Constant-time Integer Arithmetic for SQIsign

Fatna Kouider[1][0000−0002−8997−6100], Anisha Mukherjee[2][0009−0002−5565−5744], David Jacquemin[2][0009−0001−8386−006X], and Péter Kutas[1,3][0000−0002−2043−9542]

[1] Eötvös Loránd University, Hungary
[2] ISEC, Graz University of Technology, Austria
[3] University of Birmingham, United Kingdom

Abstract. SQIsign, the only isogeny-based signature scheme submitted to NIST's additional signature standardization call, achieves the smallest public key and signature sizes among all post-quantum signature schemes. However, its existing implementation, particularly in its quaternion arithmetic operations, relies on GMP's big integer functions, which, while efficient, are often not designed for constant-time execution.
In this work, we take a step toward side-channel-protected SQIsign by implementing constant-time techniques for SQIsign's big integer arithmetic, which forms the computational backbone of its quaternion module. For low-level fundamental functions including Euclidean division, exponentiation and the function that computes integer square root, we either extend or tailor existing solutions according to SQIsign's requirements such as handling signed integers or scaling them for integers up to ∼12,000 bits. Further, we propose a novel constant-time modular reduction technique designed to handle dynamically changing moduli.
Our implementation is written in C without reliance on high-level libraries such as GMP and we evaluate the constant-time properties of our implementation using Timecop with Valgrind that confirm the absence of timing-dependent execution paths. We provide experimental benchmarks across various SQIsign parameter sizes to demonstrate the performance of our constant-time implementation.

Keywords: SQIsign, big integer arithmetic, constant-time implementation

1 Introduction

Post-quantum cryptography has gained significant attention due to the imminent threats posed by quantum computers to widely deployed public-key cryptographic systems. Many classical cryptographic schemes, including RSA and those based on ECC, rely on the hardness of problems such as integer factorization and the discrete logarithm problem, which can be efficiently solved using Shor's algorithm [21,20] on a sufficiently large quantum computer. As a result, research in post-quantum cryptography (PQC) has explored alternative mathematical hard problems that remain secure against quantum attacks. Lattice-based, code-based, multivariate, hash-based, and isogeny-based cryptography have emerged

as strong candidates in this domain, each offering unique trade-offs between security, efficiency, and key/signature sizes. Among these, isogeny-based digital signatures have attracted considerable interest due to its compact key sizes and reliance on the hardness of problems related to supersingular isogeny graphs.

SQIsign [6,1] is the only isogeny-based signature scheme submitted to the National Institute of Standards and Technology (NIST) "additional signatures" call [18], a standardization initiative for post-quantum digital signatures based on mathematical hardness assumptions beyond lattice-based cryptography. It is a promising candidate that provides the smallest public key and signature sizes among all known post-quantum signature schemes, signatures as small as just 148 bytes and public keys as small as 65 bytes for NIST-I security, making it particularly attractive for applications with stringent bandwidth or storage constraints.

To ensure a thorough and comprehensive evaluation of the submitted schemes beyond their theoretical security, NIST encourages submissions to also consider side-channel resistance and implement constant-time techniques to avoid potential timing attacks. While SQIsign [6,1] offers strong theoretical security guarantees, it lacks a fully constant-time implementation of its signing procedure.

Motivation for this work: SQIsign relies on a complex framework that interpolates between supersingular elliptic curve arithmetic and its quaternion algebraic counterpart, facilitated by the Deuring correspondence. Efficient constant-time implementations for elliptic curve arithmetic have been widely explored in the literature, most recently in the context of SIKE [12], where extensive research has led to well-optimized and side-channel-resistant algorithms. However, the quaternion arithmetic involved in SQIsign is largely unexplored, as many of its computational techniques and algorithms are relatively novel and have not been extensively applied to other cryptographic constructions. This leaves significant room for further exploration, particularly in developing constant-time and efficient implementations of quaternion-based operations. Specifically, the most recent scheme specifications [1] submitted to Round-2 of the NIST call mentions that their current implementation, apart from elliptic curve and isogeny-based arithmetic, does not run in constant-time [1, Footnote-1, page-4]. Recently, [10,11] have proposed constant-time or time-independent alternatives for a selected few algorithms such as lattice reduction and GCD.

In this work, we take a step closer toward side-channel-protected implementations by introducing constant-time techniques, which form the foundation for secure cryptographic executions. The current implementation of SQIsign's quaternion arithmetic, encompassing all high-level algorithms within the quaternion module, relies on big integer arithmetic that is not inherently constant-time. To address this, we investigate the big integer functions for the source of non-constant time issues within them. We solve these issues one-by-one for all these functions within SQIsign's big integer arithmetic, as this constitutes the most fundamental computational layer upon which the higher-level quaternion arithmetic operations are built. Moreover, the big integer arithmetic in

SQIsign presents unique challenges due to its specialized nature, requiring careful handling of operations such as modular reduction, multiplication, and exponentiation to ensure efficiency and side-channel resilience without compromising correctness.

1.1 Our contributions

In this work, we present constant-time techniques for all the different integer arithmetic functions required in SQIsign, addressing several unique challenges posed by its parameter set, arithmetic structure, and lack of prior constant-time implementations. While constant-time multi-precision integer arithmetic libraries do exist in literature, SQIsign introduces specific challenges that necessitate new approaches.

Scalability to Large Integer Sizes: SQIsign requires handling integers of variable sizes, sometimes as large as $\sim 12,000$ bits due to the use of Hermite Normal Form (HNF) to perform operations with quaternion lattices. This scale raises questions about the efficiency and feasibility of existing constant-time big-integer implementations. We implement and analyze how available constant-time multi-precision arithmetic techniques scale up to such high parameters.

Handling Signed Integer Arithmetic: Many exiting cryptographic schemes such as SIKE [12], offer constant-time arithmetic techniques for several functions, but over modular arithmetic domains. SQIsign, on the other hand, operates on integer arithmetic, including signed integer representations, which introduces additional challenges. Our implementation extends constant-time techniques to handle signed integers efficiently while avoiding leakage from sign-dependent operations.

Constant-Time Implementations for Previously Under-explored Functions: Some integer functions (such as computing the nearest integer square root of a number, GCD or Euclidean division) are either very specific to SQIsign or rarely require constant-time solutions when used by other cryptographic primitives. To address such cases, we either develop constant-time versions of these functions tailored to SQIsign's specific requirements (e.g., for Euclidean division) or adapt the available constant-time solutions according to SQIsign (e.g., for GCD computation and integer square root function).

Modular Reduction with Dynamic Moduli: Unlike many cryptographic primitives that operate over a fixed modulus, for which efficient modular reduction techniques like Montgomery or Barrett reduction are well-optimized, SQIsign presents a unique challenge. During quaternion arithmetic, SQIsign involves ephemeral moduli that change dynamically throughout the signing procedure. These moduli are not only variable but often random, making conventional optimizations ineffective. We develop a custom modular reduction technique that accounts for these changing moduli while maintaining constant-time guarantees.

Implementation and benchmarks: The current SQIsign implementation heavily relies on GMP [9] functions, which, while efficient, do not guarantee constant-time execution. We provide a C implementation of the constant-time versions for the integer arithmetic functions required in SQIsign by completely eliminating such dependencies on high-level libraries. Furthermore, we conduct performance benchmarks across various SQIsign parameter sizes, including the largest configurations. To verify the constant-time nature of our implementation, we utilize Timecop with Valgrind, a tool which detects time-dependent branching and other execution variations. Additionally, we visually demonstrate the run-time behavior by plotting the average and median execution times (in CPU cycles) for the function implementations.

2 Background

In this section, we begin with a brief overview of SQIsign, followed by an analysis of the important integer functions, highlighting the non-constant time operations within them. Additionally, we discuss constant-time alternatives for some of these functions that are already available in the literature.

2.1 Notations

In this paper, we adopt the following notations for bitwise operations. Bitwise AND, OR and XOR operations are denoted by 'and', 'or' and 'xor', respectively. The operation of bitwise NOT is indicated using the overline notation, that is, $\bar{a}$ denotes the bitwise negation (complement) of a. Multiplication of two elements a and b is represented by $\cdot$, that is, $a \cdot b$. The expression $a \cdot (64)$ specifically represents the multiplication of a by $2^{64} - 1$. In this context, if $a = 1$, the result is a 64-bit value in which all bits are set to 1. Bitwise left and right shift operations of a by s positions are denoted by $a \ll s$ and $a \gg s$ respectively. The notation, '==' is used to compare the equality of two values.

