A Faster Software Implementation of the Supersingular Isogeny Diffie-Hellman Key Exchange Protocol

Armando Faz-Hernández[1], Julio López[1], Eduardo Ochoa-Jiménez[2], and Francisco Rodríguez-Henríquez[2]

[1] Institute of Computing, University of Campinas.
Campinas, São Paulo, Brazil.
armfazh@ic.unicamp.br, jlopez@ic.unicamp.br
[2] Computer Science Department.
CINVESTAV-IPN, Mexico.
jochoa@computacion.cs.cinvestav.mx, francisco@cs.cinvestav.mx

Abstract

Since its introduction by Jao and De Feo in 2011, the supersingular isogeny Diffie-Hellman (SIDH) key exchange protocol has positioned itself as a promising candidate for post-quantum cryptography. One salient feature of the SIDH protocol is that it requires exceptionally short key sizes. However, the latency associated to SIDH is higher than the ones reported for other post-quantum cryptosystem proposals. Aiming to accelerate the SIDH runtime performance, we present in this work several algorithmic optimizations targeting both elliptic-curve and field arithmetic operations. We introduce in the context of the SIDH protocol a more efficient approach for calculating the elliptic curve operation $P + [k]Q$. Our strategy achieves a factor 1.4 speedup compared with the popular variable-three-point ladder algorithm regularly used in the SIDH shared secret phase. Moreover, profiting from pre-computation techniques our algorithm yields a factor 1.7 acceleration for the computation of this operation in the SIDH key generation phase. We also present an optimized evaluation of the point tripling formula, and discuss several algorithmic and implementation techniques that lead to faster field arithmetic computations. A software implementation of the above improvements on an Intel Skylake Core i7-6700 processor gives a factor 1.33 speedup against the state-of-the-art software implementation of the SIDH protocol reported by Costello-Longa-Naehrig in CRYPTO 2016.

Keywords: SIDH protocol, Montgomery ladder, post-quantum cryptography, Montgomery reduction.

1 Introduction

Over the last decade there has been an intense research effort to find hard mathematical problems that would be presumably hard to solve by a quantum attacker and at the same time could be used to build reasonably efficient public-key cryptoschemes. One such proposal is the hardness of finding an isogeny map between two elliptic curves, i.e., given two elliptic curves E_0 and E_1, the problem of finding a morphism $\phi\colon E_0 \to E_1$ that maps points from E_0 to E_1 while preserving $\phi(\mathcal{O}_{E_0}) = \mathcal{O}_{E_1}$. This proposal has spawned a new line of research generally known as isogeny-based cryptography.

Reportedly Couveignes made the first suggestions towards the usage of isogenies for cryptographic purposes in a seminar held in 1997, which he later reported in [15]. The first published work of a concrete isogeny-based cryptographic primitive was presented by Charles, Lauter and Goren in [7,8], where the authors introduced the hardness of path-finding in supersingular isogeny graphs and its application to the design of hash functions. It has since been used as an assumption for other cryptographic applications such as key-exchange and digital signature protocols.

Stolbunov studied in [35] the hardness of finding isogenies between two ordinary elliptic curves defined over a finite field $\mathbb{F}_q$, with q a prime power. The author proposed to use this setting as the underlying hard problem for a Diffie-Hellman-like key exchange protocol. Nevertheless, Childs, Jao and Soukharev discovered in [9, 10] a $L_q(\frac{1}{2}, \frac{\sqrt{3}}{2})$ subexponential complexity quantum attack against Stolbunov's scheme.

In 2011, Jao and De Feo proposed the problem of finding the isogeny map between two supersingular elliptic curves, a setting where the attack in [9] does not apply anymore. This proposal led to the Supersingular Isogeny-based Diffie-Hellman key exchange protocol (SIDH) [21] (see also [17]). As of today, the best-known algorithms against the SIDH protocol have an exponential time complexity for both classical and quantum attackers.[1]

Although the SIDH public key size for achieving a 128-bit security level in the quantum setting was already reported as small as 564 bytes in [13], this SIDH public key size was recently further reduced in [12] to just 330 bytes. However impressive, these key size credentials have to be contrasted against SIDH relatively slow runtime performance. Indeed, the SIDH key exchange protocol has a latency in the order of milliseconds when implemented in high-end Intel processors. This timing is significantly higher than the one achieved by several other quantum-resistant cryptosystem proposals. Consequently some recent works have focused on devising strategies to reduce the runtime cost of the SIDH protocol. For example, Koziel et al. presented a parallel evaluation of isogenies implemented on an FPGA architecture [24, 25], reporting important speedups for this protocol. These developments show the increasing research interest on developing techniques able to accelerate the SIDH protocol software and hardware implementations.

In order to reduce the running time of the SIDH protocol it is important to identify performance-critical operations. Upon initial inspection it is noted that this scheme computes a shared secret by performing a high number of elliptic curve and field arithmetic operations. Taking into consideration the above, our main contributions for accelerating the performance of the SIDH key exchange protocol can be summarized as follows:

- Building on the scalar multiplication procedures reported in [33], we propose a right-to-left Montgomery ladder that efficiently computes the elliptic curve scalar multiplication $P + [k]Q$ required by the two main phases of the SIDH protocol. Our strategy achieves a factor 1.4 speedup compare with the well-known three-point ladder algorithm presented in [17]. Further, when the base point Q is known in advance our algorithm can take advantage of a precomputed look-up table derived from Q, which in principle allows us to accelerate the aforementioned computation. Nevertheless, this approach led us to discover several unforeseen implementation difficulties which are somewhat related to the parameter selection used by Costello et al. in [13]. We describe how these issues were efficiently circumvented allowing us to also report a higher speedup factor for the SIDH fixed-point scalar multiplication computation.

- We present an optimized point tripling formula specialized for Montgomery elliptic curves. We consider the case when the elliptic curve parameter A that defines the elliptic curve equation is expressed as a quotient $A = A_0/A_1$. Our formulation saves one multiplication at the cost of one squaring and one addition, which are performed in the quadratic extension field $\mathbb{F}_{p^2}$. This saving is valuable if one considers that Bob has to perform hundreds of point tripling computations in both phases of the SIDH protocol.

- We developed an optimized prime field arithmetic that takes advantage of the recent instructions devoted to achieve ultra fast integer arithmetic computations, such as the Bit Manipulation

[1] See §2 for a detailed description of the SIDH protocol.

Instructions (BMI2) and the addition instructions with independent carry chains (ADX) [18] supported by high-end Intel and AMD 64-bit processors.

Combining all the above improvements, the execution of our library achieves a factor 1.33 speedup compared with the running time associated to the fastest SIDH software implementation reported in the literature (see §6 for more details about the performance achieved by our library).

The remaining of this paper is organized as follows. In §2, a brief description of the SIDH key exchange protocol and Montgomery curves and related arithmetic is given. In §3, a detailed description of a right-to-left Montgomery ladder procedure and how to adapt it to the key generation and the key exchange phases of the SIDH protocol is presented. Based on a quotient representation of the A parameter of the Montgomery elliptic curve, we present in §4 a more efficient formulation of the point tripling elliptic curve operation. In §5, an extensive description of the underlying field arithmetic implementation is given. This material focuses in the efficient implementation of integer multiplication, squaring and modular reduction. In §6, a summary of the SIDH protocol performance speedups that were achieved by applying the techniques described in this paper are given. Finally, we draw our concluding remarks and future work in §7.

2 Supersingular Isogeny Diffie-Hellman

The main purpose of the classical Diffie-Hellman protocol is that two entities securely agree on a shared secret over a public communication channel that is considered insecure under passive attacks. In the case of the SIDH protocol this secret is obtained by computing the j-invariant of two isomorphic supersingular elliptic curves generated by Alice and Bob that happens to be isogenous to an initial supersingular curve E_0.

The SIDH domain parameters are given as follows. Choose a supersingular elliptic curve E over $\mathbb{F}_{p^2}$, where p is a large prime of the form[2],

$$p = (l_A)^{e_A}(l_B)^{e_B} f \pm 1, \tag{1}$$

and where l_A and l_B are small prime numbers, e_A and e_B are positive integers, and f is a small cofactor. Then the cardinality of E is given as, $\#E = ({l_A}^{e_A}{l_B}^{e_B}f)^2$. To simplify the notation let us define $r_A = {l_A}^{e_A}$ and $r_B = {l_B}^{e_B}$. One then chooses two pairs of independent elliptic curve points so that the subgroups $E[r_A]$ and $E[r_B]$ are generated as, $\langle P_A, Q_A\rangle = E[r_A]$ and $\langle P_B, Q_B\rangle = E[r_B]$. Notice that the prime p, the curve E and the generating points P_A, Q_A, P_B, and Q_B, are all considered public domain parameters.

Given a point R of order l^e, an isogeny $\phi\colon E \to E/\langle R\rangle$ is calculated as a composition of l-degree isogenies $\phi = \phi_{e-1} \circ \cdots \circ \phi_0$, as follows[3]. Let $E_0 = E$ and $R_0 = R$, then for $0 \le i < e$, compute: $E_{i+1} = E_i/\langle l^{e-i-1}R_i\rangle$, $\phi_i\colon E_i \to E_{i+1}$, and $R_{i+1} = \phi_i(R_i)$. Thus, $E/\langle R\rangle = E_e$. Using the point $l^{e-i-1}R_i$, the curve E_{i+1} and the isogeny ϕ_i can be readily computed in polynomial time by means of Vélu's formulas (see [38, theorem 12.16]).

The SIDH key exchange protocol consists of two main phases. In the first one, also known as the key generation phase, both parties proceed as follows:

- Alice selects two random numbers $m_A, n_A \in \mathbb{Z}_{r_A}$ and computes the isogeny $\phi_A\colon E \to E_A$ with kernel $\langle m_A P_A + n_A Q_A\rangle = \langle R_A\rangle$. Then Alice calculates $\{\phi_A(P_B), \phi_A(Q_B)\}$ and sends to Bob these points together with her computed curve E_A.

[2] Some primes used in the SIDH protocol are Pierpont primes (a prime number of the form $p = 2^i 3^j + 1$, for $i, j > 0$, is known as a Pierpont prime [6, 34]) By extending this definition, a *generalized Pierpont (GP)* prime has the form $p = \pm 1 + \Pi_{i=1}^{k}(p_i)^{e_i}$, where p_i are distinct primes.

[3] See [17] for a comprehensive discussion on optimal approaches for computing a l^e-degree isogeny.

- Analogously, Bob selects two random numbers $m_B, n_B \in \mathbb{Z}_{r_B}$ and computes the isogeny $\phi_B \colon E \to E_B$ with kernel $\langle m_B P_B + n_B Q_B \rangle = \langle R_B \rangle$. Then Bob calculates $\{\phi_B(P_A), \phi_B(Q_A)\}$ and sends to Alice these points together with his computed curve E_B.

In the second phase of the protocol, Alice and Bob compute a shared secret as follows:

- Once Alice receives $\{E_B, \phi_B(P_A), \phi_B(Q_A)\}$ from Bob, she calculates the isogeny $\phi_{AB} \colon E_B \to E_{AB}$ that has kernel $\langle m_A \phi_B(P_A) + n_A \phi_B(Q_A) \rangle = \langle \phi_B(R_A) \rangle$. Finally Alice obtains the shared secret as the j-invariant of E_{AB}.
- Once Bob receives $\{E_A, \phi_A(P_B), \phi_A(Q_B)\}$ from Alice, he calculates the isogeny $\phi_{BA} \colon E_A \to E_{BA}$ that has kernel $\langle m_B \phi_A(P_B) + n_B \phi_A(Q_B) \rangle = \langle \phi_A(R_B) \rangle$. Finally Bob obtains the shared secret as the j-invariant of E_{BA}.

One can instantiate the SIDH protocol using different elliptic curve forms such as the Edwards or the Montgomery curves. In this work we will focus in the latter form due to its generally more efficient isogeny and elliptic curve arithmetic operations.

2.1 Montgomery Curves and Their Arithmetic

We report the computational cost of the main elliptic curve operations in terms of field arithmetic operations. As usual, we denote with **M**, **S**, **A**, and **I** the cost of one multiplication, squaring, addition (or subtraction), and multiplicative inverse in the field $\mathbb{F}_q$, respectively. Note that $q = p^2$, unless otherwise specified.

