2], t_5[3]])$
$t_5 \leftarrow \text{fp2AddVec}([t_2[0], t_2[1], t_2[2], t_2[3]], [t_3[0], t_3[1], t_3[2], t_3[3]])$
$v \leftarrow \text{fp2AddVec}([t_6[0], t_6[1], t_6[2], t_6[3]], [t_6[0], t_6[1], t_6[2], t_6[3]])$
$t_3 \leftarrow \text{fp2AddVec}([t_3[0], t_3[1], t_3[2], t_3[3]], [t_3[0], t_3[1], t_3[2], t_3[3]])$
$t_1 \leftarrow \text{fp2MulVec}([A_{E_1}, A_{E_2}, A_{E_1}, A_{E_2}], [t_0[0], t_0[1], t_0[2], t_0[3]])$
$t_1 \leftarrow \text{fp2AddVec}([t_5[0], t_5[1], t_5[2], t_5[3]], [t_1[0], t_1[1], t_1[2], t_1[3]])$
$t_3 \leftarrow \text{fp2SubVec}([t_5[0], t_5[1], t_5[2], t_5[3]], [t_3[0], t_3[1], t_3[2], t_3[3]])$
$t_3 \leftarrow \text{fp2SqrVec}([t_3[0], t_3[1], t_3[2], t_3[3]])$
$w \leftarrow \text{fp2MulVec}([t_3[0], t_3[1], t_3[2], t_3[3]], [t_0[0], t_0[1], t_0[2], t_0[3]])$
$t_1 \leftarrow \text{fp2MulVec}([t_1[0], t_1[1], t_1[2], t_1[3]], [t_3[0], t_3[1], t_3[2], t_3[3]])$
$u \leftarrow \text{fp2SubVec}([t_4[0], t_4[1], t_4[2], t_4[3]], [t_1[0], t_1[1], t_1[2], t_1[3]])$
return (u, v, w)

Algorithm 12 ApplyIsomorphismVec

Input: A matrix $N_{4\times4} = [N_0, N_1, N_2, N_3]^T$, and a point P.
Output: The matrix-by-vector product in c.

$c \leftarrow \text{fp2MulVec}(\left[N_0[0], N_0[1], N_0[2], N_0[3]\right], \left[X_P, X_P, X_P, X_P\right])$
$t_0 \leftarrow \text{fp2MulVec}(\left[N_1[0], N_1[1], N_1[2], N_1[3]\right], \left[Y_P, Y_P, Y_P, Y_P\right])$
$c \leftarrow \text{fp2AddVec}(\left[c[0], c[1], c[2], c[3]\right], \left[t_0[1], t_0[1], t_0[1], t_0[1]\right])$
$t_0 \leftarrow \text{fp2MulVec}(\left[N_2[0], N_2[1], N_2[2], N_2[3]\right], \left[Z_P, Z_P, Z_P, Z_P\right])$
$c \leftarrow \text{fp2AddVec}(\left[c[0], c[1], c[2], c[3]\right], \left[t_0[1], t_0[1], t_0[1], t_0[1]\right])$
$t_0 \leftarrow \text{fp2MulVec}(\left[N_3[0], N_3[1], N_3[2], N_3[3]\right], \left[T_P, T_P, T_P, T_P\right])$
$c \leftarrow \text{fp2AddVec}(\left[c[0], c[1], c[2], c[3]\right], \left[t_0[1], t_0[1], t_0[1], t_0[1]\right])$
return c

Algorithm 13 VectorizedBatchedGluingEvalBasis

Input: Four Jacobian points P_1, P_2, P_3, P_4 are sampled from $P_A, P_B \in E_1 \times E_2$, where $P_1, P_2 \in P_A$ and $P_3, P_4 \in P_B$. Q_1, Q_2 such that are obtained by a 8-torsion point Q'' that satisfies $[4]Q'' \in ker(\Phi)$, the dual of the theta point $\Phi(Q'') = (x : x : y : y)$, and the change-of-basis matrix N.
Output: Theta coordinates of $\Phi(P_A)$, $\Phi(P_B)$