We define the representation $\mathcal{D}(a)$ for a k-bit signed integer a as the following: $a = (a_{\text{sign}}, [a_{n-1}, a_{n-2}, \cdots, a_0])$, where a_{sign} is a bit value representing the sign of a and each a_i is a 64-bit word. The large integer a is a k-bit value, where the number of words satisfies $n \geq \lceil \frac{k}{64} \rceil$,and n is a fixed upper bound on the array size used for representing integers, we enforce $a_i = 0, \forall i \geq \lceil \frac{k}{64} \rceil$.

2.2 Supersingular elliptic curves, isogenies and quaternion algebras

An isogeny between two elliptic curves is a rational map which is simultaneously a group homomorphism. The degree of the isogeny is its degree as a rational map, i.e., the degree of the field extension it induces between the function fields of the two curves. Isogenies from an elliptic curve E to itself are called endomorphisms. Endomorphisms together with the zero map form a ring under addition

and composition called the endomorphism ring of the elliptic curve. Scalar multiplications are always endomorphisms thus $\mathbb{Z} \subseteq End(E)$. When elliptic curves are defined over finite fields then their endomorphism ring is always strictly larger than $\mathbb{Z}$. If $End(E)$ is commutative then E is called ordinary, otherwise it is called supersingular. In this paper we are always concerned with supersingular elliptic curves.

One can make a much more precise statement about the structure of the endomorphism ring of a supersingular elliptic curve. A quaternion algebra [22] is a central simple algebra of degree 2 over a field K. If $char(K) \neq 2$ then it is a four dimensional vector space over K with a basis $1, i, j, ij$ with the following properties: $i^2 = a, j^2 = b, ij = -ji$ where $a, b \in K^*$. One can thus write this quaternion algebra as $(a, b)_K$. A quaternion algebra is either isomorphic to $M_2(K)$ or is a division algebra (i.e, an algebra where every nonzero element has an inverse). In the context of supersingular elliptic curves the relevant quaternion is defined over $K = \mathbb{Q}$. Rational quaternion algebras can be classified by their local behavior,i.e., whether they are full matrix algebras or division algebras when you tensor them with a completion of $\mathbb{Q}$. This tensor product is almost always isomorphic to a full matrix algebra, except for finitely many cases which are places of ramification. Let p be the characteristic of the field of definition of the supersingular elliptic curve E. Then $B_{p,\infty}$ is the quaternion algebra ramified at p and infinity. When $p \equiv 3 \pmod 4$, then this simply means that $B_{p,\infty} \cong (-1, -p)_{\mathbb{Q}}$. A maximal order $\mathcal{O}$ in $B_{p,\infty}$ is a subring containing 1 which is a 4-dimensional lattice. The Deuring correspondence asserts that the endomorphism ring of a supersingular elliptic curve is a maximal order in $B_{p,\infty}$. An isogeny $E_1 \to E_2$ corresponds to a left ideal of $End(E_1)$ which is simultaneously a right ideal of $End(E_2)$. The norm of one-sided ideal is the greatest common divisor of the reduced norm of its elements and it corresponds to the degree of the corresponding isogeny.

2.3 The SQIsign protocol

SQIsign's authentication mechanism is based on the following sigma protocol with the starting reference curve E_0, typically chosen as $y^2 = x^3 + x$. The secret isogeny is $\tau : E_0 \to E$ such that the supersingular elliptic curve E is the public key. The endomorphism rings of these curves, $\mathcal{O}_0 = \mathrm{End}(E_0)$ and $\mathcal{O} = \mathrm{End}(E)$, are linked through a hidden left ideal I_τ of large prime norm, corresponding to the secret isogeny τ. In the quaternion setting, the private key is represented by $\mathcal{O}$, the endomorphism ring of E.

- **Commitment:** The prover constructs an isogeny $\phi : E_0 \to E_1$ and sends the resulting curve E_1 to the verifier.
- **Challenge:** The verifier selects and sends a challenge isogeny $\psi : E_1 \to E_2$ to the prover.
- **Response:** The prover computes and returns an isogeny $\sigma : E \to E_2$ of degree l^e.

The verifier checks the validity of the response by ensuring that the isogeny correctly connects the expected curves and, when composed with the dual of the challenge isogeny, forms a cyclic structure. If these conditions hold, the response signature σ is accepted by the verifier as valid.

The interplay between isogenies and quaternions is fundamental to the SQIsign protocol and is evident throughout its execution. In particular, while the actual computations of the key generation and response phases in version-1 of the NIST submission [6, Algorithm 26] heavily rely on quaternion algebra arithmetic, such as those performed by the KLPT algorithm, the challenge phase predominantly involves elliptic curve arithmetic. The KLPT algorithm [6, Section 2.5.2] itself includes several key sub-routines such as the FullRepresentInteger [6, Algorithm 10], FullStrongApproximation [6, Algorithm 13] and EichlerModConstraint [6, Algorithm 11], all of which fundamentally rely on integer arithmetic including Euclidean division and GCD computations. The last step of the response however, invokes the IdealToIsogeny [6, Algorithm 18] algorithm which transitions from quaternion-based computations to explicit elliptic curve arithmetic. The most recent version submitted to round-2 of the NIST call [1] no longer uses the KLPT algorithm during the signing procedure [1, Algorithm 4.2]. However, the commitment and the response phases still include quaternion algorithms (such as a version of RepresentInteger) and hence require big integer arithmetic.

2.4 Big integer arithmetic in SQIsign

The elements of $B_{p,\infty}$ are represented as 5-tuples of integers $(a, b, c, d, r) \in \mathbb{Z}^5$, corresponding to the quaternion, $\alpha = \frac{a+bi+cj+dk}{r}$, with a canonical representation obtained via reducing by the common denominator r. This representation inherently requires efficient big integer arithmetic, as all operations on quaternions translate to computations involving large integers. Due to the variable sizes of these integers, a dynamic multi-precision integer library such as GMP is used in the reference implementation of SQIsign.

Such a library provides efficient implementations of these operations but they are not inherently designed to be constant-time as their execution times often depend on input values, conditional branching, or iterative computations. For example, such functions include basic arithmetic operations such as left or right shifts, addition, subtraction, multiplication, and division, all of which are essential for handling large integer representations of quaternions. Computing integer square roots, both approximate and exact, is another key requirement. Traditional algorithms, such as Newton's method or the binary search approach, involve iterative refinements that depend on the magnitude of the input, leading to variations in execution time. An important computational algorithm is the greatest common divisor (GCD) algorithm, which finds an integer g such that $g = \mathrm{GCD}(a, b)$.This classical Euclidean algorithm involves division and remainder operations whose iteration count depends on the values of the input numbers, leading to data-dependent execution times. Modular arithmetic, including modular reduction and exponentiation, is another fundamental operation. Techniques such as Montgomery [16] or Barrett [2] reduction algorithms which are

commonly used to optimize modular reduction are most effective when the modulus remains fixed across multiple computations. However, in SQIsign, modular arithmetic within the quaternion algebra involves dynamically changing moduli rather than a single, fixed modulus and as a result, the pre-computation advantages that make Montgomery and Barrett reduction efficient in conventional settings do not directly apply to SQIsign. This makes it more challenging to design constant-time implementations, as optimizations that rely on fixed modular parameters cannot be utilized effectively. Another aspect of integer arithmetic in SQIsign is the representation and manipulation of lattices in the quaternion algebra $B_{p,\infty}$. Lattices in this algebra are defined by a basis $(\alpha_1, \alpha_2, \alpha_3, \alpha_4)$ of $\mathbb{Q}$-linearly independent quaternions, typically represented as columns of a matrix L, with the row vector description given by: $(\alpha_1\ \alpha_2\ \alpha_3\ \alpha_4) = (1\ i\ j\ k) \cdot L$ [6, Section 2.4.4.1]. For consistency and efficiency in representation, SQIsign enforces a canonical form for lattices by computing their Hermite Normal Form (HNF). The HNF of a lattice basis ensures uniqueness up to unimodular transformations. This normalization facilitates efficient comparison and manipulation of lattices within SQIsign. However, the conversion of a lattice basis to HNF leads to a significant increase in integer sizes. Since HNF enforces integer coefficients, any rational components arising from quaternion arithmetic must be cleared by multiplying through an appropriate denominator. This process can lead to very larger coefficients, resulting in a blow-up of integer sizes during operations. The increased bit-length of integers also increases the computational complexity of fundamental arithmetic operations. Many of the integer operations discussed earlier, such as GCD or Euclidean division are heavily involved in computing the HNF. Since these operations inherently depend on input values, the computation of HNF is not constant-time in standard implementations. The integer size explosion also affects performance, as the growth of coefficients increases the cost of subsequent reductions and multiplications.