Given a finite field $\mathbb{F}_q$, Montgomery elliptic curves are defined by the equation:

$$E/\mathbb{F}_q\colon \quad By^2 = x^3 + Ax^2 + x\,, \tag{2}$$

such that $A, B \in \mathbb{F}_q$, $A^2 \neq 4$ and $B \neq 0$. The set of solutions of (2) plus the neutral element $\mathcal{O}$ (called the point at infinity) form an abelian additive group. The negative of a point (x, y) is $(x, -y)$. The point $P = (0, 0)$ satisfies (2) and has order two since $P + P = \mathcal{O}$. The j-invariant of E is calculated as,

$$j(E) = 256\frac{(A^2-3)^3}{A^2-4}\,. \tag{3}$$

Moving to projective coordinates ($\mathbb{P}^2$) implies that an affine point (x, y) is represented by $(\lambda X\colon \lambda Y\colon \lambda Z)$ such that $\lambda \neq 0$, $x = X/Z$ and $y = Y/Z$. The point at infinity is a special case that is written as $\mathcal{O} = (0\colon 1\colon 0)$. Points can also be mapped to $\mathbb{P}^1$ using[4]

$$\begin{array}{rccc} \chi\colon & \mathbb{P}^2 & \to & \mathbb{P}^1 \\ & (X\colon Y\colon Z) & \mapsto & (X\colon Z) \\ & \mathcal{O} & \mapsto & (1\colon 0)\,. \end{array} \tag{4}$$

We write $\chi(P) = (X\colon Z) \in \mathbb{P}^1(\mathbb{F}_p)$ when both X and Z belong to the field $\mathbb{F}_p$.

In his landmark paper [30], Montgomery introduced the concept of a *differential* addition operation, which given $\chi(P)$, $\chi(Q)$, and $\chi(P-Q)$, calculates $\chi(P+Q)$. It is noticed that this formula fails whenever $P - Q \in \{\mathcal{O}, (0,0)\}$ (refer to [4] for a formal proof).

Let $P, Q \in E(\mathbb{F}_q)$ and $R_0, R_1, R_2 \in \mathbb{P}^1$. Then we denote a point doubling operation as $[2]R_0$ and a differential addition as $R_0 +_{(R_2)} R_1$, such that $R_0 = \chi(P)$, $R_1 = \chi(Q)$, and $R_2 = \chi(P-Q)$. A differential addition can be computed at a cost of 4**M**+2**S**, whereas a point doubling takes 2**M**+2**S** [14]. When performing a differential addition one multiplication can be saved whenever $Z_{P-Q} = 1$.

[4] This map sets $\mathcal{O} = (1\colon 0)$ since $(0\colon 0) \notin \mathbb{P}^1$, see [14, Section 3].

Montgomery also introduced in [30] a procedure that calculates $\chi([k]P)$ from $\chi(P)$ and an integer k. This procedure is better known as the Montgomery ladder. A high level description of the Montgomery ladder is shown in Algorithm 1. To recover the y-coordinate of $[k]P$ one can use the Okeya-Sakurai technique [31], which extends the y-recovery formula of López-Dahab that applies to the binary elliptic curve case [28]. The Okeya-Sakurai formula calculates the y-coordinate of $[k]P$ from the y-coordinate of P, $\chi([k]P)$ and $\chi([k+1]P)$ (this latter value is also computed by the Montgomery ladder).

Algorithm 1 Montgomery ladder algorithm.

```
Input: $(k, \chi(P))$, where $k$ is a $t$-bit number, and $\chi(P) \in \mathbb{P}^1$ is a representation of $P \in E(\mathbb{F}_q)$.
Output: $\chi([k]P) \in \mathbb{P}^1$.
Initialize $R_0 \leftarrow \chi(\mathcal{O})$, $R_1 \leftarrow \chi(P)$, and $R_2 \leftarrow \chi(P)$.
for $i \leftarrow t-1$ to 0 do
    if $k_i = 1$ then
        $(R_0, R_1) \leftarrow (R_0 +_{(R_2)} R_1, 2R_1)$
    else
        $(R_0, R_1) \leftarrow (2R_0, R_0 +_{(R_2)} R_1)$
    end if
end for
return $R_0$                                   // For $y$-coordinate recovery, return also $R_1$.
```

The Montgomery ladder processes the scalar k from the most significant to the least significant bit updating at each iteration the accumulators R_0 and R_1. The bits of the scalar determine which of the accumulators is updated by the results of the point doubling or the differential addition operations. Each step of the ladder performs the same number of operations preserving the relation $R_0 - R_1 = P$. This is an advantageous property, since usually the scalar k is a secret value. Therefore, a regular execution pattern helps to prevent threats caused by some simple side-channel attacks. Let k be a t-bit number. Then Algorithm 1 takes $5t\mathbf{M}+4t\mathbf{S}$, which in practice translates to a 7.6 $\mathbf{M}$-per-bit cost, under the assumption that $1\mathbf{S} \approx 0.66\mathbf{M}$ in $\mathbb{F}_q$.

Performing all elliptic curve operations in $\mathbb{P}^1$ minimizes the use of multiplicative inverses, which tend to be quite costly. Hence, it is desirable to perform the scalar multiplication computations required by the SIDH using this strategy.

3 Efficient Computation of $P + [k]Q$

At each stage of the SIDH protocol, Alice and Bob must compute the kernel of an isogeny by calculating the point $[m]P + [n]Q$, where P and Q are linearly independent points of order r; and $n, m \in \mathbb{Z}_r$ are secret values. Invoking efficiency reasons, De Feo et al. [17] suggested to compute instead the point $P + [nm^{-1}]Q$, for a scalar m that has a multiplicative inverse modulo r. Therefore, the isogeny's secret kernel is generated by performing the operation $P + [k]Q$ for some $k \in \mathbb{Z}_r$.

Since it is generally more efficient to perform the SIDH scalar point multiplications using $\mathbb{P}^1$ arithmetic, we will review in the following two common strategies to compute $\chi(P + [k]Q)$.

Method 1: Given P, Q, and k, use the classical Montgomery ladder (Algorithm 1) for computing the x-coordinate of $[k]Q$ followed by the application of the Okeya-Sakurai formula to recover the y-coordinate of $[k]Q$. Finally, perform a projective point addition of the points $(x_P : y_P : 1)$ and $(X_{[k]Q} : Y_{[k]Q} : Z_{[k]Q})$ to obtain $\chi(P + [k]Q)$. This strategy requires the knowledge of the y-coordinate of the points P and Q. The time computational expense of this algorithm is given by the execution cost of the Montgomery ladder plus a constant number of prime field multiplications ($< 30\,\mathbf{M}$).

Method 2 (three-point ladder): In [17], De Feo et al. proposed a three-point ladder procedure that given the x-coordinate of the points P, Q, and $Q - P$, computes $\chi(P + [k]Q)$. This method performs two differential additions and one doubling per bit of the scalar k. This is the same number of operations as computing $[m]P + [n]Q$ using the Bernstein's two-dimensional ladder algorithm [3]. One advantage of the three-point ladder is that all elliptic curve operations are performed using only the x-coordinate of the involved points.

To improve the computation of the SIDH protocol, these two methods can be combined as follows. Notice that during the SIDH key generation phase the initial points are fixed. This situation allows us to apply the Method 1 efficiently since the y-coordinate of the points are known in advance. On the other hand, during the SIDH shared secret phase Alice and Bob exchange points in $\mathbb{P}^1$. Hence, in order to use Method 1 Alice must recover the y-coordinate of the points sent by Bob. However, this will increase the protocol's latency and/or bandwidth. Therefore, Method 2 emerges as a suitable alternative for this scenario, and in fact the three-point ladder algorithm has been adopted by most if not all state-of-the-art implementations of the SIDH protocol (see [12, 13, 16, 26]).

In the following section we introduce in the context of the SIDH protocol, novel strategies for computing $\chi(P + [k]Q)$. Our approach performs fewer elliptic curve operations than the methods presented above. Moreover, our algorithms can be used to improve the running time of both, the key generation and the shared secret computation phases of the SIDH protocol.

3.1 A Novel Algorithm for Computing $\chi(P + [k]Q)$

The Montgomery ladder (Algorithm 1) is known as a *left-to-right* algorithm, since it computes $[k]P$ by scanning the bits of the scalar k from the most-significant to the least-significant bit. A *right-to-left* evaluation of the Montgomery ladder was recently introduced to accelerate the scalar multiplication operation in the fixed-point scenario. This approach was first applied in the contex of binary elliptic curves [22,32], and then, it was further extended to Montgomery curves [33]. Building on the right-to-left ladder technique of [33] we present here an algorithm that computes $\chi(P + [k]Q)$ efficiently.

Algorithm 2 Variable-point multiplication of $\chi(P + [k]Q)$.

Input: $(k, \chi(P), \chi(Q), \chi(Q - P))$, where k is a t-bit number; and $\chi(P), \chi(Q), \chi(Q - P) \in \mathbb{P}^1$ are a representation of $P, Q, Q - P \in E(\mathbb{F}_q)$, respectively.

Output: $\chi(P + [k]Q)$.

```
Initialize $R_0 \leftarrow \chi(Q)$, $R_1 \leftarrow \chi(P)$ and $R_2 \leftarrow \chi(Q - P)$
for $i \leftarrow 0$ to $t - 1$ do
    if $k_i = 1$ then
        $R_1 \leftarrow R_0 +_{(R_2)} R_1$
    else
        $R_2 \leftarrow R_0 +_{(R_1)} R_2$
    end if
    $R_0 \leftarrow [2]R_0$
end for
return $R_1$
```

The proposed approach is shown in Algorithm 2, which given the points $\chi(P)$, $\chi(Q)$, and $\chi(Q - P)$ computes $\chi(P + [k]Q)$ provided that $P, Q - P \notin \{\mathcal{O}, (0, 0)\}$. Algorithm 2 uses three accumulators, namely $R_0, R_1, R_2 \in \mathbb{P}^1$, and scans the bits of k from the least-significant to the most-significant bit. At the i-th iteration, the k_i bit value determines whether R_0 must be accumulated in R_1 or in R_2. Thereafter R_0 is doubled unconditionally. Accumulators are updated by one differential addition and one point doubling and they always preserve the relation $R_0 - R_1 = R_2$. This is the same invariant relation that holds for the classical Montgomery ladder of Algorithm 1. However notice that in the case of Algorithm 2, the value stored in R_2 may vary between iterations, unlike Algorithm 1 where R_2

is always fixed to $\chi(P)$. In summary, Algorithm 2 performs $6t\mathbf{M}+4t\mathbf{S}$, which for a practical software implementation implies a cost of approximately $8.6\,\mathbf{M}$-per-bit.

As observed in [33], in the case that Q is a fixed point known in advance, one can construct a look-up table $T(Q)$ by pre-computing constants that are obtained from the x-coordinate of multiples of the point Q as,

$$\begin{aligned} T(Q) = (T_0, \ldots, T_{t-1}), \quad \text{where } T_i &= \frac{x_i+1}{x_i-1}, \text{ and} \\ (x_i, y_i) &= [2^i]Q, \text{ for } 0 \le i < t, \end{aligned} \tag{5}$$

where t is the size in bits of the scalar k. Using this approach, the point doubling computation in line 8 of Algorithm 2 can be replaced by a query to the look-up table T. For completeness, we show in Algorithm 3 the fixed-point version of Algorithm 2. The differential additions in lines 4 and 6 of Algorithm 3 are computed more efficiently (using $3\mathbf{M}+2\mathbf{S}$) using the precomputed value T_i as input [33]. It is worth to mention that the look-up table queries of Algorithm 3 use non-secret indexes. This is in stark contrast with other fixed-point multiplication algorithms where protecting look-up table accesses is mandatory, a measure that unavoidably introduces performance overheads. The computational cost of Algorithm 3 for computing $\chi(P+[k]Q)$ drops to $3t\mathbf{M}+2t\mathbf{S}$, which is around $4.3\,\mathbf{M}$-per-bit.