$(u, v, w) \leftarrow \text{VectorizedBatchedAddComponents}(P_1, P_2, P_3, P_4, Q_1, Q_2, E_1, E_2)$
$t_0 \leftarrow \text{fp2MulVec}(\left[u[0], u[2], u[0], w[0]\right], \left[u[1], u[3], w[1], u[1]\right])$
$t_1 \leftarrow \text{fp2MulVec}(\left[v[0], v[2], u[2], w[2]\right], \left[v[1], v[3], w[3], u[3]\right])$
$c_0 \leftarrow \text{fp2AddVec}(\left[t_0[0], t_0[1], t_0[2], t_0[3]\right], \left[t_1[0], t_1[1], t_1[2], t_1[3]\right])$
$t_2 \leftarrow \text{fp2AddVec}(\left[u[0], u[1], u[2], u[3]\right], \left[v[0], v[1], v[2], v[3]\right])$
$t_3 \leftarrow \text{fp2MulVec}(\left[t_2[0], t_2[2], w[0], w[2]\right], \left[t_2[1], t_2[3], w[1], w[3]\right])$
$t_4 \leftarrow \text{fp2SubVec}(\left[t_3[0], t_3[1], t_3[2], t_3[3]\right], \left[c_0[0], c_0[1], c_0[2], c_0[3]\right])$
$c_1 \leftarrow \text{fp2MulVec}(\left[v[0], w[0], v[2], w[2]\right], \left[w[1], v[1], w[3], v[3]\right])$
$u \leftarrow \text{ApplyIsomorphismVec}(N, \left[c_0[0], t_0[2], t_0[3], t_3[2]\right])$
$t_0 \leftarrow \text{fp2SqrVec}(\left[u[0], u[1], u[2], u[3]\right])$
$v \leftarrow \text{ApplyIsomorphismVec}(N, \left[t_4[0], c_1[0], c_1[1], 0\right])$
$t_1 \leftarrow \text{fp2SqrVec}(\left[v[0], v[1], v[2], v[3]\right])$
$u \leftarrow \text{ApplyIsomorphismVec}(N, \left[c_0[1], t_1[2], t_1[3], t_3[3]\right])$
$t_2 \leftarrow \text{fp2SqrVec}(\left[u[0], u[1], u[2], u[3]\right])$
$v \leftarrow \text{ApplyIsomorphismVec}(N, \left[t_4[1], c_1[2], c_1[3], 0\right])$
$t_3 \leftarrow \text{fp2SqrVec}(\left[v[0], v[1], v[2], v[3]\right])$
$t_0 \leftarrow \text{fp2SubVec}(\left[t_0[0], t_0[1], t_0[2], t_0[3]\right], \left[t_1[0], t_1[1], t_1[2], t_1[3]\right])$
$t_0 \leftarrow \text{HadamardVec}(\left[t_0[0], t_0[1], t_0[2], t_0[3]\right])$
$t_2 \leftarrow \text{fp2SubVec}(\left[t_2[0], t_2[1], t_2[2], t_2[3]\right], \left[t_3[0], t_3[1], t_3[2], t_3[3]\right])$
$t_2 \leftarrow \text{HadamardVec}(\left[t_2[0], t_2[1], t_2[2], t_2[3]\right])$
$\Phi(P_A) \leftarrow \text{fp2MulVec}(\left[t_0[0], t_0[1], t_0[2], t_0[3]\right], \left[y, y, x, x\right])$
$\Phi(P_A) \leftarrow \text{HadamardVec}(\left[\Phi(P_A)[0], \Phi(P_A)[1], \Phi(P_A)[2], \Phi(P_A)[3]\right])$
$\Phi(P_B) \leftarrow \text{fp2MulVec}(\left[t_2[0], t_2[1], t_2[2], t_2[3]\right], \left[y, y, x, x\right])$
$\Phi(P_B) \leftarrow \text{HadamardVec}(\left[\Phi(P_B)[0], \Phi(P_B)[1], \Phi(P_B)[2], \Phi(P_B)[3]\right])$
return $\Phi(P_A), \Phi(P_B)$

VectorizedBatchedGluingEvalBasis uses a batching way to implement two `GluingEval` operations simultaneously. To enhance efficiency and manage data dependencies, we also slightly modify the instruction sequence, allowing us to perform four extension field operations at once. This approach clearly benefits vectorization. Furthermore, for the reduced input, VectorizedBatchedGluingEvalBasis requires 48 complete reductions in total

to produce an output with $1\times$ base. Due to the smaller base of fp2MulVec output's real part, the specific sequence of fp2SubVec and HadamardVec within this algorithm allows us to perform a complete reduction only on the imaginary part. This strategic design improves the benefits gained from using lazy reductions.

The last few instructions involve `Hadamard`, which transforms the theta point. In fact, HadamardVec is functionally identical to the standard `Hadamard` in [AAA+25b], but it processes four $18\times$ 29-bit inputs, based on our configuration for this work. In particular, given a reduced input, HadamardVec yields an output that is four times the input value. Thus, we perform two complete reductions here to facilitate subsequent computations.