2.5 Existing constant-time alternatives for integer functions

The work by [3] proposes a constant-time GCD algorithm in which the iterations within the GCD algorithm no longer depend on the inputs but on auxiliary parameters. The algorithm depends on a function called the Divstep that takes a parameter δ as an input, which is the actual constant-time element in the algorithm. In this GCD implementation, the auxiliary parameter δ depends on the maximum of the two input sizes, d, i.e., for two inputs f (odd) and g, $|f| < 2^d, |g| < 2^d$. The algorithm is constant-time if d is constant. It then calculates an integer m under a special function of d, $\mathsf{iteration}(d)$. This integer m serves as an upper bound of the number of iterations, [3, Theorem 11.2]. It ensures that whenever Divstep is executed m times with initial inputs (f, g), the sequence of intermediate values f_i and g_i are updated in a way such that, $f_n \to \pm\mathrm{GCD}(f, g)$ and, $g_n \to 0$. The algorithm uses iterative shifts and subtraction-based steps for all its sub-functions. We provide the pseudo-code of this algorithm in 9 in the appendix A.

In the context of SQIsign, we introduce an additional parameter r at the beginning of the algorithm 9 to handle the cases where f is even and adjust it to be odd before computing d and calling Divstep. This allows for handling all integers inputs without affecting the core computation process. We implement the algorithm entirely using our constant-time variants of the basic operations such as addition, subtraction, left/right shifts.

Korzilius and Schoenmakers in [14] present a constant-time integer square root algorithm based on the Newton-Raphson method and fixed-point arithmetic. They use the secure fixed-point arithmetic model introduced by Catrina et al. in [4,5] with parameterized precision expressed in terms of the number of fractional bits f. For $\ell > f \geq 0$, the set $\mathbb{Q}_{\ell,f}$ of ℓ-bit fixed-point numbers with f fractional bits and $e = \ell - f$ bits for the integer part is defined as, $\mathbb{Q}_{\ell,f} = \{\bar{x} \cdot 2^{-f} : \bar{x} \in \mathbb{Z}, -2^{\ell-1} \leq \bar{x} < 2^{\ell-1}\}$, with all $x \in \mathbb{Q}_{\ell,f}$ represented used two's complement binary representation:

$$x = (d_{\ell-1}, \ldots, d_0, d_1, \ldots, d_{-f})_2 = -d_{\ell-1} \cdot 2^{\ell-1} + \sum_{i=-f}^{\ell-2} d_i \cdot 2^i, \text{with } d_i \in \{0, 1\}.$$

Their approach begins by computing the Newton-Raphson based reciprocal square root $x \approx 1/\sqrt{a}$ for an input value a where $-2^{\ell-1} \leq 0 < a < 2^{\ell-1}$ as described in [14, Algorithm 5] with an error bound that does not exceed $\delta_f = 2^{-f}$. The required number of iterations θ is independent of the input value a; a will be scaled to $b = a \cdot v$ with $b \in [0.5, 2)$ and $v = 2^{2 \cdot m}$ before the Newton-Raphson iterations and scaled back by $v^{1/2}$ at the end. To increase the precision of this computation and improve the accuracy of the final result x, additional bits n can be added. The final result x of the reciprocal square root is subsequently used to compute an integer value q such that $q \leq \sqrt{a} < q + 1$ in the integer square root computation [14, Algorithm 7].

For the SQIsign implementation, we introduce a constant-time 64-bit precision based square root algorithm 10 (appendix A) to handle integers up to 2^{64}. From our experiments, we concluded that for larger integers of size $k > 2^{64}$, a precision of at least $\log_2(k)$ bits is required. However, further analysis is required to determine the exact precision requirement for handling large integers.

3 Constant-time Big-Integer Arithmetic

The standard set of bit-wise operations including AND, OR, XOR, NOT, Left shift ($\ll$) and Right shift ($\gg$) is available in the C programming language and is constant-time for manipulating individual bits within a standard data type like int or uint64_t. Similarly, the traditional arithmetic operations (addition, subtraction, negation and multiplication) are also considered $\mathcal{O}(1)$ because they are single CPU instructions. Handling more than 64 bits integers is not supported.

The current SQIsign implementation [15] relies on GMP's multi-precision integer library for big-integer arithmetic involving integers of varying sizes. For realizing a constant-time quaternion module it is natural to first eliminate the

GMP-dependency from all fundamental multi-precision integer arithmetic functions. We elaborate on constant-time solutions in this section.

For constant-time alternatives of basic low-level functions, we consult methods developed in SIKE [12] and propose tweaks to adapt them for SQIsign. Further, for the higher-level integer functions build on these fundamental operations, we either propose constant-time solutions ourselves or modify those which do not have a directly applicable constant-time solution in literature. For all the following descriptions and implementation, we use the representation $\mathcal{D}(a)$ for a signed k-bit integer a, that is, $a = (a_{\text{sign}}, a_{\text{abs}}) = (a_{\text{sign}}, [a_n, a_{n-1}, \cdots, a_0])$.

3.1 Low-level functions

In this section, we provide pseudo-codes 1 for constant-time algorithms of fundamental functions such as addition, subtraction, multiplication, negation, computing absolute value and exponentiation.

SIKE [12] provides constant-time arithmetic implementation for various functions over modular arithmetic domains with unsigned integers. Inspired by their unsigned version, we adapt Multiplication, conditional move ct-cmov,Negation and Absolute value for compatibility with signed integers by adding slight modifications based on bit-wise operation to handle the sign. For Addition and Subtraction, we use the two's complement representation to manage the signed integers. We implement a constant-time version of Exponentiation function based on the Montgomery powering ladder algorithm discussed in [17,13] using constant-time Addition, Multiplication and bit-wise operations.

Table 1: Constant-time versions of basic functions in the integer module. We denote b in $\mathcal{D}(b)$ representation and the resultant big integer by c which has a representation $\mathcal{D}(c)$. Addition and subtraction are computed with integers represented in two's complement, while multiplication is based on SIKE's implementation [12,7].

Functions	**Pseudocode**
Addition(a, b)	$c_i = a_i + b_i + \text{carry}_{i-1}$
Subtraction(a, b)	$c_i = a_i + (-b_i)$
Multiplication(a, b)	$c_i = a_i \cdot b_i, \quad c_{\text{sign}} = a_{\text{sign}} \text{ xor } b_{\text{sign}}$
Negation(a)	$c_{\text{sign}} = 1 \text{ xor } a_{\text{sign}}, \quad c_{\text{abs}} = a_{\text{abs}}$
Absolute value(a)	$c_{\text{sign}} = 0, \quad c_{\text{abs}} = a_{\text{abs}}$
Exponentiation(a)	Montgomery ladder[17,13]
ct-cmov(a)	$c_i = -1$ and $(a_i \text{ xor } c_i), \quad c_{\text{sign}} = a_{\text{sign}}$

Computing the bit-size of small integers: The $\mathsf{nbits}_{64}(x)$ algorithm 1 computes the bit-size of an unsigned integer $x < 2^{64}$ in constant-time. We initialize the bit-size n to 0. The algorithm then iterates 64 times performing the following operations: Step 3 increments n if $x > 0$, step 4 eliminate the least significant bit of x by shifting right by 1 bit preparing for the next iteration. This algorithm ensures a fixed number of iterations independent of the value of x. Moreover, each iteration involves simple bit-wise operations of addition and right shifts, thereby making it constant-time.

Algorithm 1 $\mathsf{nbits}_{64}(x)$

Require: $x \in \mathbb{Z}, |x| < 2^{64}$.
Ensure: n the bit-size of x.
1: Set $n = 0$.
2: **for** $i \in [0, 63]$ **do**
3: $\quad n = n + (x \text{ or } (0 - x)) \gg 63$. ▷ Using a constant-time right shift, see Alg. 4
4: $\quad x = x \gg 1$.
5: **end for**
6: **return** n

3.2 Computing the bit-size of big integers

We implement a constant-time Bitsize function 2 that computes the bit-size of a big integer. The function is specifically designed to eliminate iterative and conditional branches that depends on the input value or its size to avoid execution time variations.

In particular, the function computes the bit-size of a big integer a represented using $\mathcal{D}(a)$ by iterating from the most significant word a_{n-1} downward to the least significant word a_0. Once a nonzero word is found, the computation stops updating the result k. However,this does not imply breaking out of the loop. Instead, the algorithm continues to iterate, but during these iterations, it adds zeros to k, ensuring that the result remains unchanged. The final output k is the position of the most significant bit in a which also represents the bit length of a. Regardless of the value of a, the algorithm 2 executes a fixed number of iterations depending on the upper bound on the array size n and performs the same sequence of operations in each iteration.

