Faster Public-key Compression of SIDH with Less Memory

Kaizhan Lin[1], Jianming Lin[1], Weize Wang[1], and Chang-An Zhao[1,2,3]

[1] School of Mathematics, Sun Yat-sen University,
Guangzhou 510275, P.R.China
linkzh5@mail2.sysu.edu.cn
linjm28@mail2.sysu.edu.cn
wangwz@mail2.sysu.edu.cn
zhaochan3@mail.sysu.edu.cn

[2] Guangdong Key Laboratory of Information Security,
Guangzhou 510006, P.R. China

[3] State Key Laboratory of Information Security (Institute of Information Engineering), Chinese Academy of Sciences,
Beijing 100093, P.R. China.

Abstract. In recent years, the isogeny-based protocol, namely supersingular isogeny Diffie-Hellman (SIDH) has become highly attractive for its small public key size. In addition, public-key compression makes supersingular isogeny key encapsulation scheme (SIKE) more competitive in the NIST post-quantum cryptography standardization effort. However, compared to other post-quantum protocols, the computational cost of SIDH is relatively high, and so is public-key compression. On the other hand, the storage for pairing computation and discrete logarithms to speed up the current implementation of the key compression is somewhat large.

In this paper, we mainly improve the performance of public-key compression of SIDH, especially the efficiency and the storage of pairing computation involved. Our experimental results show that the memory requirement for pairing computation is reduced by a factor of about 1.5, and meanwhile, the instantiation of key generation of SIDH is 4.06% $\sim$ 7.23% faster than the current state-of-the-art.

Keywords: SIDH · SIKE · Post-quantum Cryptography · Public-key Compression · Bilinear Pairing

1 Introduction

As we all know, Shor's algorithm [22] is a polynomial time algorithm to solve factoring large integers or discrete logarithms on a quantum computer. It causes that most of widely-used cryptographic protocols are judged to be insecure, and simultaneously promotes the rise of post quantum cryptography. Since supersingular isogeny Diffie-Hellman (SIDH) [14] was introduced by Jao and De Feo in 2011, isogeny-based protocols have received worldwide attention and finally its variant, supersingular isogeny key encapsulation (SIKE) [1], remains one of the nine key encapsulation mechanisms in Round 3 of the NIST post-quantum cryptography standardization effort. Compared with other post quantum protocols, such as code-based and lattice-based schemes, isogeny-based protocols are attractive with their relatively small public keys.

Furthermore, one can compress the public key to make the SIDH protocol more competitive. SIDH public-key compression was firstly explored by R. Azarderakhsh et al. [2] in 2016, and further improved by Costello et al. [3] in 2017, reducing the key size to $3.5\log_2 p$ in the end. But the extra computational cost is unbearable, compared to SIDH without compression. Zanon et al. [24] later proposed new speed-up techniques to decrease the runtime of SIDH compression and decompression, while M. Naehrig and J.Renes [18] applied dual isogenies to significantly reduce the cost of compression. However, it needs precomputation. To make matters worse, storing tables for pairing computation and discrete logarithm solution need large storage. Recently A. Hutchinson et al. [13] reduced discrete logarithm table sizes by a factor of 4, thanks to the SIKE parameters and torus-based representation of cyclotomic subgroup elements. In addition, Pereira and Barreto [19] attempted to utilize ECDLP instead of DLP on finite fields and bilinear pairings for compression, leading to the saving of the memory requirements as well, but it is hard to be as efficient as the original.

Our main achievement is the optimization of bilinear pairings used in public-key compression of SIDH and SIKE, including its storage and efficiency. In both cases, we present techniques to make a memory saving of about 33.4% and improve the implementation of key generation about 4.06% $\sim$ 7.23% overhead, which is the main efficiency bottleneck for the current public-key compression of SIKE.

The paper is organized as follows. In Section 2 we briefly recall the SIDH protocol, the basic knowledge of the reduced Tate pairing and public-key compression used in SIKE. Section 3 presents our ideas to speed up pairing computation, especially Miller evaluations. Our implementation and cost estimates for Miller evaluations are presented in Section 4. Finally, we make a conclusion in Section 5.

2 Preliminaries

2.1 SIDH protocol

We simply review the SIDH protocol in this subsection. SIDH operates on supersingular elliptic curves of the Montgomery form $E_A : y^2 = x^3 + Ax^2 + x$. All the curves are defined over $\mathbb{F}_{p^2} = \mathbb{F}_p(i)$, where $i = \sqrt{-1}$ in $\mathbb{F}_{p^2}$, while $p = 2^{e_2}3^{e_3} - 1$ is prime and $2^{e_2} \approx 3^{e_3}$. Besides p, the public parameters of SIDH contain the supersingular curve $E_6 : y^2 = x^3 + 6x^2 + x$ and four torsion points P_A, Q_A, P_B, Q_B, satisfying that $E_6[2^{e_2}] = \langle P_A, Q_A\rangle$, $E_6[3^{e_3}] = \langle P_B, Q_B\rangle$. There are mainly two phases in SIDH: key generation and key agreement.