Algorithm 14 HadamardVec

Input: Four elements $x, y, z, w \in \mathbb{F}_{p^2}$.
Output: $a = (x+y+z+w)$, $b = (x-y+z-w)$, $c = (x+y-z-w)$, and $d = (x-y-z+w)$.
1: **for** $i \leftarrow 0$ to 17 **do**
2: $\quad t_0, t_1 \leftarrow (x[i] + y[i]), (x[i] + q[i] - y[i])$
3: $\quad t_2, t_3 \leftarrow (z[i] + w[i]), (z[i] + q[i] - w[i])$
4: $\quad a[i], b[i], c[i], d[i] \leftarrow (t_0 + t_2), (t_1 + t_3), (t_0 + 2q[i] - t_2), (t_1 + 2q[i] - t_3)$
5: **end for**
6: return (a, b, c, d)

In addition, all core algorithms, including point doubling on both Montgomery and Weierstrass curves, and the gluing evaluation, fundamentally rely on additions and subtractions. Within the doubling algorithms, these additions/subtractions consistently follow a multiplication/squaring step, or they occur as a continuous sequence before a multiplication/squaring operation. This inherent sequence of operations highlights the importance of implementing lazy reduction. Obviously, executing a modular reduction after every single finite field operation represents a high hidden cost in elliptic curve point computation. While [AAA+25b] similarly employed an unsaturated configuration, this optimization insight was not discussed. As a result, we emphasize the significance of the lazy reduction strategy. Our implementation results demonstrate that the time savings achieved closely mirror the theoretical performance benefits of vectorization.

4.2 Vectorization for (2,2)-Isogeny

In this part, we introduce the vectorized approach for computing the $(2,2)-$isogeny chain. For a full technical explanation of the $(2,2)-$isogeny $\Phi : E_1 \to E_2$, refer to [AAA+25b].

The core computation of $(2,2)-$isogeny relies on `GenericCodomain`, `GenericCodomainWith4Torsion`, `GenericCodomainWith8Torsion`, and `GenericEval`. Due to their reliance on two or three square root computations, `GenericCodomain` and `GenericCodomainWith4Torsion` are inherently difficult to vectorize. We use a similar approach as the previous work but forcefully adopt NEON instruction set. Thus, we do not detail the specialized vectorization implementations for these two algorithms.

In contrast, `GenericCodomainWith8Torsion` and `GenericEval` are more friendly to vectorization. The former takes as input two known 8-torsion points, $P_1, P_2 \in E_1[8]$, which satisfy $e_8(P_1, P_2) = 1$, and where the kernel of Φ is $\langle [4]P_1 \rangle \oplus \langle [4]P_2 \rangle$. The latter also uses theta coordinates to represent a point and the inverse of the dual theta null point. Notably, all representations in theta coordinates comprise four elements, $x, y, z, w \in \mathbb{F}_{p^2}$, making it naturally suitable for vectorization using NEON intrinsics. With vectorized representations, we efficiently implement VectorizedGenericCodomainWith8Torsion and VectorizedGenericEval using the algorithms described in Subsection 3.5 with the aid of a few permutations.

Algorithm 15 VectorizedGenericEval

Input: A point P with the representation in theta coordinates and the inverse of the dual theta null point $I = (\alpha^{-1}, \beta^{-1}, \gamma^{-1}, \delta^{-1})$.
Output: Theta coordinates of $P' = \Phi(P)$.

$t_0 \leftarrow \text{HadamardVec}(\left[x_P, y_P, z_P, w_P\right])$
$t_0 \leftarrow \text{fp2SqrVec}(\left[t_0[0], t_0[1], t_0[2], t_0[3]\right])$
$t_0 \leftarrow \text{fp2MulVec}(\left[t_0[0], t_0[1], t_0[2], t_0[3]\right], \left[\alpha^{-1}, \beta^{-1}, \gamma^{-1}, \delta^{-1}\right])$
$P' \leftarrow \text{HadamardVec}(\left[t_0[0], t_0[1], t_0[2], t_0[3]\right])$
return P'

Algorithm 16 VectorizedGenericCodomainWith8Torsion

Input: Theta coordinates of P_1, P_2 such that kernel of $\Phi = \langle [4]P_1 \rangle \oplus \langle [4]P_2 \rangle$.
Output: The dual isogeneous theta null point $(\alpha, \beta, \gamma, \delta)$, the inverse of the dual theta null point $I = (\alpha^{-1}, \beta^{-1}, \gamma^{-1}, \delta^{-1})$, and the theta null point 0_{E_2} on the image curve E_2, where E_2 satisfies $\Phi : E_1 \to E_2$.