Algorithm 3 Fixed-point multiplication of $\chi(P+[k]Q)$.

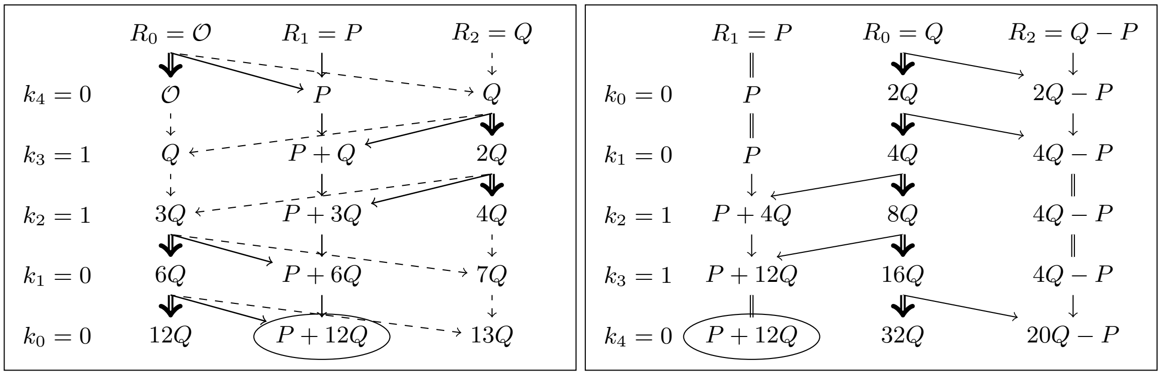
Input: $(k, \chi(P), \chi(Q-P))$, where k is a t-bit number; and $\chi(P), \chi(Q-P) \in \mathbb{P}^1$ are a representation of $P, Q-P \in E(\mathbb{F}_q)$, respectively.
Precomputation: $T(Q)$ is a look-up table defined as in Eq. 5.
Output: $\chi(P+[k]Q)$.

```
1: Initialize $R_1 \leftarrow \chi(P)$, and $R_2 \leftarrow \chi(Q-P)$.
2: for $i \leftarrow 0$ to $t-1$ do
3:     if $k_i = 1$ then
4:         $R_1 \leftarrow T_i +_{(R_2)} R_1$
5:     else
6:         $R_2 \leftarrow T_i +_{(R_1)} R_2$
7:     end if
8: end for
9: return $R_1$                // For $y$-coordinate recovery, return also $R_2$.
```

As in the case of the classical Montgomery ladder we show how to recover the y-coordinate of $P+[k]Q$ from the values computed by the right-to-left ladder algorithm. This method will be discussed at length in §3.3.

3.2 Applying the New Algorithm to the SIDH Protocol

As already mentioned, the $P+[k]Q$ operation must be performed in both phases of the SIDH protocol. During the key generation phase, this operation uses points that are known in advance. Conversely, during the shared secret generation phase operations are performed over unknown points. In the remaining of this section we describe the application of our algorithms to these scenarios and we also discuss some relevant issues that appeared on their implementations. We present first the description of the variable point case.



(a) Three-point ladder from [17].

(b) Algorithm 2 (This work)

Figure 1: Calculating $P + [12]Q$, where the scalar is a 5-bit number $(12)_{10} = (01100)_2$. In (a) we show the steps for the three-point ladder algorithm, and in (b) the steps for the ladder of Algorithm 2. If we remove the central column in (a), it becomes clear that the three-point ladder procedure is in essence a classical Montgomery ladder. Also note that the column in the center of (b) shows a sequence of consecutive point doublings of Q. When Q is a fixed point, this column can be precomputed.

3.2.1 Computing $P + [k]Q$ in the Variable-point Scenario

During the shared secret generation phase, Alice[5] receives $\chi(\phi_B(P_A))$, $\chi(\phi_B(Q_A))$ from Bob. Afterwards, Alice must calculate $\chi(\phi_B(P_A) + [n_A]\phi_B(Q_A))$. To that end, Method 1 could be applied in this scenario. Nevertheless, this would require that Alice must know the y-coordinate values of the points $\phi_B(P_A)$ and $\phi_B(Q_A)$. To circumvent this difficulty, Bob could send the y-coordinate of these two points, but this would increase the public key sizes considerably. Alternatively, Bob could encode the y-coordinate of each point into one bit. However, this would force Alice to decompress a point using time-consuming square-roots over $\mathbb{F}_q$. We conclude that Method 1 becomes an inadequate choice for this scenario.

On the other hand if Bob additionally sends $\chi(\phi_B(P_A - Q_A))$, then Alice can perform the three-point ladder algorithm [17] (corresponding to Method 2 described in the previous section). This is the mechanism followed by most of the state-of-the-art implementations, such as [12,13,16,26]. Nevertheless notice that the three-point ladder algorithm has a higher computational cost as compared to Method 1.

A more efficient approach consists of applying Algorithm 2 since it provides a significant saving of field arithmetic operations when compared to Method 1 or Method 2. Furthermore given the same input values, Algorithm 2 and the three-point ladder procedure produce the same output. This implies that both algorithms can share the same interface, which is especially valuable for minimizing the changes of existent software implementations. An extra advantage of adopting Algorithm 2 is that it does not increase the public key size. Figure 1 shows an example contrasting the execution of Algorithm 2 and the three-point ladder procedure when processing the same scalar $k = 12$.

By replacing the three-point ladder algorithm with Algorithm 2, we estimate to achieve a factor 1.38 speedup (cf. Table 1). In §6, this prediction is experimentally verified through the benchmarking of our SIDH protocol implementation.

[5] Since the same analysis applies to Bob by just swapping sub-indexes, in this section we only summarize the operations performed by Alice.

3.2.2 Computing $P+[k]Q$ in the Fixed-point Scenario

In this phase, one can exploit more avenues for further optimizations. First of all, the involved points are fixed, allowing us to use precomputed look-up tables that could possibly accelerate operations. In this scenario it is clear that Method 1 is more efficient than Method 2. However, it is not obvious how these two strategies could benefit from pre-computation techniques.

Taking advantage of the fact that the points are known in advance one can directly apply Algorithm 3. Using this procedure one can expect a factor 1.77 speedup when compared with Method 1 (cf. Table 1). Hence, the SIDH key generation phase can be also accelerated in a similar way as it happens in the ECDH protocol implementation reported in [33].

Algorithm 3 was designed to perform arithmetic operations over $\mathbb{F}_q$, where in the general setting $q=p^2$. However, Algorithm 3 can also be useful in the setting $q=p$. For the rest of this section, we adhere to the elliptic curve parameters proposed by Costello et al. [13].

Let $P_A=(x_{P_A},y_{P_A})$ and $P_B=(x_{P_B},y_{P_B})$ be the base points of Alice and Bob, respectively. By construction, the authors of [13] selected P_A and P_B in such a way that their affine coordinates lie in $\mathbb{F}_p$. Using the distortion map action, they obtained the points $Q_A=(-x_{P_A},y_{P_A}i)$ and $Q_B=(-x_{P_B},y_{P_B}i)$, which happen to be linearly independent to P_A and P_B, respectively. Let us recall that Alice's points have order $r_A=2^{e_A}$, while Bob's points have order $r_B=3^{e_B}$. Under these conditions, Costello et al. [13] employed the following method to compute $\chi(P+[k]Q)$ in the fixed-point scenario.

Method 3: Compute the Montgomery ladder algorithm in $\mathbb{F}_p$ to obtain $\chi([k]Q)\in\mathbb{P}^1(\mathbb{F}_p)$. Then recover the y-coordinate of $[k]Q$, and finally perform a projective point addition to obtain $\chi(P+[k]Q)$. The cost of Method 3 is of about $8.2\,\widehat{\mathbf{m}}$-per-bit, where $\widehat{\mathbf{m}}$ denotes a multiplication in $\mathbb{F}_p$. Assuming $1\mathbf{M}=3\widehat{\mathbf{m}}$, one concludes that Method 3 takes around one third of the cost of Method 1.

Sticking to the same conditions, we consider to perform operations of Algorithm 3 over $\mathbb{F}_p$. To that end, this procedure requires $\chi(Q-P)\in\mathbb{P}^1(\mathbb{F}_p)$. Unfortunately, this is not the case for the points selected using the parameter generation of [13]. Indeed, note that one of the projective coordinates of $\chi(Q-P)=((x_P^2+1)i\colon 2x_P)$ is not in $\mathbb{F}_p$. As a consequence, it would appear that the point selection method used in [13] restricts the use of Algorithm 3.

However, not all is lost. We prevent these issues and propose an alternative solution that combines the efficiency offered by Algorithm 3 and the parameter selection described above.

Our idea consists of employing Algorithm 3 to compute $\chi(S+[k']Q)$ for an order-d point $S\in E(\mathbb{F}_{p^2})$, such that $\chi(Q-S)\in\mathbb{P}^1(\mathbb{F}_p)$ and $k'\equiv k/d \mod r$. Thereafter one can compute $\chi([k]Q)=[d]\chi(S+[k']Q)$, followed by the recovery of the y-coordinate of $[k]Q$, and finally the addition of the point P (as in Method 3) to end up with $\chi(P+[k]Q)$. We must ensure that $S\notin\langle Q\rangle$, and for efficiency reasons we also impose the restriction that the point S must have a low order $d\neq 2$. These steps are summarized in Algorithm 4.

Looking for a suitable point S the most natural choice to obtain low order points is that Alice uses Bob's points and vice versa. Hence, let us define the following points,

$$S=\begin{cases} S_A=[3^{e_B-1}]Q_B\,, & \text{for Alice;}\\ S_B=[2^{e_A-2}]Q_A\,, & \text{for Bob.}\end{cases} \tag{6}$$

By construction S_A and S_B were chosen such that both $\chi(Q_A-S_A)$ and $\chi(Q_B-S_B)$ are in $\mathbb{P}^1(\mathbb{F}_p)$.

The cost of Algorithm 4 is similar to the cost of Algorithm 3 plus a constant number of multiplications ($<30\,\mathbf{M}$). However, the scalar multiplication operations are performed over $\mathbb{F}_p$ resulting in a cost of approximately $4.6\,\widehat{\mathbf{m}}$-per-bit. Thus, Algorithm 4 provides an acceleration of a factor 1.78 speedup compared to the performance of Method 3.

Table 1 summarizes the computational costs of the algorithms discussed in this section. We considered two scenarios: the first one is when points are fixed and known in advance; and the second one when dealing with unknown points. For both scenarios our methods outperform the techniques used in

Algorithm 4 Proposed algorithm to compute $\chi(P + [k]Q)$ in the fixed-point scenario and adapted to the elliptic curve parameters defined in [13]. Let $I \in \{A = \text{Alice}, B = \text{Bob}\}$ denote the SIDH protocol participant.

Input: (k, t_I, P_I, Q_I, S_I), where k is an t_I-bit number ($t_A = 372$ and $t_B = 379$), P_I and Q_I are points of order r_I (defined as in [13]), and S_I is a point of order d_I (defined as in Eq. 6).

Precomputation: Compute a look-up table $T(Q_I)$ defined as in Eq. 5. Compute $U_0, U_1, V_0, V_1 \in E(\mathbb{F}_{p^2})$ defined as in Eq. 8.

Output: $\chi(P_I + [k]Q_I)$.

```
$k' \leftarrow k/d_I \mod r_I$
if $I =$ Alice then                                                  // Section 3.3.1.
    $(\alpha, \beta, k') \leftarrow (k'_{e_A-1}, k'_{e_A-2}, k' \mod 2^{e_A-2})$
end if
$R_1, R_2 \leftarrow$ ALGORITHM3$_{q=p,T(Q_I)}(k', \chi(S_I), \chi(Q_I - S_I))$
                                                          // $R_1 = \chi(S_I + [k']Q_I)$
$R_1 \leftarrow [d_I]R_1$, $R_2 \leftarrow [d_I]R_2$
$(X_{R_1} : Y_{R_1} : Z_{R_1}) \leftarrow y$-RECOVER$(R_1, R_2)$                 // Section 3.3.
if $I =$ Alice then                                                  // Section 3.3.1.
    $U \leftarrow$ CMOVE$(\alpha, U_0, U_1)$
    $V \leftarrow$ CMOVE$(\beta, V_0, V_1)$
    $R_3 \leftarrow (X_{R_1} : Y_{R_1} : Z_{R_1}) + U + V$
else if $I =$ Bob then
    $R_3 \leftarrow (X_{R_1} : Y_{R_1} : Z_{R_1}) + P_B$
end if
return $\chi(R_3)$
```

state-of-the-art implementations [12,13,16,26]. In §6, we report the impact yielded by these algorithms on the SIDH protocol overall performance.