Theorem 1. *The algorithm* $\mathsf{Bitsize}(a)$ *is correct and terminates in constant-time for a fixed upper bound of* n.

Proof. For a fixed upper bound on the array size of n, the algorithm is trivially constant-time as it is always executed in n iterations with constant-time operations.

Algorithm 2 $\mathsf{Bitsize}(a)$

Require: $a \in \mathbb{Z}$ is a k-bit integer represented using $\mathcal{D}(a)$ with n some pre-fixed upper bound on the array size.
Ensure: k the bit-size of a.
1: Set $k = 0$
2: **for** $i \in [n-1, 0]$ **do**
3: $s = \mathsf{nbits}_{64}(a_i)$
4: $r = 1$ xor $(k > 0)$.
5: $t = r \cdot (64)$ and s
6: $m = (t > 0)$ and $(r > 0)$
7: $k = k + t + 64 \cdot i \cdot m$
8: **end for**
9: **return** k

Now we prove correctness. Let a be a k-bit integer represented as $\mathcal{D}(a)$. Consider the index, $j = \lfloor \frac{k}{64} \rfloor$ which gives us the relation, $k = k \pmod{64} + j \cdot 64$. Within the *for* loop in Step 2, alg. 2, we scan through all the a_i's for comparing index i with j starting from $i = n-1$ to 0 using alg. 1. Let, $s_i = \mathsf{nbit64}(a_i)$.

There are three possibilities for s_i in alg. 1: if $i > j$, then $s_i = \mathsf{nbit64}(a_i) = 0$. If $i = j$, then $s_i = k \pmod{64}$ and we can then compute $k = \mathsf{nbit64}(a_i) + j \cdot 64$ to get the correct bit-size of a. Lastly, if $i < j$, $\mathsf{nbit64}(a_i)$ is not useful and needs to be discarded. This process of segregating the required output as per to the three conditions given above is carried out using three variables r, t and m that take values (for every iteration i) according to the definitions of the following three functions:

$$f : \mathbb{N} \to \{0,1\} \text{ such that, } k \mapsto \begin{cases} r = 1, & \text{if, } k = 0 \\ r = 0 & \text{if, k} > 0 \end{cases}$$

$$g : \mathbb{N}^2 \to \mathbb{N} \text{ such that, } (r,s) \mapsto \begin{cases} t = s, & \text{if, } r = 1 \\ t = 0, & \text{otherwise} \end{cases}$$

$$h : \mathbb{N}^2 \to \{0,1\} \text{ such that, } (k,t) \mapsto \begin{cases} m = 1, & \text{if, } k = 0 \ \text{ AND } \ t \neq 0 \\ m = 0, & \text{otherwise} \end{cases}$$

The constant-time expressions corresponding to the functions f, g and h are given in Steps 4-6 of the algorithm respectively.

Set $k = 0$; in Step 7, for all $i > j$, $s_i = 0$, $r_i = f(k_{i-1}) = 1$, $t_i = g(r_i, s_i) = 0$ and $m_i = h(k_{i-1}, t_i) = 0$, and thus, $k_i = k_{i-1} + t_i + 64 \cdot i \cdot m_i = k_{i-1} + g(r_i, s_i) + h(k_{i-1}, t_i) \cdot 64 \cdot i = k_{i-1} + 0 + 0 \cdot 64 \cdot i = 0$.

When $i = j$, $s_i \neq 0$ while $k_{i-1} = 0$, so, $r_i = 1$, $t_i = g(r_i, s_i) = s_i$ and $m_i = h(k_{i-1}, t_i) = 1$. Hence, $k_i = k_{i-1} + t_i + 64 \cdot i \cdot m_i = 0 + s_i + 64 \cdot j \cdot 1 = k$, which is the required correct output.

Finally, when $i < j$, $k_{i-1} \neq 0$, thus, $r_i = f(k_{i-1}) = 0$, $t_i = g(r_i, s_i) = 0$ and $m_i = h(k_{i-1}, t_i) = 0$. So, $k_i = k_{i-1} + t_i + 64 \cdot i \cdot m_i = k + 0 + 64 \cdot j \cdot 0 = k$. This means that once $i < j$, the value of k_i is not updated anymore in the remaining iterations. The output of this algorithm is therefore, $s_j + 64 \cdot j = k$.

3.3 Left and Right shift functions for big integers

As mentioned in Section 3, the standard set of bit-wise operations including Left/Right shift is available in the C programming language and are constant-time. However, shifts of more than 64 bits are not supported.

A left shift can be seen as a special case of multiplication. Shifting a number to the left by s bits is equivalent to multiplying it by 2^s. For example, $11_2 \ll 3 = 11000_2$ is equivalent to $3_{10} \cdot 2^3 = 24_{10}$. On the other hand, a right shift is a special form of division. Shifting a number to the right by s bits is the same as dividing it by 2^s. For example $11000_2 \gg 2 = 110_2$ is equivalent to $\frac{24_{10}}{2^2} = 6_{10}$, where the numbers in the subscript are the base of the representation.

While the basis of our proposed left/right shift algorithms are inspired from SIKE[12], we have modified them to remove some data dependencies which were present in the original constructions. These dependencies appeared because the input of the shift value s was also treated as a secret. We have redesigned the left shift operation (see Alg. 3) to be constant-time w.r.t s as well. Additionally, we propose a constant-time right shift algorithm (Alg. 4) similar to the constant-time left shift algorithm.

Algorithm 3 $\mathsf{Left_Shift}(a, s)$

Require: $a, s \in \mathbb{Z}$, with a an k-bit integer represented using $\mathcal{D}(a)$ and $\mathsf{Bitsize}(\mathcal{D}(a)) \leq n \cdot 64$, $s < n \cdot 64$ with n some upper bound on the array size.
Ensure: $a = (a \cdot 2^s)$
1: $r = s \bmod 64$
2: $j = s \gg 6$
3: Set $l = 1$
4: **for** $i \in [n-1, 0)$ **do**
5: $\quad a_i = l \cdot a_i$
6: $\quad j = l \cdot j$
7: $\quad l = l$ and $(i > j)$
8: $\quad a_i = (a_{i-j} \ll r)$ xor $((a_{i-j-1} \cdot l) \gg (64 - r))$
9: **end for**
10: $a_0 = (a_0 \ll r) \cdot l$
11: **return** b

Theorem 2. *The algorithm* $\mathsf{Left_Shift}(a, s)$ *is correct and terminates in constant-time for fixed upper bounds of* n *and* s*.*

Proof. Let a be a k-bit integer represented using $\mathcal{D}(a)$ representation. For a fixed upper bound on the array size n and a shift value $s < n \cdot 64$, the algorithm performs $n - 1$ iterations, each involving constant-time operations, thus the overall execution time of the algorithm 3 is constant-time regardless of the values of a and s.

Now we prove correctness. The goal of the algorithm is to compute the result of left shift of a by s, i.e, $a = a \cdot 2^s$. However, since the result is stored within

the n-word array representation using $\mathcal{D}(a)$, the result is taken module $2^{64 \cdot n}$. This can be expressed as: $a = (a \cdot 2^s) \pmod{2^{64 \cdot n}}$. Let, j, r be the quotient and remainder of s by 64, $s = r + 64 \cdot j$ with $r = s \pmod{64}$ and $j = s \gg 6$. The variable r handles shifts within a single 64-bit block, while j represents the number of 64-bit blocks that is modified by the shift s.

The *for* loop scans through all words of a, updating them according to their relative positions with respect to j. For each iteration i of the loop, we define l as an indicator variable, which determines whether the index i is within the shifted range or not. Then the transformation is applied to a_i using the shift logic in step 8.

In step 8, the left shift is performed by taking the relevant bits from a_{i-j} (shifted left by r positions) and the carry bits from a_{i-j-1}(shifted right by $64 - r$ positions). These two sets of bits will then be combined using a bit-wise xor operation to form the final result. This ensures that the shifted value and the carry bits are properly integrated. Furthermore, each bit that was originally at position x in a will be moved to position $x + s$ in the final result.