$T_1, T_2 \leftarrow \text{HadamardVec}(\left[x_{P_1}, y_{P_1}, z_{P_1}, w_{P_1}\right]), \text{HadamardVec}(\left[x_{P_2}, y_{P_2}, z_{P_2}, w_{P_2}\right])$
$T_1, T_2 \leftarrow \text{fp2SqrVec}(\left[T_1[0], T_1[1], T_1[2], T_1[3]\right]), \text{fp2SqrVec}(\left[T_2[0], T_2[1], T_2[2], T_2[3]\right])$
if $0 \in \{T_1[0], T_1[1], T_2[0], T_2[1], T_2[2], T_2[3]\}$ **then**
    raise Exception
end if
$t0 \leftarrow \text{fp2MulVec}(\left[T_1[1], T_1[0], T_2[3], T_2[2]\right], \left[T_2[0], T_2[1], T_2[2], T_2[3]\right])$
$[\alpha, \beta, \gamma, \delta] \leftarrow \text{fp2MulVec}(\left[t_0[2], t_0[3], t_0[1], t_0[0]\right], \left[T_2[0], T_2[1], T_2[2], T_2[3]\right])$
$I \leftarrow \text{fp2MulVec}(\left[t_0[2], t_0[3], 0, 0\right], \left[T_1[1], T_1[0], T_2[3], T_2[2]\right])$
$I[2], I[3] \leftarrow \delta, \gamma$
if Processing in Verification **then**
    $t_1 \leftarrow \text{fp2MulVec}(\left[T_1[0], T_1[1], T_1[2], T_1[3]\right], \left[I[0], I[1], I[2], I[3]\right])$
    if $t_1[0] \neq t_1[1]$ or $t_1[2] \neq t_1[3]$ **then**
        raise Exception
    end if
    $t_1 \leftarrow \text{fp2MulVec}(\left[T_2[0], T_2[1], T_2[2], T_2[3]\right], \left[I[0], I[1], I[2], I[3]\right])$
    if $t_1[0] \neq t_1[1]$ or $t_1[2] \neq t_1[3]$ **then**
        raise Exception
    end if
end if
$0_{E_2} \leftarrow \text{HadamardVec}(\left[codo[0], codo[1], codo[2], codo[3]\right])$
return $[\alpha, \beta, \gamma, \delta], I, 0_{E_2}$

5 Benchmark

This section presents the benchmarking results of our vectorized implementation on high-performance Arm architectures, with a specific focus on the Cortex-A76. Subsection 5.1 evaluates the performance of our vectorized field arithmetic and elliptic curves operations used in IdealToIsogeny. Subsection 5.2 summarizes the overall performance of our vectorized SQIsign implementation. All reported results represent the median of $1,000$ benchmark runs, where `M` denotes 10^6 cycles.

We particularly focus on the profiling of the SQIsign signing algorithm, since it represents the scheme's predominant computation bottleneck. As shown in Table 2, IdealToIsogeny accounts for more than **80%** of the total signature generation cost, even when potential signing-failure cases are excluded. Therefore, the performance gains achieved in IdealToIsogeny through vectorization are directly reflected in the improvement of the overall signing procedure. Because IdealToIsogeny is also the main component in key generation, we observe similar gains there. Finally, because verification exclusively

consists of isogeny evaluations and finite field arithmetic, which are the core target of our vectorization, we observe significant gains there too.

Table 2: Performance profiles of the SQIsign signing algorithm at NIST security level I.

Algorithm	Subroutine	Reference	Qlapoti	This work	Orthogonality
Sign	Commit	157.08 M (1×)	89.86 M (1.75×)	97.62 M (1.61×)	55.53 M (2.83×)
	Challenge	0.18 M (1.00×)	0.18 M (1.00×)	0.18 M (1.00×)	0.08 M (1.00×)
	Response	210.09 M (1×)	142.84 M (1.47×)	138.97 M (1.51×)	98.10 M (2.14×)
	ChangeOfBasis	21.41 M (1×)	21.41 M (1.00×)	19.34 M (1.11×)	19.28 M (1.11×)
Commit	SampleIdealGivenNorm	3.12 M (1×)	3.12 M (1.00×)	3.12 M (1.00×)	3.12 M (1.00×)
	ReducedNorm	2.63 M (1×)	2.63 M (1.00×)	2.63 M (1.00×)	2.63 M (1.00×)
	IdealToIsogeny	153.94 M (1×)	83.93 M (1.83×)	92.39 M (1.67×)	49.58 M (3.10×)
Challenge	Challenge	0.18 M (1×)	0.18 M (1.00×)	0.18 M (1.00×)	0.18 M (1.00×)
Response	ComputeChallengeIdeal	1.74 M (1×)	1.74 M (1.00×)	1.74 M (1.00×)	1.74 M (1.00×)
	ComputeResponseQuaternion	5.41 M (1×)	5.41 M (1.00×)	5.41 M (1.00×)	5.41 M (1.00×)
	BacktrackingAndNormalize	0.03 M (1×)	0.03 M (1.00×)	0.03 M (1.00×)	0.03 M (1.00×)
	ComputeAuxiliaryNorm	0.71 M (1×)	0.71 M (1.00×)	0.71 M (1.00×)	0.70 M (1.01×)
	SampleAuxiliaryIdeal	13.34 M (1×)	13.34 M (1.00×)	13.33 M (1.00×)	13.33 M (1.00×)
	AuxiliaryIdealToIsogeny	151.55 M (1×)	84.21 M (1.80×)	91.85 M (1.65×)	50.24 M (3.02×)
	SplitAuxiliaryIsogeny	23.73 M (1×)	23.73 M (1.00×)	13.33 M (1.78×)	13.33 M (1.78×)
	ComputeRemainingChain	1.85 M (1×)	1.85 M (1.00×)	1.85 M (1.00×)	1.85 M (1.00×)
	ComputeChallengeCodmain	7.95 M (1×)	7.93 M (1.00×)	7.89 M (1.01×)	7.77 M (1.02×)

5.1 Component profiling

This part details the performance improvements achieved for each component of our vectorized implementation. The components include $\mathbb{F}_p$ and $\mathbb{F}_{p^2}$ arithmetic, as well as the various subroutines introduced in Section 4.