3.3 Recovering the y-coordinate of $P + [k]Q$

As in the classical Montgomery ladder, one can recover the y-coordinate of $P + [k]Q$ using the values computed in the last iteration of the right-to-left algorithm. This can be done by restating the formula given in Okeya-Sakurai's paper [31, Corollary 2] as discussed next.

Let us consider an affine point (x, y) with $y \neq 0$, and the points $P_i = (X_i \colon Z_i) \in \mathbb{P}^1$, for $i = 1, 2, 3$ such that, $(X_2 \colon Z_2) = (X_1 \colon Z_1) - (x, y)$, and $(X_3 \colon Z_3) = (X_1 \colon Z_1) + (x, y)$. Then one can compute,

$$\begin{aligned} X_1' &= 4ByZ_1Z_2Z_3X_1 \\ Y_1' &= (X_2Z_3 - Z_2X_3)(X_1 - Z_1x)^2 \\ Z_1' &= 4ByZ_1Z_2Z_3Z_1\,, \end{aligned} \tag{7}$$

such that the point $(X_1' \colon Y_1' \colon Z_1') \in \mathbb{P}^2$ belongs to the same equivalence class of the point $(X_1 \colon Z_1) \in \mathbb{P}^1$.

Recall that the loop-invariant of the right-to-left ladder is $R_0 - R_1 = R_2$. Thus, the accumulators in the i-th iteration hold the values $R_0 = [2^i]Q$, $R_1 = P + [k \mod 2^i]Q$, and $R_2 = [2^i - (k \mod 2^i)]Q - P$, respectively. Since k is a t-bit number, then after t iterations one can compute $R_3 = R_0 +_{(R_2)} R_1$ and use the points stored in those four accumulators to apply Eq. 7 as: $(x, y) \leftarrow R_0$; $(X_1 \colon Z_1) \leftarrow R_1$; $(X_2 \colon Z_2) \leftarrow R_2$; and $(X_3 \colon Z_3) \leftarrow R_3$. This allows the recovery of the y-coordinate of the point $R_1 = P + [k]Q$. The cost of the y-coordinate recovering just described is one differential addition more than the original Okeya-Sakurai technique. Thus, the only requirement is to have a previous knowledge of the point $[2^t]Q$.

Table 1: Algorithms for computing $\chi(P+[k]Q)$ in the fixed- and variable-point scenario. The third column shows ladder step arithmetic operation costs and the fourth column shows the predicted acceleration factor. We assume that $1\mathbf{M} = 3\widehat{\mathbf{m}}$, $1\mathbf{S} = 0.66\mathbf{M}$, and $1\widehat{\mathbf{s}} = 0.8\widehat{\mathbf{m}}$.

Scenario	Field	Mult-per-bit	AF	Algorithms
Fixed-point	$\mathbb{F}_{p^2}$	$7.6\,\mathbf{M}$		Method 1
		$4.3\,\mathbf{M}$	1.77	**Alg. 3 (this work)**
	$\mathbb{F}_p$	$8.2\,\widehat{\mathbf{m}}$		Method 3 [13]
		$4.6\,\widehat{\mathbf{m}}$	1.78	**Alg. 4 (this work)**
Variable-point	$\mathbb{F}_{p^2}$	$11.9\,\mathbf{M}$		3-point ladder [17]
		$8.6\,\mathbf{M}$	1.38	**Alg. 2 (this work)**

In the fixed-point scenario, the point $[2^t]Q$ can be saved together with the look-up table constants. This enables the usage of Algorithm 3 as a subroutine of Algorithm 4 for accelerating the $P+[k]Q$ operation in the fixed-point scenario. Nonetheless, the fact that Alice uses points of 2-smooth order produces some troubles for recovering the y-coordinate of $[k]Q_A$. We dedicate the next subsection for exposing this issue and the solution that we found to it.

3.3.1 An Implementation Issue: Alice's y-Coordinate Recovering

We found a subtle issue when Alice tries to recover the y-coordinate of $[k]Q_A$. Since Q_A has order 2^{e_A} then $R_0 = [2^i]Q_A = \mathcal{O}$ for all $i \geq e_A$. Note that for a t-bit scalar k, the point $R_0 = [2^t]Q$ is directly involved in the recovery of the projective coordinates of the point $P+[k]Q$. Hence, after running e_A steps of the right-to-left ladder we end up having $y_{R_0} = 0$, which makes the usage of Eq. 7 impossible. In order to overcome this problem we propose the solution described in Algorithm 4. The main idea consists of running only t' iterations of Algorithm 3, where t' is the largest number such that $t' < e_A$ and the y-coordinate of $R_0 = [2^{t'}]Q$ is different than 0. This allows us to recover the y-coordinate using Eq. 7. However notice that if we set $t' = e_A - 1$, then R_0 becomes a point of order two, i.e. $y_{R_0} = 0$. For this reason, we chose $t' = e_A - 2$, since then $R_0 = [2^{t'}]Q_A$, and $y_{R_0} \neq 0$. The points corresponding to the last two missing steps of the ladder can be conditionally added together with the point P_A.

Referring to Algorithm 4, in step 1 the scalar k' is computed. Then in steps 2-4 the values of the two most significant bits of k' are saved as $\alpha = k'_{e_A-1}$ and $\beta = k'_{e_A-2}$. Also k' is updated to consider only its t' least significant bits. Then, Algorithm 3 computes $S_A + [k']Q_A$ performing exactly t' iterations. After clearing S_A, the accumulators hold $R_0 = R_1 + R_2 = [2^{e_A-2}]Q_A$ and $y_{R_0} \neq 0$. This allows to recover the y-coordinate of $[3(k' \mod 2^{e_A-2})]Q_A$ using Eq. 7. Thereafter, in steps 9-11 the points $3k'_{e_A-1}2^{e_A-1}Q_A$ and $3k'_{e_A-2}2^{e_A-2}Q_A$ are conditionally added to obtain $[3k']Q_A = [k]Q_A$. Finally the procedure returns $\chi(P_A + [k]Q_A)$.

The conditional point additions of steps 9-11 must be computed in a secure way. One common technique is to conditionally select $U \in E(\mathbb{F}_q)$ from $\{U, \mathcal{O}\}$ according to the bit value (this can be securely implemented using a conditional move or conditional swap). However, there is an issue when the bit chooses $\mathcal{O}$ due to the projective point addition is not complete, i.e. it can not handle the point at infinity. To remedy this situation, we precompute the following points:

$$\begin{aligned} U_0 &= -P_A\,, & U_1 &= U_0 + [3 \times 2^{e_A-1}]Q_A\,, \\ V_0 &= [2]P_A\,, & V_1 &= V_0 + [3 \times 2^{e_A-2}]Q_A. \end{aligned} \tag{8}$$

Steps 9-11 of Algorithm 4 show how to select these points by using the auxiliary function CMOVE, which conditionally moves the points according to the input bit value. Note that regardless the bit values, our procedure always add P_A.

Table 2: Cost of point tripling formulas (in $\mathbb{P}^1$) for a Montgomery elliptic curve with parameter $A = A_0/A_1$.

$A = A_0/A_1$	Cost	Precomputation	Reference
$A_1 = 1$	$7\mathbf{M} + 4\mathbf{S} + 8\mathbf{A}$	$\{(A+2)/4\}$	[30]
	$6\mathbf{M} + 5\mathbf{S} + 9\mathbf{A}$	$\emptyset$	[36]
A_1 arbitrary	$8\mathbf{M} + 4\mathbf{S} + 8\mathbf{A}$	$\{A_0 + 2A_1, 4A_1\}$	[13]
	$7\mathbf{M} + 5\mathbf{S} + 10\mathbf{A}$	$\{A_0 \pm 2A_1\}$	[11]
	$7\mathbf{M} + 5\mathbf{S} + 9\mathbf{A}$	$\{A_0 - 2A_1, 2A_1\}$	**Our work**

The overhead caused by these modifications in Alice's side is negligible in comparison with Bob's method. By using this approach, both Alice and Bob can benefit from the usage of a pre-computation table to accelerate the key generation phase.

4 Optimization of Point Tripling in Montgomery Curves

The calculation of large-degree isogenies requires to compute either $[2^i]\chi(P)$ or $[3^i]\chi(P)$ for some point $P \in E(\mathbb{F}_q)$ and some integer i. These operations are computed repeatedly applying point doubling or tripling algorithms using projective formulas in $\mathbb{P}^1$. For the sake of efficiency, we look for an optimized formula that computes the point tripling operation faster.

A common technique to compute $[3]P$ consists of performing a point doubling followed by a differential point addition, i.e. $[3]P = [2]P +_{(P)} P$. This method has a cost of $7\mathbf{M}$+$4\mathbf{S}$+$8\mathbf{A}$ field arithmetic operations. Recently, Subramanya Rao [36] showed a more efficient formula to compute a point tripling. Given $P = (X_1 \colon Z_1)$ and let A be the Montgomery curve parameter, such a formula calculates $[3]P = (X_3 \colon Z_3)$ as follows:

$$\begin{aligned} \lambda &= \left({X_1}^2 - {Z_1}^2\right)^2 \\ \gamma &= 4({X_1}^2 + {Z_1}^2 + AX_1Z_1) \\ X_3 &= X_1\left(\lambda - \gamma {Z_1}^2\right)^2 \\ Z_3 &= Z_1\left(\lambda - \gamma {X_1}^2\right)^2 . \end{aligned} \tag{9}$$

This formula is derived by coalescing the point doubling and differential addition and its computational cost is $6\mathbf{M}$+$5\mathbf{S}$+$9\mathbf{A}$ field operations.

In the SIDH context, the parameter A of the Montgomery elliptic curve (see Eq. 2) is not fixed, since it may change due to the computation of isogenies. Because of this, the parameter A is represented as a quotient $A = A_0/A_1$. This representation, which was introduced in [13], avoids the usage of inversions for the computation of large-degree isogenies. Therefore, tripling formulas must be modified to operate with A_0 and A_1. Table 2 shows the cost of several tripling formulas reported in the literature.