- If $(s = 64 \cdot j)$, then $(r = 0)$ and there is no within-block shift, thus, $a_i = a_{i-j}$ in the shifted range $i \in [n-1, j]$ and a_i set to 0 outside this range.
- If $(s = 64 \cdot j + r), (r \neq 0)$, Step 8 correctly handles the shift operation and properly manages the carry bit from a_{i-j-1} within the shifted range, while a_i is set to 0 outside this range. The correctness condition depends on the following three cases of the relationship between i and j:

1. $i > j$ (case that handles most significant words beyond the shifted range): When $i > j$ we set $l = 1$, thus, $a_i = a_i$. The step 8 correctly shifts the bits left by s within this range, $i \in \{n-1, \ldots, j+1, j\}$, the step also manages the carry-over from the lower word. The updated words are, $a_i = (a_{i-j} \ll r)$ xor $((a_{i-j-1}) \gg (64 - r))$.
2. $i = j$ (case that handles the particular word a_i where the shift transitions occur): When $i = j$, we still have $l = 1$ from previous iterations, therefore $a_i = a_i$ and $j = j$. Now l will be set to 0 because $i = j$ and then Step 8 will perform a left shift operation. The updated word is given as, $a_i = (a_0 \ll r)$ xor 0.
3. $i < j, i \neq 0$ (case that handles least significant words that should be zeroed): When $i < j$, all bits in a_i must be discarded, so we enforce $a_i = 0$ using the indicator variable l, $a_i = l \cdot a_i = 0$ and $j = 0$, all updates to a_i become zero in this range $i \in [j-1, 0)$. The updated words are, $a_i = (a_i \ll r)$ xor $((a_{i-1} \cdot 0) \gg (64 - r))$.

Then for the first element a_0, when $i = 0$, we perform a left shift by r only if $s < 64$ represented by the equation, $a_0 = (a_0 \ll r) \cdot l$.

Theorem 3. *The algorithm* Right_Shift(a, s) *is correct and terminates in constant-time for fixed upper bounds of n and s.*

Proof. The proof of constant-time exactly follows the proof 3.3 of Theorem 2. Again, the proof for correctness closely follows 3.3 with suitable changes in steps 4, 7, 8 and 10.

Algorithm 4 Right_Shift(a, s)

Require: $a, s \in \mathbb{Z}$, a is k-bit integer represented using $\mathcal{D}(a)$ and $\mathsf{Bitsize}(\mathcal{D}(a)) \leq n \cdot 64$, $s < n \cdot 64$ and n an upper bound on $\mathcal{D}(a)$'s array size.
Ensure: $a = \lfloor a \cdot 2^{-s} \rfloor$
1: $r = s \bmod 64$
2: $j = s \gg 6$
3: Set $l = 1$
4: **for** $i \in [0, n-1)$ **do**
5: $\quad j = j \cdot l$
6: $\quad a_i = a_i \cdot l$
7: $\quad l = l$ and $((i + j + 1) < n)$.
8: $\quad a_i = (a_{i+j} \gg r)$ xor $((a_{i+j+1} \cdot l) \ll (64 - r))$
9: **end for**
10: $a_{n-1} = (a_{n-1} \gg r) \cdot l$
11: **return** a

3.4 Constant-time Euclidean division

The work in [11] introduces a constant-time half-GCD algorithm. It employs the same technique as in [3], adapting it to the Euclidean algorithm. Specifically, the algorithm also uses two sequences, with one that converges to the required output and the other to zero, as explained in Section 2.5, ensuring both correctness and constant-time.

In our work, we build upon this concept with certain modifications to perform Euclidean division 5 instead of half-GCD. By adjusting the stopping condition, we terminate the algorithm earlier while preserving the constant-time property.

Algorithm 5 Euclidean_division(a, b)

Require: $a, b \in \mathbb{Z}$
Ensure: $q, r \in \mathbb{Z}$ such as $a = b \cdot q + r$ with $0 \leq r < b$.
1: $m = a$
2: $q = 0$
3: $n = 2 \cdot (\mathsf{Bitsize}(a) - \mathsf{Bitsize}(b)) + 1$
4: **for** $i \in [0, n-1]$ **do**
5: $\quad s = b - m$
6: $\quad \text{sign} = (s_{sign}$ and $(b == m))$
7: $\quad u = b \cdot sign.$
8: $\quad \alpha = \mathsf{Bitsize}(m) - \mathsf{Bitsize}(u) - 1$, or 0.
9: $\quad m = m - u \ll \alpha$ ▷ Using a constant-time left shift algorithm, see Alg. 3
10: $\quad q = q + sign \ll \alpha$
11: **end for**
12: Set $r = m$
13: **return** q, r

Theorem 4. *The algorithm* Euclidean_division(a, b) *is correct and terminates in constant-time for a fixed upper bound of* n*.*

Proof. Let a and b be atmost k-bit integers represented using $\mathcal{D}(a)$ and $\mathcal{D}(b)$. For a fixed upper bound on the array size n, the algorithm operates in exactly n iterations, each consisting of constant-time operations, making its execution constant-time.

Now we prove correctness. This division algorithm is derived from the constant-time half-GCD for Cornacchia's algorithm presented in [11]. Its correctness is closely related to that of the constant-time half-GCD algorithm. By modifying step 5, where the control variable is changed from $l = \sqrt{a}$ in [11] to b, we alter the point at which the algorithm stagnates. Consequently, the termination condition is adjusted to $m < b$. Since our algorithm follows the structure of the Euclidean algorithm, it produces $m = a \mod b$, where $m \in [0, b-1]$.

As our algorithm follows the Euclidean algorithm, keeping a track of the number of times a is subtracted from b gives the quotient q of the Euclidean division. This is done in Step 10, by adding the left-shifted value of the variable 'sign' by α.

3.5 Constant-time modular reduction for varying moduli

In most applications, the algorithms that are used for modular reduction are either the Barrett reduction or the Montgomery modular reduction. These techniques are particularly efficient when modular arithmetic is performed within a fixed prime field, as they allow certain constants derived from the modulus to be precomputed, significantly reducing the cost of division operations. For example, Barrett reduction approximates division by precomputing an estimate of the reciprocal of the modulus b. Specifically, for a fixed modulus b, one precomputes: $\mu = \left\lfloor \frac{2^{2k}}{b} \right\rfloor$, where k is chosen such that $2^k > b$. This method is efficient because it replaces costly division operations with multiplication and bit-shifting. However, it is only practical when b is fixed so that μ can be precomputed once and reused. Similarly, Montgomery reduction avoids explicit division by transforming numbers into a special representation known as the Montgomery domain, where modular multiplication is performed efficiently. The core idea is to precompute a constant Montgomery factor, $R = 2^k$, where k is chosen such that $R > b$. Additionally, the modular inverse of b with respect to R and the modular inverse of R with respect to b are also pre-computed and reused over multiple operations. This approach is particularly beneficial for modular exponentiation, as multiplication in the Montgomery domain avoids costly modular division. However that is not valid in SQIsign, specially for quaternion arithmetic which requires modular reduction of $a \mod b$, with both a and b as large integer variable inputs. One approach is to use a constant-time version of the Euclidean Algorithm (see Alg. 5). We also propose a constant-time modular reduction for varying moduli, as given in Alg. 6 based on table access, which would make it hardware friendlier.

Theorem 5. *The algorithm* Reduction(a, p) *is correct and terminates in constant-time for a fixed upper bound of* k *on the bit-size of the input* a*.*

Algorithm 6 Reduction(a, p)

Require: A k-bit integer a, an n-bit prime p, with $k > n$
Ensure: $b = a \pmod p$
1: Select a small integer r
2: Create the table T such that $T : s \in [0, 2^r - 1] \rightarrow s \cdot 2^n \mod p$
3: Set $b = a$
4: **for** $i \in [1, (k-n)/r + 1]$ **do**
5: Set b_H as the r-MSB of b ▷ Can be done using Bitsize on b followed by a right shift operation.
6: $c = b_H \cdot 2^{k-r\cdot i}$
7: Generate $d = T[b_H] \cdot 2^{n-r\cdot i}$. ▷ Using the table T 7
8: $b = b - c + d \cdot 2^{n-r\cdot i}$ ▷ Reduction by $r-1$-bit
9: **end for**
10: $b = \text{finalstep}(b, p, r)$ ▷ Done using a iterative subtraction 8.
11: **return** b

Proof. Let a be a k-bit integer represented using $\mathcal{D}(a)$ and p be an n-bit prime. For a fixed value of r, k and n, the algorithm operates in exactly $(k-n)/r + 1$ iterations, with each iteration consisting of constant-time sub-operations such as constant-time addition, subtraction and shifts. Since the number of iterations is bounded for a fixed k, n and r, the overall execution steps remain constant relative to input variations within the bound.