Many components within SQIsign are inherently well-suited for vectorization. Theoretically, SIMD instructions can yield a speedup of up to 4× over scalar integer operations. However, since the NEON instruction set does not support 64-bit integer multiplication in SIMD, the achievable speedup is typically limited to around 2×. As demonstrated in the following tables, our experimental results not only align with this theoretical expectation but, in several cases, even exceed it.

In Table 3, **Total** reports the measured cycle counts, while the numbers in parentheses represent the theoretical totals computed as **Count** · **Cycles**. Additionally, we compare our vectorized elliptic curve operations and isogeny chain computations with the reference implementations in Table 4. The algorithms with a star mark (*) are not constant time. We thus report the distribution of the main isogeny chain cycles, including minimum (**Min.**), median (**Median**), and maximum (**Max.**) statistics.

Our vectorization makes IdealToIsogeny **1.67**× faster. In particular, IdealToIsogeny consists of distinct phases: constructing two auxiliary isogeny chains, evaluating the main

(2,2)-isogenies chain via `ComputeIsogenyChain`, and computing the change of basis. As the primary computational bottleneck, `ComputeIsogenyChain` achieves a **2.32**× speedup. Additional performance gains come from constructing the auxiliary chains (1.76× faster) and applying endomorphisms to the precomputed basis (1.67× faster). However, other subroutines such as SuitableIdeals, lattice reduction, and Cornacchia's algorithm are not suitable for vectorization. Although these subroutines limit the overall speedup, `ComputeIsogenyChain` dominates the runtime of IdealToIsogeny, allowing the algorithm to still achieve a meaningful acceleration.

Table 3: Performance of $\mathbb{F}_p$ and $\mathbb{F}_{p^2}$ arithmetic at NIST security level I.

Algorithm	Cycles	Count(s)	Total	Improvement
`fpMul`	260	4	888 (1040)	1×
fpMulVec	300	1	300	**2.96**×
`fpSqr`	172	4	602 (688)	1×
`fpSqrVec`	212	1	212	**2.84**×
`fp2Add`	62	4	144 (248)	1×
fp2AddVec	56	1	56	**2.57**×
`fp2Sub`	58	4	132 (232)	1×
fp2SubVec	63	1	63	**2.10**×
`fp2Mul`	731	4	2831 (2924)	1×
fp2MulVec	951	1	951	**2.98**×
`fp2Sqr`	495	4	1849 (1980)	1×
fp2SqrVec	647	1	647	**2.86**×
`CompleteReduction`	182	1	182	1×

Table 4: Performance of components in IdealToIsogeny at NIST security level I.

Algorithm	Min.	Median	Max.	Improv.
IdealToIsogeny*	142.52 M	153.94 M	193.03 M	**1.67**×
IdealToIsogenyVec*	82.53 M	92.39 M	137.75 M	
`ComputeIsogenyChain*`	37.27 M	39.29 M	39.58 M	**2.32**×
`ComputeIsogenyChainVec*`	16.19 M	16.94 M	17.03 M	
`FixedDegreeIsogenyEval*`	30.07 M	35.19 M	83.65 M	**1.76**×
`FixedDegreeIsogenyEvalVec*`	15.44 M	19.96 M	68.94 M	
`EndomorphismApp*`	8.84 M	8.97 M	9.11 M	**1.67**×
`EndomorphismAppVec*`	5.23 M	5.36 M	5.48 M	
`GenericCodomainWith8Torsion*`	13568	13857	14136	**2.42**×
VectorizedGenericCodomainWith8Torsion*	5448	5735	5966	
Algorithm	**Cycles**	**Count(s)**	**Total**	**Improv.**
`DBL`	8638	2	17189	**2.39**×
VectorizedBatchedDBL	7205	1	7205	
`DBLW`	5941	2	11830	**1.19**×
VectorizedBatchedDBLW	9958	1	9958	
`GluingEval`	110404	1	110404	**2.68**×
VectorizedBatchedGluingEvalBasis	41261	1	41261	
`GenericEval`	26063	1	26063	**2.92**×
VectorizedGenericEval	8917	1	8917	

5.2 Performance

We present the performance improvements of our vectorized implementation in Table 5. Specifically, our optimizations yielded a significant improvement in execution time compared to the reference implementation: our signing process is **1.52**× faster than the reference [AAA+25b], key generation and verification are **1.54**× and **1.44**× faster, respectively.