During key generation, Alice chooses a random integer s_A such that $s_A \in [0, 2^{e_2} - 1]$, and applies the three point ladder algorithm [10] to compute $S_A =$

$P_A + [s_A]Q_A$. Obviously it is a point of order 2^{e_2} over E_6. Next, Alice computes the 2^{e_2}-isogeny ϕ_A with kernel $\langle S_A \rangle$ by decomposing it as a chain of 2-isogenies, which can be obtained easily by Vélu's formula [23]. Finally, she computes $\phi_A(P_B)$, $\phi_A(Q_B)$ and the image curve parameter A, thereafter sends the triple $(\phi_A(P_B), \phi_A(Q_B), A)$ to Bob. Similar to Alice, Bob chooses a random element $s_B \in [0, 3^{e_3} - 1]$ as his secret key and computes the 3^{e_3}-isogeny ϕ_B with kernel $S_B = P_B + [s_B]Q_B$, and then calculates $\phi_B(P_A)$, $\phi_B(Q_A)$ and the image curve parameter B and transmits this information to Alice.

After receiving the message from Bob, Alice begins her key agreement phase. In the first instance she uses her secret key s_A and $\phi_B(P_A)$, $\phi_B(Q_A)$ to construct $\phi_B(P_A) + [s_A]\phi_B(Q_A)$ (Note that it is a point over E_B) and the corresponding 2^{e_2}-isogeny ϕ'_A. Different from the key generation phase, only the parameter of the image curve E_{BA} of ϕ'_A is needed. Analogously Bob acts the 3^{e_3}-isogeny with kernel $\phi_A(P_B) + [s_B]\phi_A(Q_B)$ on E_A and finds out its image curve E_{AB}. Since $E_{BA} \cong E_{AB}$, they can share the j-invariant of E_{AB} and E_{BA} as their secret.

For more details of the SIDH protocol and its quantum security analysis, we refer to [1, 7, 12, 14, 15].

Remark 1. Indeed, Alice can only utilize the x-coordinates of $P_A, Q_A, R_A = P_A - Q_A, P_B, Q_B, R_B = P_B - Q_B$ and her secret key to complete all the work during the whole process of key generation. In this case Alice should transmit the triple $(x_{\phi_A(P_B)}, x_{\phi_A(Q_B)}, x_{\phi_A(R_B)})$ to Bob. The same situation makes available for Bob. Furthermore, the key agreement phase for both of them can be also optimized in the same way. See [5, 14] for more details.

2.2 Reduced Tate pairing

Before describing public-key compression of SIDH, we introduce the reduced Tate pairing [9] first. It is a variant of Tate pairing [11], guaranteeing the uniqueness of pairing value.

Let $k = \mathbb{F}_q$ be a field and E an elliptic curve over k. The reduced Tate pairing of order n is denoted by

$$e_n : E[n] \times E(k)/nE(k) \to \mu, \quad (P, Q) \mapsto f_{n,P}(Q)^{\frac{q-1}{n}},$$

where μ is the set of n-th roots of unity, and $f_{n,P}$ a rational function satisfying

$$\mathrm{div}(f_{n,P}) = n(P) - n(\mathcal{O}).$$

This kind of pairing also has the same properties as the Tate pairing, i.e.,

– Bilinearity: $\forall P_1, P_2 \in E[n]$, $\forall Q_1, Q_2 \in E(k)/nE[k]$,

$$e_n(P_1 + P_2, Q) = e_n(P_1, Q)e_n(P_2, Q),$$

$$e_n(P, Q_1 + Q_2) = e_n(P, Q_1)e_n(P, Q_2);$$

- Non-degeneracy: $\forall\ P \in E[n]$, $\exists Q \in E(k)/nE(k)$ such that
$$e_n(P,Q) \neq 1.$$
And similarly, for all $Q \in E(k)/nE(k)$ there exists a point $P \in E[n]$ satisfying the above inequality;
- Compatibility with isogenies: $\forall P \in E[n], \forall Q \in E(k)/nE[k]$,
$$e_n(P,\phi_m(Q)) = e_n(\widehat{\phi_m}(P),Q),$$
where ϕ_m is a non-zero m-isogeny defined over $\mathbb{F}_q$, and $\widehat{\phi_m}$ is its dual, in particular,
$$e_n(\phi_m(P),\phi_m(Q)) = e_n(P,Q)^{m^2}.$$

2.3 Public-key compression of SIDH

In Remark 1 we can see that the size of the public key is $6\log_2 p$ bits. However, one can reduce the size to $3.5\log_2 p$ bits at a greater cost of computational resources. Now we briefly recall public-key compression used in SIKE and focus on pairing computation. For the sake of simplicity we only consider the case of Alice, while that of Bob is similar.