Now, we prove correctness. Let, $l = (k-n)/r+1$ and let $i \in \{1, 2, \ldots, l\}$. We define b_H as the most significant r bits of b and b_l the remaining bits of b, such that, $b = b_H \cdot 2^{k-n\cdot i}$. We set d as $T[b_H] \cdot 2^{n-r\cdot i}$. We prove that step 8 ensures that the remaining bits are correctly updated while reducing b by $(r-1)$-bits modulo p. Suppose, $D = d \cdot 2^{k-n-r\cdot i}$ and $C = b_H \cdot 2^{k-r\cdot i} \pmod p$. Now, consider the following equation:

$$\begin{aligned} D - C \pmod p &= T[b_H] \cdot 2^{k-n-r\cdot i} - b_H \cdot 2^n \cdot 2^{k-n-r\cdot i} \\ &= (b_H \cdot 2^n \pmod p - b_H \cdot 2^n) \cdot 2^{k-n-r\cdot i} \pmod p \\ &= 0 \pmod p \end{aligned}$$

Subtracting by $C = b_H \cdot 2^{k-r\cdot i}$ refers to removing the r most significant bit of b, while adding $D = d \cdot 2^{k-n-r\cdot i}$ is used as a correction terms to ensure that the result is correctly reduced. This implies that the update maintains the correct modular equivalence at each step. The size of $(b - b_H \cdot 2^{k-r\cdot i})$ is $(k - r \cdot i)$-bit. However, $d \cdot 2^{k-n-r\cdot i}$ is also $(k - r \cdot i)$-bit. As a result, the addition of both is $(k - r \cdot i + 1)$-bit. Hence, the *for* loop in steps 4-9 updates b_i by b_{i+1} which are have the same value modulo p, but b_{i+1} is $(r-1)$-bit smaller than b_i. This means that after l iterations, the updated value of b_l is either the correct reduction of $a \pmod p$ or it is still a partial reduction of size at most $(n + r)$-bits.

Thus, the last step in 10 involves computing a conditional constant-time subtraction of b_l by p until $0 \leq b < p$. This guarantees that the final result b is the correct reduction of a modulo p. The proof of the table generation Table_reduction and Final_Step is discussed later.

Remark 1. The proposed reduction method would be better suited for hardware implementations than for software ones. As shown in Table 2, its performance in software is slower compared to the Euclidean division method, which also performs a modular reduction.

Algorithm 7 Table_reduction(a, p)

Require: p and n-bit prime and $r > 1$ a small integer.
Ensure: A table T with $\forall i, T[i] = (i \cdot 2^n) \mod p$

Set $T[0] = 0$
Set $T[1] = 2^n - p$
$m = 2^r - 1$
for $i \in [2, m]$ **do**
  $a = i \cdot 2^n - i \cdot p$
  for $j \in [0, i]$ **do**
    $t = a - p$
    $s = t_{sign}$ ▷ Test to see if subtraction creates a negative integer
    $a = a - \overline{s} \cdot p$ ▷ Constant-time conditional subtraction
  end for
  $T[i] = a$
end for
return T

Theorem 6. *The algorithm* Table_reduction(a, p) *is correct and terminates in constant-time for a fixed* r*.*

Proof. For a fixed r, the algorithm operates in exactly $(2^r - 1) \cdot 2^{r-1}$ iterations, with each iteration consisting of constant-time sub-operations such as constant-time subtraction and shifts. Thus, it is naturally constant-time.

Now we prove by induction that $T[i]$ correctly stores intermediate modular values. $T[0] = 0$ and $T[1] = 2^n - p$ are initialized correctly. Suppose $T[i]$ is correct for all $i < m$. For $i = m$:

- Compute $a = i \cdot 2^n - i \cdot p$.
- Iteratively subtract p using a conditional subtraction until $a < p$, ensuring $T[i] = (i \cdot 2^n) \pmod p$.
- Since $2^n > p$, this implies, $i \cdot 2^n - 2 \cdot i \cdot p < 0 \leq T[i] < i \cdot 2^n - i \cdot p$, and hence the computed value is valid.

Thus, the table is correctly generated.

Corollary 1. *The algorithm* Final_step(a, p) *is correct and terminates in constant-time for a fixed upper bound of* r*.*

Proof. The arguments for constant-time and correctness directly follow the proof of Theorem 6.

Algorithm 8 Final_step(a, p)

Require: An $n + r$-bit integer a, an n-bit prime p and a small integer r
Ensure: $b = a \pmod{p}$
1: $m = 2^r - 1$
2: **for** $i \in [0, m]$ **do**
3: $\quad t = a - p$
4: $\quad s = \overline{t_{sign}}$
5: $\quad a = a - s \cdot p$
6: **end for**
7: Set $b = a$
8: **return** b

4 Benchmarking results

We conducted our experiments on a Linux-based system with Ubuntu 14.2.0 operating system equipped with an 11th Gen Intel(R) Core(TM) i5-1135G7 CPU, x64-based processor and 16GB of RAM. All algorithms were implemented in C programming language and compiled using GCC version 14.2.0 with -march=native -O3 optimization flags,with Turbo Boost and Hyper-Threading disabled. Each benchmark was run and repeated 1000 times with random inputs, we measured and recorded the average and median execution times in terms of CPU cycles (cc) in Table 2.

Table 2: Average and median execution times for 1000 runs.

Size (bits)	**Average (in cc)/Median (in cc)**		
	320	640	12000
Addition	219/217	454/455	7015/6975
Subtraction	233/234	485/486	7181/7110
Multiplication	488/490	2126/2111	660028/654979
Euclidean division	3216/3195	7585/7541	767073/759785
Right shift	104/106	236/231	6665/6614
Left shift	116/117	254/253	6905/6874
Bitsize	478/478	925/928	17355/17246
GCD	1607482/1605695	5985343/5978259	$2.26 \cdot 10^9/2.26 \cdot 10^9$
Reduction	80623/79971	193507/192049	21102273/21098608

Following the summary of the execution times in 2, we present runs versus cycles plots that visually represent the performance of our algorithms across inputs of maximum bit-lengths of 320 and 640 bits. These plots illustrate the relationship between the input sizes and the execution time across 1000 runs, with the runs represented on the x-axis and the execution time (measured in CPU cycles) represented on the y-axis. The data points in each plot correspond

to the average execution time in blue and median execution times in orange, derived from 1000 repeated runs with random inputs.

Linear growth in run-time with respect to upper bound on input array size: The execution time is constant for a given upper bound n on the array size in the integer representation $\mathcal{D}(a)$ of a. It exhibits a linear scaling factor for addition, subtraction, left and right shifts algorithms, that is, the execution time for these operations increases proportionally as the upper bound n increases, as can be seen in figures 1, 2, 3, 4. Some variability due to measurement noise in a CPU and occasional outliers can be observed in the plots. Note, however, that these plots are not intended as a visual proof or evidence of constant-time behavior of our implementation. Instead, it was verified using Timecop and Valgrind.

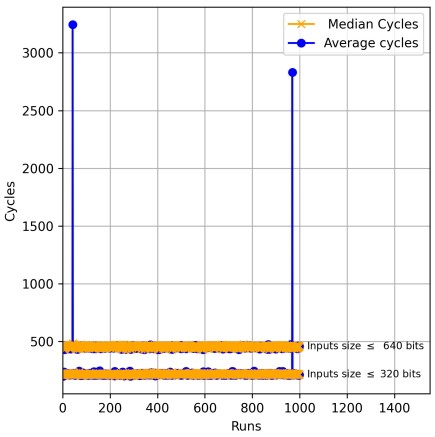


Fig. 1: Execution time of Addition

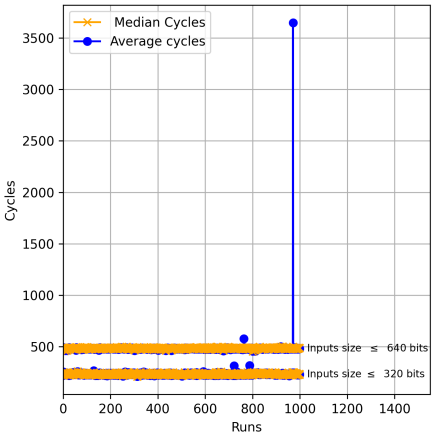


Fig. 2: Execution time of subtraction

Unscaled constant time complexity: The execution time for multiplication, GCD, Euclidean division, modular reduction and bit-size algorithms also increases as the the upper bound n grows. However, unlike the other basic functions, the run-times for these functions do not follow a straightforward linear scaling relationship with respect to the value of n. This non-linear behavior is demonstrated in plots 6, 7 for GCD, plots 9, 10 for division computation results, followed by plot 12 and plot 13 for multiplication and modular reduction, respectively.