The result conclusively demonstrates that `IdealToIsogeny` is both a major computational bottleneck within SQIsign and an exceptionally good target for vectorization.

Table 6 illustrates the performance of SQIsign across all NIST security levels. We naturally extend our vectorized field arithmetic approach to all NIST security levels. As discussed in Subsection 3.2, we tailor the unsaturated representation for level III and level V field elements respectively to implement vectorization. Hence, our vectorization approach provides effective acceleration and similar results across all levels.

Furthermore, Table 6 shows that our vectorized implementation remains highly competitive in the signing algorithm compared to Qlapoti [BCE+25]. More importantly, our optimizations are orthogonal to those of Qlapoti. While Qlapoti optimized the finding of equivalent ideals and reduced quaternion computations and isogeny computations, the core evaluation of isogeny chain remains the dominant cost within IdealToIsogeny. In conclusion, by integrating both approaches, SQIsign achieves an acceleration factor above **2.46**× in key generation and **2.24**× in signing, relative to the reference implementation.

Table 5: Benchmarks for SQIsign Level I. Results are the median of 1,000 benchmark runs, where timing is in Megacycles.

Architecture	**SQIsign**	**Reference**[AAA+25b]	**This work**	**Improvement**
Cortex-A76	KeyGen	170.59 M	110.63 M	**1.54**×
	Sign	390.23 M	256.37 M	**1.52**×
	Verify	32.93 M	22.85 M	**1.44**×
Cortex-A72	KeyGen	193.10 M	148.00 M	1.31×
	Sign	442.17 M	340.81 M	1.30×
	Verify	34.64 M	26.21 M	1.32×
Cortex-A53	KeyGen	304.65 M	252.60 M	1.20×
	Sign	692.34 M	577.60 M	1.20×
	Verify	46.72 M	36.50 M	1.28×
Apple M1	KeyGen	56.96 M	45.91 M	**1.24**×
	Sign	130.85 M	106.51 M	**1.22**×
	Verify	9.45 M	7.36 M	**1.28**×
Apple M3	KeyGen	49.84 M	42.37 M	1.17×
	Sign	115.25 M	97.89 M	1.17×
	Verify	7.99 M	6.59 M	1.21×

Table 6: The comparison shows that our vectorization is orthogonal to prior algorithmic optimizations and provides multiplicative speedups. SQIsign on Cortex-A76. Results represent the median over 1,000 runs, timing in 10^6 cycles.

NIST Level	**SQIsign**	**Approach**	**KeyGen**		**Sign**	
			Cycles	Improvement	Cycles	Improvement
I	Reference [AAA+25b]	Baseline	170.59 M	1×	390.23 M	1×
	Qlapoti [BCE+25]	Algorithmic	103.27 M	1.65×	255.19 M	1.53×
	This work	Vectorization	110.63 M	1.54×	256.37 M	1.52×
	Qlapoti+This work	Orthogonality	69.45 M	2.46×	174.58 M	2.24×
III	Reference [AAA+25b]	Baseline	494.15 M	1×	1132.27 M	1×
	Qlapoti [BCE+25]	Algorithmic	331.84 M	1.49×	873.02 M	1.30×
	This work	Vectorization	328.23 M	1.51×	762.81 M	1.48×
	Qlapoti+This work	Orthogonality	233.46 M	2.12×	620.84 M	1.82×
V	Reference [AAA+25b]	Baseline	970.20 M	1×	2255.85 M	1×
	Qlapoti [BCE+25]	Algorithmic	620.32 M	1.56×	1583.94 M	1.42×
	This work	Vectorization	655.45 M	1.48×	1560.39 M	1.45×
	Qlapoti+This work	Orthogonality	421.99 M	2.30×	1156.70 M	1.95×

Acknowledgements

Luca De Feo is supported by SNSF Consolidator Grant CryptonIs 213766. Taiwanese authors are supported by Academia Sinica Grand Challenge Project (AS-GCP-114-M01)

NSTC Grant 114-2221-E-001-014-MY3 and 114-2634-F-001-001-MBK. The work of Ting-Yuan Wang is also supported in part by a Google PSS Privacy Faculty Award.

References

[AAA+25a] Marius A. Aardal, Gora Adj, Arwa Alblooshi, Diego F. Aranha, Isaac Andrés Canales Martinez, Jorge Chávez-Saab, Décio Luiz Gazzoni Filho, Krijn Reijnders, and Francisco Rodríguez-Henríquez. Optimized one-dimensional SQIsign verification on intel and cortex-M4. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2025(1):497–522, 2025.