We optimize the calculation of the tripling formula observing that $2X_1Z_1$ can be calculated from ${X_1}^2$, ${Z_1}^2$, and $(X_1 + Z_1)^2$ as:

$$2X_1Z_1 = (X_1 + Z_1)^2 - ({X_1}^2 + {Z_1}^2)\,; \tag{10}$$

thus, λ from (9) can be also calculated using (10) as follows:

$$\begin{aligned} \lambda &= (X_1 + Z_1)^2(X_1 - Z_1)^2 \\ &= (X_1 + Z_1)^2\left[({X_1}^2 + {Z_1}^2) - 2X_1Z_1\right] ; \end{aligned} \tag{11}$$

likewise, γ from (9) is given as:

$$\begin{aligned} \gamma &= 4\big({X_1}^2 + {Z_1}^2 + AX_1Z_1\big) \\ &= 2\big[2({X_1}^2 + {Z_1}^2 + AX_1Z_1)\big] \\ &= 2\big[2(X_1 + Z_1)^2 + (A-2)(2X_1Z_1)\big]\,. \end{aligned} \tag{12}$$

Using Eq. (10)-(12) and considering that $A = A_0/A_1$, we calculate $[3]P = (X_3 \colon Z_3)$ as follows:

$$\begin{aligned} \lambda &= (2A_1)(X_1 + Z_1)^2\big[({X_1}^2 + {Z_1}^2) - 2X_1Z_1\big] \\ \gamma &= 4\big[(2A_1)(X_1 + Z_1)^2 + (A_0 - 2A_1)(2X_1Z_1)\big] \\ X_3 &= X_1\Big(\lambda - \gamma {Z_1}^2\Big)^2 \\ Z_3 &= Z_1\Big(\lambda - \gamma {X_1}^2\Big)^2\,. \end{aligned} \tag{13}$$

Assuming $A'_0 = A_0 - 2A_1$ and $A'_1 = 2A_1$ are precomputed, then our point tripling formula requires **7M**+5**S**+9**A** using the following sequence of operations:

```
1:  $t_0 \leftarrow (X_1)^2$
2:  $t_1 \leftarrow (Z_1)^2$
3:  $t_2 \leftarrow X_1 + Z_1$
4:  $t_2 \leftarrow (t_2)^2$
5:  $t_3 \leftarrow t_0 + t_1$
6:  $t_4 \leftarrow t_2 - t_3$
7:  $t_5 \leftarrow A'_0 \times t_4$
8:  $t_2 \leftarrow A'_1 \times t_2$
9:  $t_5 \leftarrow t_2 + t_5$
10: $t_5 \leftarrow t_5 + t_5$
11: $t_5 \leftarrow t_5 + t_5$
12: $t_0 \leftarrow t_0 \times t_5$
13: $t_1 \leftarrow t_1 \times t_5$
14: $t_4 \leftarrow t_3 - t_4$
15: $t_2 \leftarrow t_2 \times t_4$
16: $t_0 \leftarrow t_2 - t_0$
17: $t_1 \leftarrow t_2 - t_1$
18: $t_0 \leftarrow (t_0)^2$
19: $t_1 \leftarrow (t_1)^2$
20: $X_3 \leftarrow X_1 \times t_1$
21: $Z_3 \leftarrow Z_1 \times t_0$
```

It can be seen that our formula improves point tripling computation by 1**M**-1**S**-1**A** with respect to the formula used by Costello et al. in [13] (cf. Table 2). Independent work of Costello and Hisil [11] gives formulas for point tripling; however, our formulas are one field addition faster.

Bob's isogeny computations require the frequent computation of point tripling operations to calculate points of the form $[3^i]P$. Therefore, one can see that any improvement in the tripling formula impacts directly the calculation of large-degree isogenies, which are by far the most time-consuming operations in the SIDH protocol.

5 Finite Field Arithmetic Implementation

The instantiation of the SIDH protocol by Costello et al. [13] uses a prime modulus of the form, $p_{\text{CLN}} = 2^{372}3^{239} - 1$. Notice that this prime can be represented using twelve 64-bit words.

Since the SIDH protocol computes isogenies of supersingular elliptic curves defined over the field $\mathbb{F}_{p^2}$, a sensible implementation of the SIDH protocol must implement fast arithmetic in the quadratic field $\mathbb{F}_{p^2}$. Quadratic field arithmetic can be performed more efficiently by means of a field towering approach that relies on an optimized implementation of the base field arithmetic $\mathbb{F}_p$. For example, the multiplication and squaring operations in the quadratic extension field translate to the computation of three and two field multiplications in the base field $\mathbb{F}_p$ as discussed next.

Let $\mathbb{F}_{q=p^2} = \mathbb{F}_p[i]/(i^2+1)$, where $i^2 + 1$ is an irreducible binomial[6] in $\mathbb{F}_p[i]$. The field elements $a, b \in \mathbb{F}_q$ can be written as $a = a_0 + a_1 \cdot i$ and $b = b_0 + b_1 \cdot i$. Using a Karatsuba approach the field multiplication $c = a \cdot b = c_0 + c_1 \cdot i$ can be computed as,

$$\begin{aligned} c_0 &= a_0 \cdot b_0 - a_1 \cdot b_1, \\ c_1 &= (a_0 + a_1)(b_0 + b_1) - a_0 \cdot b_0 - a_1 \cdot b_1, \end{aligned} \tag{14}$$

[6] Always true whenever $p \bmod 4 = 3$.

which can be performed at a cost of three integer multiplications, five integer additions and two modular reductions. Similarly the field squaring operation, for $a \in \mathbb{F}_q$, is computed as $a^2 = (a_0 + a_1 \cdot i)^2 = (a_0 + a_1) \cdot (a_0 - a_1) + 2a_0a_1 \cdot i$ at a cost of two integer multiplications, two modular reductions and three integer additions.

In the following, several programming and algorithmic techniques that lead to a fast implementation of the field multiplication and squaring operations in the base field $\mathbb{F}_p$ will be discussed. Our software library relies heavily on novel instruction sets recently introduced in modern Intel and AMD processors, which have been especially designed for achieving a faster execution of multi-precision integer arithmetic. For the sake of concreteness our description will be mainly focused on the popular prime modulus p_{CLN} striving to exploit its very special form.

5.1 Features of x86_64 Arithmetic Instructions

In order to reduce the latency of the field arithmetic operations, we took advantage of the `ADCX/ADOX` and `MULX` instructions available in the newest Intel and AMD micro-architectures. These instructions were designed for speeding-up integer multi-precision arithmetic operations.

Starting from the Intel Haswell micro-architecture, the instruction `MULX` was introduced as a part of the Bit Manipulation Instruction set (BMI2). `MULX` is an extension of the traditional 64-bit multiplication instruction `MUL`. More specifically, `MULX` computes the multiplication of two unsigned 64-bit operands without affecting the arithmetic flags. Additionally, `MULX` uses a three-operand code that allows the programmer to choose the registers that will be used for storing the upper and lower part of the output product. This feature permits to preserve the data stored in the input registers. Further, `MULX` can be combined with specialized addition instructions that do not affect the carry chain state.

On the other hand the set of instructions ADX (addition instructions with independent carry chains) [18], which was first supported in the Intel Broadwell micro-architecture, includes the instructions `ADCX` and `ADOX`. These two instructions are extensions of the traditional 64-bit addition instructions `ADD`/`ADC`, and were designed for handling two independent carry chains. The new instructions compute unsigned 64-bit integer additions with an input carry, and generate an output carry without modifying the carry flag (CF) and the overflow flag (OF), respectively. These features permit that `ADCX` and `ADOX` can be executed concurrently.

The combined usage of these novel instructions, it allows a more efficient implementation of the field arithmetic operations that will be studied in the remaining of this section.

5.2 Base Field Multiplication and Squaring

Field multiplication (or squaring) over $\mathbb{F}_p$ is performed by first computing an integer multiplication (or an integer squaring) followed by a modular reduction. Our integer arithmetic implementation is described next.

5.2.1 Integer Multiplication

Integer multiplication was performed using a combination of the Karatsuba and the schoolbook multiplication methods. After experimenting with different settings we arrived to an optimal combination that consists of implementing one Karatsuba level recursion followed by the application of the schoolbook method for the lower product computations.

Let $a, b \in \mathbb{F}_p$ be written as, $a = a_L + a_H \cdot x$ and $b = b_L + b_H \cdot x$, such that $x = r^{n/2}$, $r = 2^{64}$, and $n = 12$. Using once again a Karatsuba approach (this time at the base field level), let us define c_L, c_M, and c_H as,

$$c_L = a_L \cdot b_L \, , \; c_M = (a_L + a_H) \cdot (b_L + b_H) \, , \; c_H = a_H \cdot b_H .$$

It follows that the integer product $c = a \cdot b$ can be computed as,

$$c = c_L + (c_M - c_L - c_H) \cdot x + c_H \cdot x^2 \, .$$

In spite of its quadratic complexity with respect to the word size of the operands, for small values of n the schoolbook method tends to outperform a Karatsuba-based product computation. Hence, the multiplications involving six-word operands required for obtaining the auxiliary operands c_H, c_L, and c_M were performed using the schoolbook method.

The efficiency of the schoolbook method crucially depends on the selection of the partial products and the way that they are added. Integer multiplication can be performed using a product-scanning multiplication strategy that computes all the partial products column-wise. Alternatively, one can also process the operands row-wise. In this case the multiplicand operand is multiplied by each word of the multiplier. After performing all word multiplications, the partial products are properly shifted and added to obtain the output product. For Intel architectures this latter approach appears to be more advantageous because the carry propagation performed by means of the CF flag provides a more efficient carry management. This approach also permits to take full advantage of the `MULX` instruction, since it does not corrupt the carry flag when combined with the addition instructions[7]. Finally, this strategy profits from the usage of the `ADCX/ADOX` instructions, which can handle two independent carry chains concurrently. Hence, integer multiplication $a \cdot b$ for up to eight-word operands a and b was performed using the operand scanning implementation of the schoolbook multiplication, at a computational cost of n^2 and $(n-1)^2$ 64-bit word multiplications and additions, respectively.

We only used a first level of Karatsuba because according to our experiments, the schoolbook method outperforms the Karatsuba approach for operands that can be accommodated with eight 64-bit words or less. For example, an eight-word Karatsuba multiplication performs 48 word multiplications, for a total execution time of 147 clock cycles, whereas the schoolbook method requires 64 word multiplications at an overall runtime of 141 clock cycles. For operands requiring nine words or more, our experiments show that the Karatsuba approach yields a better time performance than the one associated to the schoolbook method.

5.2.2 Integer Squaring

Field squaring was performed using a combination of the Karatsuba and the schoolbook methods as follows. Let us represent an arbitrary element $a \in \mathbb{F}_p$ as $a = a_L + a_H \cdot x$, such that $x = r^{n/2}$, $r = 2^{64}$, and $n = 12$. Then the integer squaring operation $c = a^2$ can be computed as,

$$c = c_L + 2 \cdot (a_L \cdot a_H) \cdot x + c_H \cdot x^2,$$

where $c_L = {a_L}^2$ and $c_H = {a_H}^2$. Integer squaring was computed using two Karatusba recursion levels, which are followed by a schoolbook computation of integer squaring and multiplication operations performed on three-word operands, respectively. The reason why in this case we used one more Karatsuba level is mainly because by unrolling the main loop of the schoolbook algorithm, we managed to optimize the required addition computations. This way, the addition computations were optimized by performing a batch computation of as many of them as possible. From our experiments, we observed that for a six-word operand a, the Karatsuba and the schoolbook squaring methods require 73 and 102 clock cycles, respectively. In the case of a four-word operand, the schoolbook method requires 34 clock cycles versus 41 clock cycles required by the Karatsuba method.

[7] We stress that this observation applies mainly for architectures supporting the ADX instruction set, such as the Intel Skylake micro-architecture.

5.2.3 Modular Reduction

Montgomery introduced in [29] an algorithm that performs a modular multiplication $a \cdot b \bmod p$, without computing expensive divisions by p. Montgomery multiplication projects the operands $a, b \in \mathbb{Z}_p$ to the integers $\tilde{a}, \tilde{b} \in \mathbb{Z}_p$ using the mappings, $\tilde{a} = a \cdot R \bmod p$ and $\tilde{b} = b \cdot R \bmod p$. The parameter R is usually chosen as a power of two, which is co-prime to the odd modulus p. The integers $\tilde{a}, \tilde{b}$ are said to be in the Montgomery domain. Montgomery reduction [29], named by its author as the REDC algorithm, is generally considered the most efficient approach for performing multi-precision modular arithmetic.

Once again, let n denote the number of words or digits used to represent integers in radix-2^w, where w is the word size of the targeted architecture. The REDC algorithm can be implemented computing exactly $n^2 + n$ word multiplications [29, p. 2]. Algorithm 5 presents a multi-precision version of the REDC algorithm. The input parameter T of Algorithm 5 is an $2n$-digit register that holds the result of performing the integer multiplication $T = a \cdot b$, where $a, b \in \mathbb{Z}_p$ are n-digit numbers. A constant-time execution of Algorithm 5 can be achieved by omitting the conditional subtraction of steps 7-9 using the techniques introduced by Walter in [37].

Algorithm 5 Multi-precision REDC algorithm.