Verification of the constant time implementation: To verify that our implementation is constant time, we used the "TIMECOP" toolkit [8] which works in combination with the "Valgrind" analysis tool [19]. These tools are

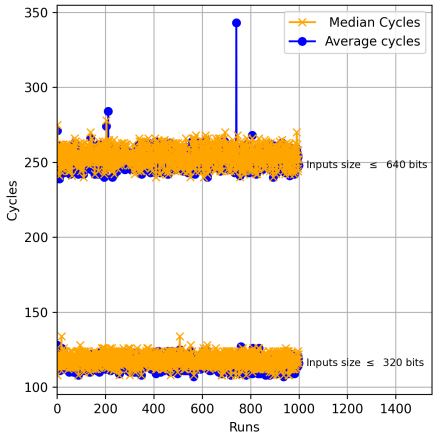


Fig. 3: Execution time of Left shift

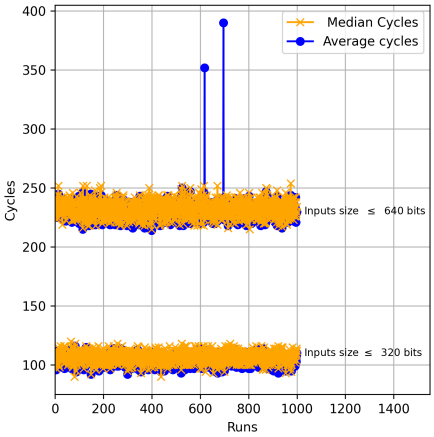


Fig. 4: Execution time of Right shift

Fig. 5: Left and right shift benchmarks for different input sizes

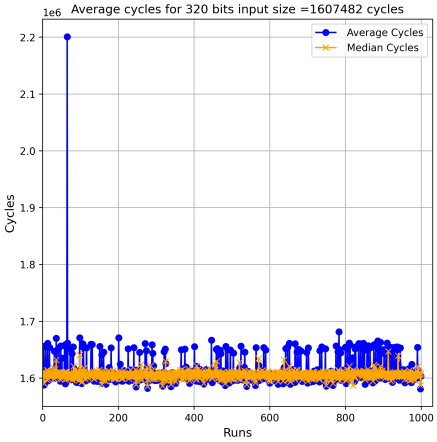


Fig. 6: GCD Benchmark for 320 bits input size

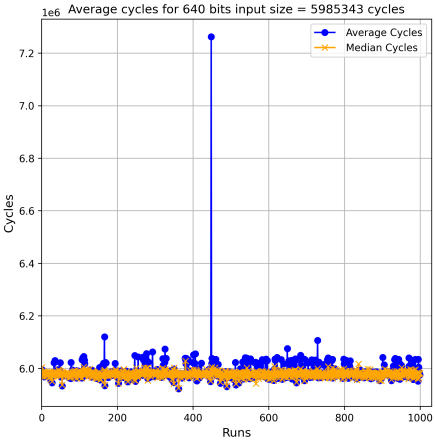


Fig. 7: GCD Benchmark for 640 bits input size

Fig. 8: GCD benchmarks for different input sizes

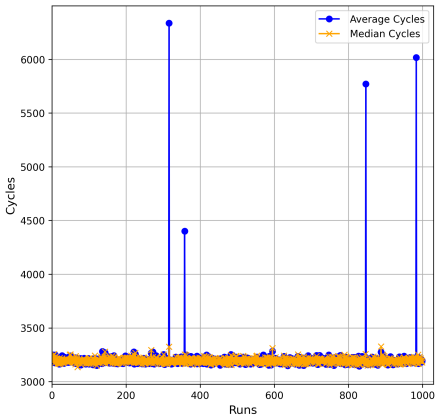


Fig. 9: Division Benchmark for 320 bits input size

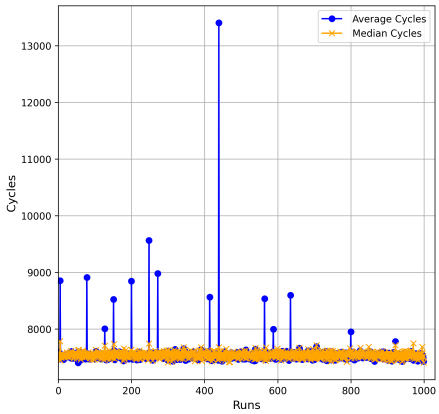


Fig. 10: Division Benchmark for 640 bits input size

Fig. 11: Division benchmarks for different input sizes

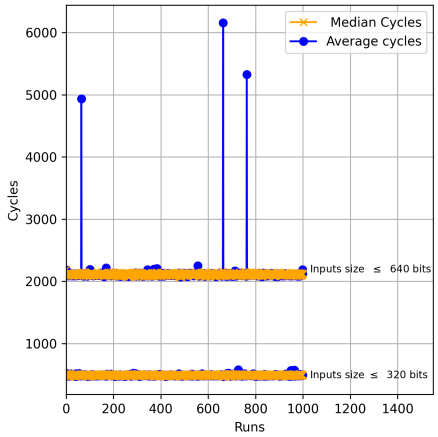


Fig. 12: Multiplication benchmark results

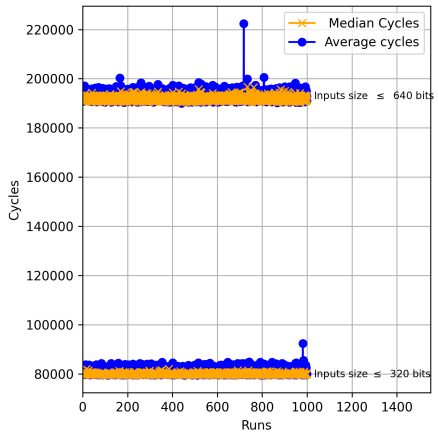


Fig. 13: Reduction benchmark results

used to detect secret-dependent branching or other secret-dependent program behavior, which we run on all of our functions successfully.

5 Conclusion

In this work, we analyzed and implemented constant-time alternatives for the main big integer functions that are used in SQIsign's quaternion arithmetic module. We modified and adapted already existing solutions according to specific requirements of quaternion arithmetic. Further, we also proposed a constant-time modular reduction algorithm that is useful when the moduli varies across operations. We implemented these functions in C programming language without depending on high-level libraries like GMP and also provided the average and median run-times of these implementations. We verified the constant-time behavior of the implementation using Timecop with Valgrind. The precision analysis for efficient implementation of the integer square root function for large integers is left for future exploration. Note that in this work, we aimed at resolving constant-time issues with the integer functions used in SQIsign's quaternion arithmetic. The next line of work in this direction would be to explore constant-time algorithms for the quaternion functions so that the implementation proposed in this work can be integrated with the quaternion module of SQIsign. With our work, we aim to contribute towards a secure and practical implementation of SQIsign, facilitating its deployment in post-quantum cryptographic applications.

Acknowledgments. Anisha Mukherjee is supported by the Austrian FWF grant PAT6402023. Péter Kutas is supported by the Hungarian Ministry of Innovation and Technology NRDI Office within the framework of the Quantum Information National Laboratory Program and by the grant "EXCELLENCE-151343". He is also supported by the János Bolyai Research Scholarship of the Hungarian Academy of Sciences and partly by EPSRC through grant number EP/V011324/1.

A Pseudo-codes for constant-time GCD and square-root computations

In this section,we will present the constant-time GCD[3] as well as the Newton-Raphson-based square root[14] pseudo-codes outlined in Section 2.5.
The constant-time GCD Algorithm algorithm 9 presents a slightly modified version of the existing constant-time GCD from [3]. We extend its functionality by handling both odd and even integers, whereas the original version was restricted to the first input being odd. We introduce step 3 along with Bitzeros constant-time function to factor out powers of 2 from even input integers.
The Newton-Raphson-based square root Algorithm 10 was originally proposed in[14]. This algorithm is based on fixed point arithmetic for a precision dependent on the bit-size of the fractional part and some additional bits $f + n$ for better accuracy.