[AAA+25b] Marius A. Aardal, Gora Adj, Diego F. Aranha, Andrea Basso, Isaac Andrés Canales Martínez, Jorge Chávez-Saab, Maria Corte-Real Santos, Pierrick Dartois, Luca De Feo, Max Duparc, Jonathan Komada Eriksen, Tako Boris Fouotsa, Décio Luiz Gazzoni Filho, Basil Hess, David Kohel, Antonin Leroux, Patrick Longa, Luciano Maino, Michael Meyer, Kohei Nakagawa, Hiroshi Onuki, Lorenz Panny, Sikhar Patranabis, Christophe Petit, Giacomo Pope, Krijn Reijnders, Damien Robert, Francisco Rodríguez-Henríquez, Sina Schaeffler, and Benjamin Wesolowski. SQIsign. Technical report, National Institute of Standards and Technology, 2025.

[ABC+24] Gorjan Alagic, Maxime Bros, Pierre Ciadoux, David Cooper, Quynh Dang, Thinh Dang, John M. Kelsey, Jacob Lichtinger, Carl A. Miller, Dustin Moody, Rene Peralta, Ray Perlner, Angela Robinson, Hamilton Silberg, Daniel Smith-Tone, Noah Waller, and Yi-Kai Liu. Status report on the first round of the additional digital signature schemes for the nist post-quantum cryptography standardization process, 2024-10-24 04:10:00 2024.

[ARM21] ARM. Arm architecture reference manual armv8, for a-profile architecture. https://developer.arm.com/documentation/ddi0487/latest, 2021. Accessed: 2026-01-15.

[Bar87] Paul Barrett. Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor. In Andrew M. Odlyzko, editor, *Advances in Cryptology – CRYPTO'86*, volume 263 of *Lecture Notes in Computer Science*, pages 311–323, Santa Barbara, CA, USA, August 1987. Springer Berlin Heidelberg, Germany.

[BCE+25] Giacomo Borin, Maria Corte-Real Santos, Jonathan Komada Eriksen, Riccardo Invernizzi, Marzio Mula, Sina Schaeffler, and Frederik Vercauteren. Qlapoti: Simple and efficient translation of quaternion ideals to isogenies. In Goichiro Hanaoka and Bo-Yin Yang, editors, *Advances in Cryptology – ASIACRYPT 2025, Part IV*, volume 16248 of *Lecture Notes in Computer Science*, pages 174–205, Melbourne, VIC, Australia, December 8–12, 2025. Springer, Singapore, Singapore.

[BDD+24] Andrea Basso, Pierrick Dartois, Luca De Feo, Antonin Leroux, Luciano Maino, Giacomo Pope, Damien Robert, and Benjamin Wesolowski. SQIsign2D-West - the fast, the small, and the safer. In Kai-Min Chung and Yu Sasaki, editors, *Advances in Cryptology – ASIACRYPT 2024, Part III*, volume 15486 of *Lecture Notes in Computer Science*, pages 339–370, Kolkata, India, December 9–13, 2024. Springer, Singapore, Singapore.

[BDE+17] {Daniel J.} Bernstein, Christoph Dobraunig, Maria Eichlseder, Scott Fluhrer, Stefan-Lukas Gazdag, Andreas Hülsing, Panos Kampanakis, Stefan Kölbl,

Tanja Lange, {Martin M.} Lauridsen, Florian Mendel, Ruben Niederhagen, Christian Rechberger, Joost Rijneveld, and Peter Schwabe. Sphincs+ - submission to the nist post-quantum cryptography project, 2017. Submission available at \https://sphincs.org.

[BL] Daniel J. Bernstein and Tanja Lange. Explicit-Formulas Database. https://www.hyperelliptic.org/EFD/. Accessed: 2026-01-15.

[CD23] Wouter Castryck and Thomas Decru. An efficient key recovery attack on SIDH. In Carmit Hazay and Martijn Stam, editors, *Advances in Cryptology – EUROCRYPT 2023, Part V*, volume 14008 of *Lecture Notes in Computer Science*, pages 423–447, Lyon, France, April 23–27, 2023. Springer, Cham, Switzerland.

[CEMR24] Maria Corte-Real Santos, Jonathan Komada Eriksen, Michael Meyer, and Krijn Reijnders. AprèsSQI: Extra fast verification for SQIsign using extension-field signing. In Marc Joye and Gregor Leander, editors, *Advances in Cryptology – EUROCRYPT 2024, Part I*, volume 14651 of *Lecture Notes in Computer Science*, pages 63–93, Zurich, Switzerland, May 26–30, 2024. Springer, Cham, Switzerland.

[CFA+12] Henri Cohen, Gerhard Frey, Roberto Avanzi, Christophe Doche, Tanja Lange, Kim Nguyen, and Frederik Vercauteren. *Handbook of Elliptic and Hyperelliptic Curve Cryptography, Second Edition*. Chapman & Hall/CRC, 2nd edition, 2012.