Input: T, an integer such that $0 \leq T < Rp$, $R = 2^{wn}$, and a constant $p' = -p^{-1} \mod 2^w$.
Output: C, an integer such that $C = TR^{-1} \mod p$.

```
 1: for $i \leftarrow 1$ to $n$ do
 2:     $t \leftarrow T \mod 2^w$
 3:     $q \leftarrow tp' \mod 2^w$
 4:     $T \leftarrow (T + q \cdot p)/2^w$
 5: end for
 6: $C \leftarrow T$
 7: if $C \geq p$ then
 8:     $C \leftarrow C - p$
 9: end if
10: return $C$
```

Several authors [1, 20, 23, 27] have exploited special classes of moduli, which are especially amenable for reducing the number of word multiplications associated to the REDC Algorithm. These moduli, which are sometimes named Montgomery-friendly, have the following property. We say that a modulus p is λ-*Montgomery-friendly* if $p \equiv \pm 1 \mod 2^{\lambda \cdot w}$ for positive integers λ and w.[8] This property implies that $-p^{-1} \equiv \mp 1 \mod 2^{\lambda \cdot w}$, which saves the multiplication computation of step 3 in Algorithm 5. This saving immediately implies that the REDC algorithm can be computed using only n^2 digit multiplications. However, we can improve the performance of the REDC algorithm further as discussed next.

Exploiting the Special Form of the SIDH Moduli

The main algorithmic idea of the REDC multi-precision version shown in Algorithm 5 is that of calculating a quotient q that makes $T + q \cdot p$ divisible by 2^w. This allows to update T as $(T + q \cdot p)/2^w$, which implies that at each iteration of Algorithm 5, the size of T is decreased by one word. Notice that the value of q in step 3 directly depends on the updated value of T. This situation is commonly known as a loop-carried dependency that prevents a further parallelization of Algorithm 5. Therefore, this procedure can only process one $q \cdot p$ product per iteration with an associated cost of one $1 \times n$ digit multiplication.

[8] Notice that the prime p_{CLN} is a 5-Montgomery-friendly modulus setting $w = 64$.

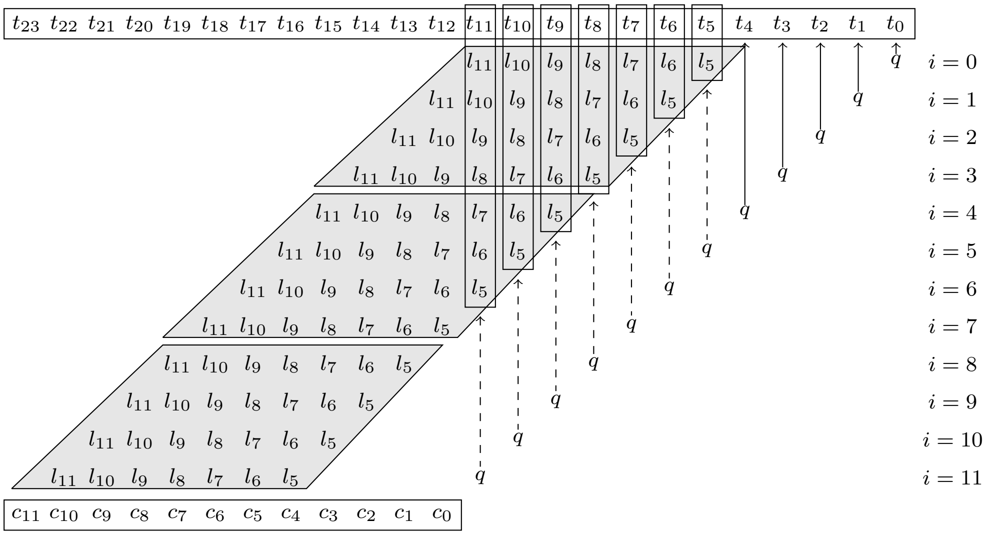

t_{23} t_{22} t_{21} t_{20} t_{19} t_{18} t_{17} t_{16} t_{15} t_{14} t_{13} t_{12} t_{11} t_{10} t_9 t_8 t_7 t_6 t_5 t_4 t_3 t_2 t_1 t_0
l_{11} l_{10} l_9 l_8 l_7 l_6 l_5
q
$i = 0$
$i = 1$
$i = 2$
$i = 3$
$i = 4$
$i = 5$
$i = 6$
$i = 7$
$i = 8$
$i = 9$
$i = 10$
$i = 11$
c_{11} c_{10} c_9 c_8 c_7 c_6 c_5 c_4 c_3 c_2 c_1 c_0

Figure 2: Multi-precision execution of $C = \text{REDC}(T)$ for $n = 12$. Given the input $T = (t_0, \ldots, t_{23})$, REDC calculates n times the product $q \cdot (p + 1)$, where p is a 5-Montgomery-friendly prime. This implies that $p + 1$ can be expressed as $(p_{11}, p_{10}, p_9, p_8, p_7, p_6, p_5, 0, 0, 0, 0, 0)$. In order to update T, at each iteration the partial products $l_j = p_j q$, for $5 \leq j < 12$ are computed. The dependency for calculating q at each iteration is highlighted with arrows. Notice that the first five values of q only depends on the unmodified value of T (this fact is represented by solid arrows).

Nevertheless, when Algorithm 5 is executed using a λ-Montgomery-friendly modulus the loop-carried dependency can be avoided in up to λ iterations of the main loop. To see how this trick works notice that in step 2 the value t is assigned with the least significant word of T. If p is a λ-Montgomery-friendly modulus and $p' = 1$. This implies that in step 3 there is no multiplication to be performed but a simple assignment $q = t$. It follows that step 4 can be computed as,

$$\frac{(T + q \cdot p)}{2^w} = \frac{(T + t \cdot p)}{2^w} = \frac{(T + t \cdot (p - 1) - t)}{2^w}.$$

Since $p+1$ can be represented as $(p_{n-1}, \ldots, p_0)$, where $p_i = 0$ for $0 \leq i < \lambda$, the λ least-significant words of the product $t \cdot (p + 1)$ are all equal to zero. This implies that we can compute step 4 of Algorithm 5 by multiplying t with the $n - \lambda$ most-significant words of $p + 1$, adding the resulting product with T, and completely ignoring the least-significant word of this computation. In other words,

$$\frac{(T + q \cdot p)}{2^w} = \frac{(T + t \cdot (p + 1) - t)}{2^w} = \left\lfloor \frac{T + t \cdot (p + 1)}{2^w} \right\rfloor.$$

We observe that since the least-significant words of T are not modified, then the value of q for the next iteration can be obtained in advance, thus breaking the loop-carried dependency. In general, for a λ-Montgomery-friendly prime one can calculate the value of q for λ iterations without the knowledge of the values that T will be getting in those iterations.

We illustrate in Figure 2 the execution of the multi-precision REDC algorithm using as a modulus $p = p_{\text{CLN}}$, which is a 5-Montgomery-friendly modulus that has a size of $n = 12$ words. At the i-th iteration of the REDC algorithm an updated value of q is calculated and multiplied by $p+1$. Then, the result is added to T (on top), and the least-significant word of T is removed. After n iterations, the final result is stored in C, which is composed of the twelve most-significant words of T. The vertical arrows denote the dependencies associated to the computation of q. Since p is a 5-Montgomery-friendly prime, from the first to the fifth iteration q only depends on the original value of T. However in the sixth iteration q depends on T and on the value of $q \cdot (p+1)$ from the first iteration (this fact is highlighted by the dashed arrows and the vertical rectangles). As can be seen, no loop-carried dependencies appear during the first five iterations, allowing to compute up to five $q \cdot (p+1)$ products before updating T becomes necessary. These products can be viewed as a 7×4 digit multiplication followed by a 64-bit left-shift in the case of p_{CLN} modulus (these operations are highlighted in the shadowed area). Thus, after performing three 7×4 digit multiplications and three 64-bit left-shifts the modular reduction is completed. This latter observation inspired us to come out with the modified version of Algorithm 5 shown in Algorithm 6.

Algorithm 6 Modified modular reduction algorithm for a λ-Montgomery-friendly modulus.

Input: T, an integer such that $0 \leq T < Rp$, $R = 2^{wn}$, p is a λ-Montgomery-friendly modulus, and $0 < B \leq \lambda$.

Output: C, an integer such that $C = TR^{-1} \mod p$.

```
 1: $\lambda_0 \leftarrow \lfloor n/B \rfloor$
 2: $\lambda'_0 \leftarrow n \mod B$
 3: $M \leftarrow \lfloor (p+1)/2^{\lambda \cdot w} \rfloor$
 4: for $i \leftarrow 1$ to $\lambda_0$ do
 5:     $Q \leftarrow T \mod 2^{B \cdot w}$
 6:     $T \leftarrow \lfloor T/2^{B \cdot w} \rfloor + 2^{(\lambda - B) \cdot w} Q \cdot M$
 7: end for
 8: if $\lambda'_0 \neq 0$ then
 9:     $Q \leftarrow T \mod 2^{\lambda'_0 \cdot w}$
10:     $T \leftarrow \lfloor T/2^{\lambda'_0 \cdot w} \rfloor + 2^{(\lambda - \lambda'_0) \cdot w} Q \cdot M$
11: end if
12: $C \leftarrow T$
13: if $C \geq p$ then
14:     $C \leftarrow C - p$
15: end if
16: return $C$
```

The modified REDC procedure is presented in Algorithm 6. Given a λ-Montgomery-friendly modulus p, the number of iterations without loop-carried dependency can be chosen as, $0 < B \leq \lambda$ (this value B is equals four for the example in Figure 2). In step 5 of Algorithm 6 the value $Q = T \mod 2^{B \cdot w}$ is computed. Thereafter Q is multiplied by the $\lambda_1 = n - \lambda$ most-significant words of $p+1$ given by $M = \lfloor (p+1)/2^{\lambda \cdot w} \rfloor$. It is noticed that Q is a B-digit number. Hence, $Q \cdot M$ can be calculated as a $B \times \lambda_1$ digit product. At this point, the value $2^{(\lambda - B) \cdot w} Q \cdot M$ is added to $\lfloor T/2^{B \cdot w} \rfloor$. Hence, the B least-significant digits of T are discarded. Repeating this procedure $\lambda_0 = \lfloor \frac{n}{B} \rfloor$ times, the size of T is decreased by $B\lambda_0$ words. In the case that $n \mod B = 0$, the modular reduction has been completed. Otherwise, $\lambda'_0 = n \mod B$ digits of T must still be reduced by applying one extra iteration using $B = \lambda'_0$ (lines 8-11 of Algorithm 6).

Algorithm 6 shares similar ideas as the ones presented by Bos and Friedberger in [5]. In particular, the strategy named *shifted* (SH) in [5] that allows to trade multiplications by right-shift operations can be easily adapted to the setting described here and shown in Algorithm 6.

Correctness

From the previous discussion, it follows that the first B iterations of the multi-precision REDC Algorithm 5 do not show loop-carried dependencies. Thus, the first B values of q can be computed at once,

Table 3: Performance comparison of different modular reduction algorithms. For Algorithm 6, the admissible values of B for the prime $p_{\text{CLN}} = 2^{372}3^{239} - 1$ were measured. The timings are reported in clock cycles measured on a Skylake micro-architecture. SH stands for the *shifted* technique as proposed in [5].

Ref.	B	Instr. Set	Operation Counts				Clock Cycles
			Mul	Add	Mov	Other	
This work	1	mul/adc	84	251	204	8	281
		mulx/adc	84	191	24	8	232
		mulx/adx	84	191	24	8	230
	2	mul/adc	84	289	207	10	244
		mulx/adc	84	257	27	10	208
		mulx/adx	84	149	27	16	187
	3	mul/adc	84	301	210	10	227
		mulx/adc	84	281	34	10	210
		mulx/adx	84	137	34	18	193
	4	mul/adc	84	307	210	10	218
		mulx/adc	84	292	36	10	191
		mulx/adx	84	130	42	17	162
	4 + (SH)	mul/adc	72	265	186	46	204
		mulx/adc	72	253	36	46	189
		mulx/adx	**72**	**118**	**36**	**55**	**156**
[5]	1	mul/adc	84	332	157	41	254
	2	mul/adc	84	358	202	61	275
	1+(SH)	mul/adc	72	299	223	86	240

by setting $Q = T \mod 2^{B\cdot w}$ (line 5 of Algorithm 6). Then T is updated with $\lfloor T/2^{\lambda_0'\cdot w}\rfloor + 2^{(\lambda-\lambda_0')\cdot w}Q\cdot M$. In this way the B least-significant words of T are removed. After λ_0 iterations, the size of T will be reduced $B\lambda_0$ words exactly as it would happen after $n - \lambda_0'$ iterations of a regular execution of the multi-precision REDC algorithm. Whenever $B \nmid n$, one additional iteration is processed to reduce the remaining λ_0' digits. Since Algorithm 6 performs the same reduction that Algorithm 5 computes, the final conditional subtraction step of lines 13-15 is also required.