Algorithm 9 $\mathsf{GCD}(a, b)$

1: $\mathsf{ct} - \mathsf{cmov}(f, a)$.
2: $\mathsf{ct} - \mathsf{cmov}(g, b)$.
3: $r = \mathsf{Min}(\mathsf{Bitzeros}(f), \mathsf{Bitzeros}(g))$.▷ $\mathsf{Bitzeros}$(a) returns the largest α s.t $a = 2^\alpha \cdot x$.
4: $\mathsf{Right_Shift}(f, r)$.
5: $\mathsf{Right_Shift}(g, r)$.
6: d=$\mathsf{Max}(\mathsf{Bitsize}(f), \mathsf{Bitsize}(g))$.
7: $m = \mathsf{Iteration}(d)$.
8: **if** (f_0 and 1) **then**
9: $\quad \mathsf{Divstep}(m, m + d, 1, f, g)$.
10: $\quad \mathsf{ct} - \mathsf{cmov}(c, f)$.
11: **else**
12: $\quad \mathsf{Divstep}(m, m + d, 1, g, f)$.
13: $\quad \mathsf{ct} - \mathsf{cmov}(c, g)$.
14: **end if**

Algorithm 10 $\mathsf{SQRT}(a)$

Require: $a \in \mathbb{Z}, -2^{f-1} \leq a < 2^{f-1}$, a is a 64-bit integer, represented using $\mathcal{D}(a)$ binary representation
Ensure: a boolean s if $q = \sqrt{a}$ exists and was computed .

1: $a = a \cdot 2^f$ ▷ Using constant time left shift
2: $v, v^{1/2} \leftarrow \mathsf{Scale(a)}$ ▷ $v = \pm 2^k, k$ is even integer
3: $b = 2^{-f+n} \cdot a \cdot v$
4: $\beta = (\sqrt{2} - 1)/4$
5: $\tau = 3/\sqrt{2}$
6: $\theta = \lceil log_2 \cdot \ log_{\tau \cdot \beta}(\tau \cdot 2^{-(f+n)}) \rceil$
7: $c_0 = 3/2 + \beta - \mathsf{Round}_1(b/2, \text{deterministic})$
8: **for** $i \in [1, \theta]$ **do**
9: $\quad z_1 = \mathsf{Round}_{f+n}(c_{i-1} \cdot b)$
10: $\quad z_2 = 3 - \mathsf{Round}_{f+n}(c_{i-1} \cdot z_1)$
11: $\quad c_i = \mathsf{Round}_{f+n+1}(\frac{1}{2} \cdot c_{i-1} \cdot z_2)$
12: **end for**
13: $w = \mathsf{Round}_{f-n}(a \cdot v^{\frac{1}{2}})$
14: $\tilde{q} = \mathsf{Round}_{f+2n}(c_\theta \cdot w, \text{deterministic})$ ▷ $-2^{\ell-1} \leq \tilde{q} < 2^{\ell-1}$
15: $\bar{q} = \mathsf{Round}_f(\tilde{q}, \text{deterministic})$
16: $q = \bar{q} - (\bar{q}^2 > a)$
17: $s = (q^2 == a)$ **return** s, q.

References

1. Aardal, M.A., Adj, G., Aranha, D.F., Basso, A., Andrés Canales Martínez, I., Chavez-Saab, J., Corte-Real Santos, M., Dartois, P., De Feo, L., Duparc, M., , Komada Eriksen, J., Boris Fouotsa, T., Luiz Gazzoni Filho, D., Hess, B., Kohel, D., Leroux, A., Longa, P., Maino, L., Meyer, M., Nakagawa, K., Onuki, H., Panny, L., Patranabis, S., Petit, C., Pope, G., Reijnders, K., Robert, D., Rodríguez-Henríquez, F., Schaeffler, S., Wesolowski, B.: SQISIGN: Algorithm specifications and supporting documentation. National Institute for Standards and Technology (2025)
2. Barrett, P.: Implementing the rivest-shamir and adleman public key encryption algorithm on a standard digital signal processor. In: Advances in Cryptology-CRYPTO'86. LNCS, vol. 263, pp. 311–323. Springer-Verlag (1987), https://link.springer.com/chapter/10.1007/3-540-47721-7_24
3. Bernstein, D.J., Yang, B.Y.: Fast constant-time gcd computation and modular inversion. IACR Transactions on Cryptographic Hardware and Embedded Systems **2019**(3), 340–398 (2019), https://doi.org/10.13154/tches.v2019.i3.340-398
4. Catrina, O., de Hoogh, S.: Secure multiparty linear programming using fixed-point arithmetic. In: Computer Security—ESORICS 2010, Lecture Notes in Computer Science, vol. 6345, pp. 134–150. Springer, Berlin/Heidelberg, Germany (2010), https://doi.org/10.1007/978-3-642-15497-3_9
5. Catrina, O., Saxena, A.: Secure computation with fixed-point numbers. In: Financial Cryptography and Data Security—FC 2010, Lecture Notes in Computer Science, vol. 6052, pp. 35–50. Springer, Berlin/Heidelberg, Germany (2010), https://doi.org/10.1007/978-3-642-14577-3_6
6. Chavez-Saab, J., Corte-Real Santos, M., De Feo, L., Komada Eriksen, J., Hess, B., Kohel, D., Leroux, A., Longa, P., Meyer, M., Panny, L., Patranabis, S., Petit, C., Rodríguez-Henríquez, F., Schaeffler, S., Wesolowski, B.: SQISIGN: Algorithm specifications and supporting documentation. National Institute for Standards and Technology (2023)
7. Comba, P.G.: Exponentiation cryptosystems on the ibm pc. IBM Systems Journal **29**(4), 526–538 (1990). https://doi.org/10.1147/sj.294.0526
8. Crypto, P.A.: Timecop: Automated dynamic analysis for timing side-channels. https://www.post-apocalyptic-crypto.org/timecop/
9. Granlund, T., the GMP development team: GNU MP: The GNU Multiple Precision Arithmetic Library, https://gmplib.org/
10. Hanyecz, O., Karenin, A., Kirshanova, E., Kutas, P., Schaeffler, S.: Constant time lattice reduction in dimension 4 with application to sqisign. IACR Trans. Cryptogr. Hardw. Embed. Syst. (2024), https://eprint.iacr.org/2025/027
11. Jacquemin, D., Mukherjee, A., Kutas, P., ROY, S.S.: Ready to SQI? safety first! towards a constant-time implementation of isogeny-based signature, SQIsign. Cryptology ePrint Archive, Paper 2023/807 (2023), https://eprint.iacr.org/2023/807
12. Jao, D., Azarderakhsh, R., Campagna, M., Costello, C., Feo, L.D., Hess, B., Hutchinson, A., Jalali, A., Karabina, K., Koziel, B., LaMacchia, B., Long, P., Naehrig, M., Pereira, G., Renes, J., Soukharev, V., Urbanik, D.: Sidh-spec (2022), https://sike.org/files/SIDH-spec.pdf
13. Joye, M., Yen, S.M.: The montgomery powering ladder. In: Kaliski, B.S., Koç, ç.K., Paar, C. (eds.) Cryptographic Hardware and Embedded Systems - CHES 2002. pp. 291–302. Springer Berlin Heidelberg, Berlin, Heidelberg (2003)
14. Korzilius, S., Schoenmakers, B.: Divisions and square roots with tight error analysis from newton–raphson iteration in secure fixed-point arithmetic. Cryptography **7**(3), 43 (2023), https://doi.org/10.3390/cryptography7030043

15. Longa, P., Leroux, A.: sqisign-ec23 (2023). https://github.com/SQISign/sqisign-ec23 (2023)
16. Montgomery, P.L.: Modular multiplication without trial division. Mathematics of Computation **44**(170), 519–521 (1985), https://www.ams.org/journals/mcom/1985-44-170/S0025-5718-1985-0777282-X/S0025-5718-1985-0777282-X.pdf
17. Montgomery, P.L.: Speeding the pollard and elliptic curve methods of factorization. Mathematics of Computation **48**(177), 243–264 (January 1987), https://wstein.org/edu/124/misc/montgomery.pdf
18. National Institute for Standards and Technology (NIST): Post-quantum cryptography: Additional digital signature schemes, https://csrc.nist.gov/projects/pqc-dig-sig/round-2-additional-signatures
19. Seward, J., Nethercote, N., et al.: Valgrind: A framework for heavyweight dynamic binary instrumentation. Website (2007), https://valgrind.org/
20. Shor, P.W.: Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. SIAM J. Comput. **26**(5), 1484–1509 (1997)
21. Shor, P.: Algorithms for quantum computation: discrete logarithms and factoring. In: Proceedings 35th Annual Symposium on Foundations of Computer Science. pp. 124–134 (1994). https://doi.org/10.1109/SFCS.1994.365700
22. Voight, J.: Quaternion Algebras, Graduate Texts in Mathematics, vol. 288. Springer Nature (2021), https://library.oapen.org/handle/20.500.12657/50018