[DKL+18] Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, Peter Schwabe, Gregor Seiler, and Damien Stehlé. Crystals-dilithium: A lattice-based digital signature scheme. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2018(1):238–268, Feb. 2018.

[DKL+20] Luca De Feo, David Kohel, Antonin Leroux, Christophe Petit, and Benjamin Wesolowski. SQISign: Compact post-quantum signatures from quaternions and isogenies. In Shiho Moriai and Huaxiong Wang, editors, *Advances in Cryptology – ASIACRYPT 2020, Part I*, volume 12491 of *Lecture Notes in Computer Science*, pages 64–93, Daejeon, South Korea, December 7–11, 2020. Springer, Cham, Switzerland.

[DLRW24] Pierrick Dartois, Antonin Leroux, Damien Robert, and Benjamin Wesolowski. SQIsignHD: New dimensions in cryptography. In Marc Joye and Gregor Leander, editors, *Advances in Cryptology – EUROCRYPT 2024, Part I*, volume 14651 of *Lecture Notes in Computer Science*, pages 3–32, Zurich, Switzerland, May 26–30, 2024. Springer, Cham, Switzerland.

[FHK+19] 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. 2019.

[KLKL26] Won Kim, Jeonghwan Lee, Hyeonhak Kim, and Changmin Lee. Sqisign with fixed-precision integer arithmetic. In Shi Bai and Edoardo Persichetti, editors, *Public-Key Cryptography – PKC 2026*, pages 3–30, Cham, 2026. Springer Nature Switzerland.

[Mic26] Microsoft. Microsoft azure virtual machines documention. https://learn.microsoft.com/en-us/azure/virtual-machines/sizes/

general-purpose/dpdsv5-series?tabs=sizebasic, 2026. Accessed: 2026-01-15.

[MMP+23] Luciano Maino, Chloe Martindale, Lorenz Panny, Giacomo Pope, and Benjamin Wesolowski. A direct key recovery attack on SIDH. In Carmit Hazay and Martijn Stam, editors, *Advances in Cryptology – EUROCRYPT 2023, Part V*, volume 14008 of *Lecture Notes in Computer Science*, pages 448–471, Lyon, France, April 23–27, 2023. Springer, Cham, Switzerland.

[Mon85] Peter L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44:519–521, 1985.

[NOC+24] Kohei Nakagawa, Hiroshi Onuki, Wouter Castryck, Mingjie Chen, Riccardo Invernizzi, Gioella Lorenzon, and Frederik Vercauteren. SQIsign2D-East: A new signature scheme using 2-dimensional isogenies. In Kai-Min Chung and Yu Sasaki, editors, *Advances in Cryptology – ASIACRYPT 2024, Part III*, volume 15486 of *Lecture Notes in Computer Science*, pages 272–303, Kolkata, India, December 9–13, 2024. Springer, Singapore, Singapore.

[Ora26] Oracle. Oracle cloud infrastructure (oci) ampere compute platform. https://www.oracle.com/asean/cloud/compute/arm/, 2026. Accessed: 2026-01-15.

[PSB+20] Andrea Pellegrini, Nigel Stephens, Magnus Bruce, Yasuo Ishii, Joseph Pusdesris, Abhishek Raja, Chris Abernathy, Jinson Koppanalil, Tushar Ringe, Ashok Tummala, Jamshed Jalal, Mark Werkheiser, and Anitha Kona. The Arm Neoverse N1 Platform: Building Blocks for the Next-Gen Cloud-to-Edge Infrastructure SoC . *IEEE Micro*, 40(02):53–62, March 2020.

[RFA+25] Félix Carvalho Rodrigues, Décio Luiz Gazzoni Filho, Gora Adj, Isaac Andrés Canales Martinez, Jorge Chávez-Saab, Julio Cesar López-Hernández, Michael Scott, and Francisco Rodríguez-Henríquez. Generation of fast finite field arithmetic forCortex-M4 with ECDH and SQIsign applications. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2025(4):588–620, 2025.

[Rob23] Damien Robert. Breaking SIDH in polynomial time. In Carmit Hazay and Martijn Stam, editors, *Advances in Cryptology – EUROCRYPT 2023, Part V*, volume 14008 of *Lecture Notes in Computer Science*, pages 472–503, Lyon, France, April 23–27, 2023. Springer, Cham, Switzerland.

[Ser26] Amazon Web Services. Amazon ec2 instance types. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/instance-types.html, 2026. Accessed: 2026-01-15.

[Wes22] Benjamin Wesolowski. The supersingular isogeny path and endomorphism ring problems are equivalent. In *62nd Annual Symposium on Foundations of Computer Science*, pages 1100–1111, Denver, CO, USA, February 7–10, 2022. IEEE Computer Society Press.