Case Study $p_{\mathbf{CLN}} = 2^{372}3^{239} - 1$

Here we focus our attention to the problem of fine-tuning the design parameters of Algorithm 6 when dealing with the modulus p_{CLN}.

We performed several experiments with different values of the parameter B trying to determine the optimal value of this parameter that yields the modular reduction with the smallest latency. In order to provide a fair comparison, we performed the implementation of this operation using three different variants, which mainly differ in the type of x86_64 arithmetic instructions that were used (cf. §5.1). The benchmarked timings obtained from our experiments are reported in Table 3. It can be seen that the best results were obtained using the combination of `MULX` and `ADCX`/`ADOX` instructions and setting $B = 4$, along with the shifted technique, this latter techinque was proposed in [5]. Using this design choice, the modular reduction has a cost of three 6×4 64-bit multiplications, three 52-bit right-shifts over 10-word operands, and three additions over 11-word operands. The measured latency is of 156 clock cycles.

Our fastest modular reduction timing (reported in Table 3) is more efficient by a factor 1.6 than the one achieved in [5] with $B = 1$, which corresponds to the modular reduction based in the product scanning multiplication as presented in Costello et al. [13]. Moreover, we obtained a modular reduction that is faster by a factor 1.5 than the one reported in [5] with $B = 1$ and the shifted technique (SH). This latter result somewhat contradicts the conjecture that a value $B > 1$, may lead to a lower performance than the one associated with the choice $B = 1$, adopted by Bos and Friedberger in [5].

6 Implementation and Benchmark Results

We benchmarked our software on an Intel Core i7-4770 processor supporting the Haswell micro-architecture and on an Intel Core i7-6700K processor that supports the Skylake micro-architecture. To guarantee the reproducibility of our measurements, the Intel Hyper-Threading and Intel Turbo Boost technologies were disabled. Our source code was compiled using the GNU C Compiler (gcc) v6.1.0 with the `-O3` optimization flag and using the options `-mbmi2 -fwrapv -fomit-frame-pointer` and `-mbmi2 -madx -fwrapv -fomit-frame-pointer` for the Haswell and Skylake micro-architectures, respectively. Our code is available at: [`http://github.com/armfazh/flor-sidh-x64`].

6.1 Related Works

Due to the novelty of the SIDH protocol only a few software and hardware implementations have so far been reported. Several of these implementations use different elliptic curve parameters, which makes it difficult to come out with a fair comparison. The publicly-available implementation of Costello et al. [13] is a portable software library called SIDH v2.0. This library includes optimized 64-bit code for field arithmetic, public key compression algorithms and an instantiation of the Diffie-Hellman protocol. SIDH v2.0 is widely considered the state-of-the-art software library for implementing the SIDH protocol. Other SIDH publicly available software libraries include [2, 16]. In [16], De Feo reports an implementation of the SIDH protocol supporting several prime sizes [16]. His implementation relies on the GMP library [19] as a modular arithmetic back-end. The implementation by Azarderakhsh et al. [2] is also publicly available. However, the performance of this library is significant slower than the library presented in [13].

In this work, we rely on the software library of Costello et al. [13], since it is the fastest one reported in the open literature. Further, in order to report a more complete picture of the SIDH protocol acceleration provided by the techniques presented in this paper, we plugged-in our elliptic curve and field arithmetic functions in that library.

6.2 Prime Field Arithmetic

In Table 4, the running time of relevant prime field and elliptic curve operations for the Haswell and Skylake micro-architectures are reported.

Comparing with the implementation of Costello et al., the multiplication in the quadratic extension field $\mathbb{F}_{p^2}$, which is a performance-critical operation, was consistently accelerated by a factor 1.32-1.34 speedup in both platforms. This improvement produces an immediate acceleration of all elliptic curve operations, yielding a factor 1.13-1.25 speedup in the Haswell micro-architecture. For Skylake, the impact of our implementation is higher, since our library benefits from more specialized multi-precision arithmetic instructions. In Skylake, the elliptic curve operations achieved a factor 1.14-1.32 speedup.

6.3 Impact of the $P + [k]Q$ optimization

We measured the performance rendered by the ladder algorithms presented in §3. To that end, we take as a baseline the original SIDH v2 library and plugged in our algorithms using the same prime field arithmetic interface.

Table 4: Timing performance of selected base field, quadratic and elliptic-curve arithmetic operations. The last column shows the acceleration factor that our library obtained in comparison with the SIDH v2 library [13]. All timings are reported in clock cycles measured in the Haswell and Skylake micro-architectures.

Domain	Operation	Haswell			Skylake		
		CLN [13]	**Our work**	AF	CLN [13]	**Our work**	AF
$\mathbb{F}_p$	Modular reduction	279	242	1.15	212	156	1.36
	Multiplication	670	605	1.11	486	415	1.17
	Squaring	724	526	1.38	523	395	1.32
	Inversion	622,761	462,099	1.35	456,621	354,373	1.29
$\mathbb{F}_{p^2}$	Multiplication	2,143	1,626	1.32	1,582	1,183	1.34
	Squaring	1,420	1,256	1.13	1,026	880	1.16
	Inversion	625,904	463,773	1.35	458,706	355,889	1.29
$E(\mathbb{F}_{p^2})$	Dif. Addition	10,160	8,316	1.22	7,371	5,896	1.25
	Point Doubling	12,019	9,619	1.25	8,855	6,969	1.27
	Point Tripling	24,024	19,247	1.25	17,799	13,528	1.32
	Ladder Step ($\mathbb{F}_{p^2}$)	19,715	16,123	1.22	14,384	11,802	1.22
	Ladder Step ($\mathbb{F}_p$)	7,403	6,085	1.22	5,259	4,327	1.21
	Iso. Gen. 3-degree	11,678	9,737	1.19	8,537	6,873	1.24
	Iso. Gen. 4-degree	8,174	7,252	1.13	5,980	5,241	1.14
	Iso. Eval. 3-degree	15,817	12,842	1.23	11,864	9,369	1.27
	Iso. Eval. 4-degree	21,480	17,154	1.25	15,932	12,377	1.29

The benchmarked timings are summarized in Table 5. In all the cases, we were able to corroborate the theoretical predictions summarized in Table 1. For example for the variable-point scenario, the SIDH v2 library computes the three-point ladder in 11.2×10^6 Haswell clock cycles. Our software accelerates this timing by a factor 1.38 speedup to compute the same operation. Thank to this, Alice and Bob shared-secret time performance are accelerated by around 6-7% (cf. Table 6). In the case of the fixed-point scenario it can be observed that using either Algorithm 3 or Algorithm 4 our approach is ≈ 1.7 faster than the methods implemented in the SIDH v2 library. Once again these results confirm the theoretical estimates given in Table 1. The pre-computed look-up tables have a size of around 35 KB. This relatively moderate size permits that a large part of the look-up tables can fit in the Level-1 Data cache memory of the target platforms (which have a size of 32 KB).

Regarding side-channel protection, we want to note that the right-to-left algorithms were implemented considering classic countermeasures; for example, using a straight and a regular execution of instructions. Moreover, no secret values were used to index look-up tables or to bifurcate the execution of any function.

6.4 Point Tripling Impact

Clearly, the most time consuming SIDH operation is the calculation of large-degree isogenies. In the case of Bob, this process implies to perform a large number of point tripling computations.

Our implementation of the point tripling formula proposed in §4 saves up to 400 clock cycles, corresponding to the difference 1**M**-1**S**-1**A** (cf. Table 4). This reduction in the cost of the point tripling computation yields a small but noticeable acceleration of the whole protocol. More concretely, replacing the tripling formula implemented in the SIDH v2 library by our proposed formula yields a speedup of around 1-2% in the SIDH protocol execution.

Table 5: Performance comparison of different methods to compute $\chi(P+[k]Q)$. The implementation of Methods 1, 2 and 3 were taken from the SIDH-v2 library [13]. All timings are given in 10^6 clock cycles and were measured on a Haswell and on a Skylake micro-architecture.

Scenario	Field	Haswell	Skylake	Algorithm
Fixed-point	$\mathbb{F}_{p^2}$	6.7	4.9	Method 1
		3.9	2.9	**Alg. 3 (this work)**
	$\mathbb{F}_p$	2.5	1.7	Method 3
		1.5	1.0	**Alg. 4 (this work)**
Variable-point	$\mathbb{F}_{p^2}$	11.2	8.1	3-point ladder
		8.0	5.9	**Alg. 2 (this work)**

6.5 Performance Comparison of the SIDH Protocol

In Table 6, the running timings associated with the execution of both phases of the SIDH protocol are reported. It is noted that the achieved speedups are highly correlated with the ones obtained for the multiplication operation in the quadratic extension field $\mathbb{F}_{p^2}$. This confirms the high-impact of this operation in the performance of the whole protocol. For all of the SIDH operations, the performance measured on Skylake was between 1.38 to 1.41 times faster than the one measured on the Haswell processor (cf. Table 6). This acceleration can be seen as a consequence of the higher performance achieved by the latest integer arithmetic instruction sets (which are available in Skylake but not in Haswell).

Table 6: Performance comparison of the SIDH protocol. The running time is reported in 10^6 clock cycles to compute the two phases of the SIDH protocol. Additionally, the speedup factor with respect to the SIDH v2 library [13] is also reported.

Protocol Phase		Haswell			Skylake		
		CLN [13]	**This work**	AF	CLN [13]	**This work**	AF
Key Generation	Alice	48.3	38.0	1.27	35.7	26.9	1.33
	Bob	54.5	42.8	1.27	39.9	30.5	1.31
Shared Secret	Alice	45.7	34.3	1.33	33.6	24.9	1.35
	Bob	52.8	39.6	1.33	38.4	28.6	1.34

7 Conclusions

In this work we presented a number of optimizations targeting the supersingular isogeny-based Diffie-Hellman protocol. We focused our attention on optimizing both the finite field and the elliptic curve arithmetic layers.

We accelerated operations in the base field $\mathbb{F}_p$ and in its quadratic extension $\mathbb{F}_{p^2}$, using the newest arithmetic instruction sets available in modern Intel processors and also, by taking advantage of the special form of the p_{CLN} prime chosen in [13]. The combination of these techniques allowed us to compute finite field arithmetic about 1.38 faster than the performance obtained by running the library of [13] on the same Intel processor architectures.

Building on [33], we adapted a right-to-left Montgomery ladder variant to the context of the SIDH protocol, where the elliptic curve operation $P + [k]Q$ must be computed. In the case when the involved points are known in advance, our algorithm enables for the first time the usage of precomputed look-up tables to accelerate the SIDH key generation phase. We also presented an improved formula for elliptic curve point tripling. Our formula permits to save one multiplication at the cost of one extra squaring and one extra addition performed in the quadratic extension $\mathbb{F}_{p^2}$.

Executing our software on an Intel Skylake Core i7-6700 processor we are able to compute the two phases of the SIDH protocol, namely, key generation and shared secret, in less than 51.8 and 59.1 millions of clock cycles for Alice's and Bob's computations, respectively. This gives us a 1.33 times speedup against the software implementation of Costello et al.

As a final remark, we stress that any source code based on the library SIDH v2 [13], can also be benefited by our optimizations. For instance, the isogeny-based signature scheme recently presented in [39].

References

[1] T. Acar and D. Shumow, "Modular Reduction without Pre-computation for Special Moduli," Microsoft Research, Tech. Rep., 2010.

[2] R. Azarderakhsh, D. Fishbein, and D. Jao, "Efficient Implementation of a Quantum-Resistant Key-Exchange Protocol on Embedded Systems," Center of Applied Cryptographic Research (CACR), Tech. Rep. CACR 2014-20, 2014.

[3] D. J. Bernstein, "Differential addition chains," Feb. 2006. [Online]. Available: https://cr.yp.to/ecdh/diffchain-20060219.pdf

[4] D. J. Bernstein and T. Lange, "Montgomery curves and the Montgomery ladder," in *Topics in Computational Number Theory Inspired by Peter L. Montgomery*, ser. London Mathematical Society Lecture Note. Cambridge University Press, Oct. 2017, ch. 4, pp. 82–115.

[5] J. W. Bos and S. Friedberger, "Fast Arithmetic Modulo $2^x p^y - 1$," in *2017 IEEE 24th Symposium on Computer Arithmetic (ARITH)*, July 2017, pp. 148–155. [Online]. Available: http://doi.org/10.1109/ARITH.2017.15

[6] C. K. Caldwell, "The Prime Glossary," Dec. 2016. [Online]. Available: http://primes.utm.edu/glossary/xpage/PierpontPrime.html

[7] D. Charles, E. Goren, and K. Lauter, "Cryptographic hash functions from expander graphs," Cryptology ePrint Archive, Report 2006/021, 2006. [Online]. Available: http://eprint.iacr.org/2006/021

[8] D. X. Charles, K. E. Lauter, and E. Z. Goren, "Cryptographic Hash Functions from Expander Graphs," *Journal of Cryptology*, vol. 22, no. 1, pp. 93–113, Jan 2009. [Online]. Available: https://doi.org/10.1007/s00145-007-9002-x

[9] A. Childs, D. Jao, and V. Soukharev, "Constructing elliptic curve isogenies in quantum subexponential time," *Journal of Mathematical Cryptology*, vol. 8, no. 1, pp. 1–29, Feb. 2014. [Online]. Available: https://doi.org/10.1515/jmc-2012-0016

[10] A. M. Childs, D. Jao, and V. Soukharev, "Constructing elliptic curve isogenies in quantum subexponential time," 2010. [Online]. Available: http://arxiv.org/abs/1012.4019

[11] C. Costello and H. Hisil, "A Simple and Compact Algorithm for SIDH with Arbitrary Degree Isogenies," in *Advances in Cryptology – ASIACRYPT 2017: 23nd International Conference on the Theory and Application of Cryptology and Information Security, Hong Kong, China, December 3-7, 2017, Proceedings*, Dec. 2017.

[12] C. Costello, D. Jao, P. Longa, M. Naehrig, J. Renes, and D. Urbanik, "Efficient Compression of SIDH Public Keys," in *Advances in Cryptology – EUROCRYPT 2017: 36th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Paris, France, April 30 – May 4, 2017, Proceedings, Part I*, J.-S. Coron and J. B. Nielsen, Eds. Cham: Springer International Publishing, 2017, pp. 679–706. [Online]. Available: https://doi.org/10.1007/978-3-319-56620-7_24

[13] C. Costello, P. Longa, and M. Naehrig, "Efficient Algorithms for Supersingular Isogeny Diffie-Hellman," in *Advances in Cryptology – CRYPTO 2016: 36th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2016, Proceedings, Part I*, M. Robshaw and J. Katz, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 572–601. [Online]. Available: https://doi.org/10.1007/978-3-662-53018-4_21

[14] C. Costello and B. Smith, "Montgomery curves and their arithmetic," *Journal of Cryptographic Engineering*, pp. 1–14, 2017. [Online]. Available: http://dx.doi.org/10.1007/s13389-017-0157-6

[15] J.-M. Couveignes, "Hard Homogeneous Spaces," Cryptology ePrint Archive, Report 2006/291, 2006, http://eprint.iacr.org/2006/291.

[16] L. De Feo, "Software for "Quantum-Resistant Cryptosystems from Supersingular Elliptic Curve Isogenies"," 2011, http://github.com/defeo/ss-isogeny-software.

[17] L. De Feo, D. Jao, and J. Plût, "Towards Quantum-Resistant Cryptosystems from Supersingular Elliptic Curve Isogenies," *Journal of Mathematical Cryptology*, vol. 8, no. 3, pp. 209–247, Sep. 2014. [Online]. Available: http://doi.org/10.1515/jmc-2012-0015

[18] V. Gopal, J. Guilford, G. Wolrich, W. Feghali, E. Ozturk, M. Dixon, S. Mirkes, M. Merten, T. Li, and T. Bret, "Addition instructions with independent carry chains," Jan. 9 2014, uS Patent App. 13/993,483. [Online]. Available: https://www.google.com/patents/US20140013086

[19] T. Granlund and the GMP development team, *GNU MP: The GNU Multiple Precision Arithmetic Library*, 6th ed., 2016, http://gmplib.org/.

[20] S. Gueron and V. Krasnov, "Fast prime field elliptic-curve cryptography with 256-bit primes," *Journal of Cryptographic Engineering*, pp. 1–11, 2014. [Online]. Available: http://dx.doi.org/10.1007/s13389-014-0090-x

[21] D. Jao and L. De Feo, "Towards Quantum-Resistant Cryptosystems from Supersingular Elliptic Curve Isogenies," in *Post-Quantum Cryptography: 4th International Workshop, PQCrypto 2011, Taipei, Taiwan, November 29 – December 2, 2011. Proceedings*, B.-Y. Yang, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 19–34. [Online]. Available: https://doi.org/10.1007/978-3-642-25405-5_2

[22] M. Joye, "Highly Regular Right-to-Left Algorithms for Scalar Multiplication," in *Cryptographic Hardware and Embedded Systems - CHES 2007: 9th International Workshop, Vienna, Austria, September 10-13, 2007. Proceedings*, P. Paillier and I. Verbauwhede, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 135–147. [Online]. Available: https://doi.org/10.1007/978-3-540-74735-2_10

[23] M. Knežević, F. Vercauteren, and I. Verbauwhede, "Speeding Up Bipartite Modular Multiplication," in *Arithmetic of Finite Fields: Third International Workshop, WAIFI 2010, Istanbul, Turkey, June 27-30, 2010. Proceedings*, M. A. Hasan and T. Helleseth, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 166–179. [Online]. Available: https://doi.org/10.1007/978-3-642-13797-6_12

[24] B. Koziel, R. Azarderakhsh, M. M. Kermani, and D. Jao, "Post-Quantum Cryptography on FPGA Based on Isogenies on Elliptic Curves," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 64, no. 1, pp. 86–99, Jan 2017. [Online]. Available: http://doi.org/10.1109/TCSI.2016.2611561

[25] B. Koziel, R. Azarderakhsh, and M. Mozaffari-Kermani, "Fast Hardware Architectures for Supersingular Isogeny Diffie-Hellman Key Exchange on FPGA," in *Progress in Cryptology – INDOCRYPT 2016: 17th International Conference on Cryptology in India, Kolkata, India, December 11-14, 2016, Proceedings*, O. Dunkelman and S. K. Sanadhya, Eds. Cham: Springer International Publishing, 2016, pp. 191–206. [Online]. Available: http://doi.org/10.1007/978-3-319-49890-4_11

[26] B. Koziel, A. Jalali, R. Azarderakhsh, D. Jao, and M. Mozaffari-Kermani, "NEON-SIDH: Efficient Implementation of Supersingular Isogeny Diffie-Hellman Key Exchange Protocol on ARM," in *Cryptology and Network Security: 15th International Conference, CANS 2016, Milan, Italy, November 14-16, 2016, Proceedings*, S. Foresti and G. Persiano, Eds. Cham: Springer International Publishing, 2016, pp. 88–103. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-48965-0_6

[27] A. K. Lenstra, "Generating RSA Moduli with a Predetermined Portion," in *Advances in Cryptology — ASIACRYPT'98: International Conference on the Theory and Application of Cryptology and Information Security Beijing, China, October 18–22, 1998 Proceedings*, K. Ohta and D. Pei, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 1–10. [Online]. Available: https://doi.org/10.1007/3-540-49649-1_1

[28] J. López and R. Dahab, "Fast Multiplication on Elliptic Curves Over GF(2m) without precomputation," in *Cryptographic Hardware and Embedded Systems: First International Workshop, CHES'99 Worcester, MA, USA, August 12–13, 1999 Proceedings*, Ç. K. Koç and C. Paar, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 316–327. [Online]. Available: http://dx.doi.org/10.1007/3-540-48059-5_27

[29] P. L. Montgomery, "Modular multiplication without trial division," *Mathematics of Computation*, vol. 44, pp. 519–521, 1985. [Online]. Available: http://doi.org/10.1090/S0025-5718-1985-0777282-X

[30] ——, "Speeding the Pollard and Elliptic Curve Methods of Factorization," *Mathematics of Computation*, vol. 48, no. 177, pp. 243–264, 1987. [Online]. Available: http://dx.doi.org/10.2307/2007888

[31] K. Okeya and K. Sakurai, "Efficient Elliptic Curve Cryptosystems from a Scalar Multiplication Algorithm with Recovery of the y-Coordinate on a Montgomery-Form Elliptic Curve," in *Cryptographic Hardware and Embedded Systems — CHES 2001: Third International Workshop Paris, France, May 14–16, 2001 Proceedings*, Ç. K. Koç, D. Naccache, and C. Paar, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 126–141. [Online]. Available: http://dx.doi.org/10.1007/3-540-44709-1_12

[32] T. Oliveira, D. F. Aranha, J. López, and F. Rodríguez-Henríquez, “Fast Point Multiplication Algorithms for Binary Elliptic Curves with and without Precomputation,” in *Selected Areas in Cryptography – SAC 2014: 21st International Conference, Montreal, QC, Canada, August 14-15, 2014, Revised Selected Papers*, A. Joux and A. Youssef, Eds. Cham: Springer International Publishing, 2014, pp. 324–344. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-13051-4_20

[33] T. Oliveira, J. López, H. Hışıl, A. Faz-Hernández, and F. Rodríguez-Henríquez, “How to (pre-)compute a ladder,” in *Selected Areas in Cryptography – SAC 2017: 24th International Conference, Ottawa, Ontario, Canada, August 16 - 18, 2017, Revised Selected Papers*, J. Camenisch and C. Adams, Eds. Springer International Publishing, Aug. 2017.

[34] J. Pierpont, “On an undemonstrated theorem of the Disquisitiones Arithmeticæ,” *Bull. Amer. Math. Soc.*, vol. 2, no. 3, pp. 77–83, 12 1895. [Online]. Available: http://projecteuclid.org/euclid.bams/1183414527

[35] A. Stolbunov, “Constructing public-key cryptographic schemes based on class group action on a set of isogenous elliptic curves,” *Advances in Mathematics of Communications*, vol. 4, no. 2, pp. 215–235, 2010. [Online]. Available: http://doi.org/10.3934/amc.2010.4.215

[36] S. R. Subramanya Rao, “Three Dimensional Montgomery Ladder, Differential Point Tripling on Montgomery Curves and Point Quintupling on Weierstrass’ and Edwards Curves,” in *Progress in Cryptology – AFRICACRYPT 2016: 8th International Conference on Cryptology in Africa, Fes, Morocco, April 13-15, 2016, Proceedings*, D. Pointcheval, A. Nitaj, and T. Rachidi, Eds. Cham: Springer International Publishing, 2016, pp. 84–106. [Online]. Available: https://doi.org/10.1007/978-3-319-31517-1_5

[37] C. D. Walter, “Montgomery exponentiation needs no final subtractions,” *Electronics Letters*, vol. 35, no. 21, pp. 1831–1832, Oct 1999. [Online]. Available: http://doi.org/10.1049/el:19991230

[38] L. C. Washington, *Elliptic Curves: Number Theory and Cryptography, Second Edition*, 2nd ed. Chapman & Hall/CRC, 2008.

[39] Y. Yoo, R. Azarderakhsh, A. Jalali, D. Jao, and V. Soukharev, “A Post-Quantum Digital Signature Scheme based on Supersingular Isogenies,” in *Financial Cryptography and Data Security: FC 2017 International Workshops, BITCOIN, TA, VOTING, WAHC, and WTSC, Malta, April, 2017.* Berlin, Heidelberg: Springer Berlin Heidelberg, Apr. 2017.