2.3.1 Linear representation Azarderakhsh et al. [2] firstly proposed a way to reduce the key size. The main idea is to implement a deterministic pseudo-random number generator to find out another 3^{e_3}-torsion basis w.r.t. the curve parameter A to linearly represent $\phi_A(P_B)$ and $\phi_A(Q_B)$:
$$\begin{bmatrix} \phi_A(P_B) \\ \phi_A(Q_B) \end{bmatrix} = \begin{bmatrix} a_0 & b_0 \\ a_1 & b_1 \end{bmatrix} \begin{bmatrix} U_A \\ V_A \end{bmatrix}. \tag{1}$$

A question raised here is how to compute a_0, b_0, a_1, b_1. Taking advantage of the bilinearity and non-degeneracy of the reduced Tate pairing, we consider
$$\begin{aligned}
g_0 &= \mathrm{e}_{3^{e_3}}(U_A, V_A), \\
g_1 &= \mathrm{e}_{3^{e_3}}(U_A, \phi_A(P_B)) = \mathrm{e}_{3^{e_3}}(U_A, a_0U_A + b_0V_A) = g_0^{b_0}, \\
g_2 &= \mathrm{e}_{3^{e_3}}(U_A, \phi_A(Q_B)) = \mathrm{e}_{3^{e_3}}(U_A, a_1U_A + b_1V_A) = g_0^{b_1}, \\
g_3 &= \mathrm{e}_{3^{e_3}}(V_A, \phi_A(P_B)) = \mathrm{e}_{3^{e_3}}(V_A, a_0U_A + b_0V_A) = g_0^{-a_0}, \\
g_4 &= \mathrm{e}_{3^{e_3}}(V_A, \phi_A(Q_B)) = \mathrm{e}_{3^{e_3}}(V_A, a_1U_A + b_1V_A) = g_0^{-a_1}.
\end{aligned} \tag{2}$$

It remains how to solve four discrete logarithms, which is easy by using the Pohlig-Hellman algorithm [21].

Instead of $(x_{\phi_A(P_B)}, x_{\phi_A(Q_B)}, x_{\phi_A(R_B)})$, Alice regards (a_0, b_0, a_1, b_1, A) as her public key and dispatches it to Bob. Note that $a_0, b_0, a_1, b_1 \in \mathbb{Z}/3^{e_3}\mathbb{Z}$ and $A \in \mathbb{F}_{p^2}$, the size is reduced to about $4\log_2 p$.

Remark 2. After receiving the message from Alice, Bob can implement the same pseudo-random number generator w.r.t. A to generate U_A, V_A, and hence he is able to recover $\phi_A(P_B)$ and $\phi_A(Q_B)$. The efficient way to generate the torsion basis can be seen in [20, 24].

Costello et al. [3] observed that either a_0 or b_0 is invertible, and therefore $(a_0^{-1}b_0, a_0^{-1}a_1, a_0^{-1}b_1, 0, A)$ (or $(b_0^{-1}a_0, b_0^{-1}a_1, b_0^{-1}b_1, 1, A)$), whose size is approximate $3.5\log_2 p$, could substitute for (a_0, b_0, a_1, b_1, A). In this case, Bob could only construct the kernel of the isogeny ϕ_B', instead of restoring $\phi_A(P_B)$ and $\phi_A(Q_B)$, but it does not affect the progress of key agreement.

2.3.2 Reverse basis decomposition Further optimization was explored by Zanon et. al [24]. Due to the fact that $\langle \phi_A(P_B), \phi_A(Q_B) \rangle = E_A[3^{e_3}]$, the matrix in Equation (1) is invertible and therefore,

$$\begin{bmatrix} U_A \\ V_A \end{bmatrix} = \begin{bmatrix} c_0 & d_0 \\ c_1 & d_1 \end{bmatrix} \begin{bmatrix} \phi_A(P_B) \\ \phi_A(Q_B) \end{bmatrix}.$$

It is easy to check that $(a_0^{-1}b_0, a_0^{-1}a_1, a_0^{-1}b_1) = (-d_1^{-1}d_0, -d_1^{-1}c_1, d_1^{-1}c_0)$ if d_1 is invertible (If not then $(b_0^{-1}a_0, b_0^{-1}a_1, b_0^{-1}b_1) = (-d_0^{-1}d_1, d_0^{-1}c_1, -d_0^{-1}c_0)$ holds). In this way Equation (2) need to be modified correspondingly,

$$\begin{aligned}
h_0 &= \mathrm{e}_{3^{e_3}}\left(\phi_A(P_B), \phi_A(Q_B)\right) \\
&= \mathrm{e}_{3^{e_3}}\left(P_B, Q_B\right)^{2^{e_2}}, \\
h_1 &= \mathrm{e}_{3^{e_3}}\left(\phi_A(P_B), U_A\right) \\
&= \mathrm{e}_{3^{e_3}}\left(\phi_A(P_B), c_0\phi_A(P_B) + d_0\phi_A(Q_B)\right) = h_0^{d_0}, \\
h_2 &= \mathrm{e}_{3^{e_3}}\left(\phi_A(P_B), V_A\right) \\
&= \mathrm{e}_{3^{e_3}}\left(\phi_A(P_B), c_1\phi_A(P_B) + d_1\phi_A(Q_B)\right) = h_0^{d_1}, \\
h_3 &= \mathrm{e}_{3^{e_3}}\left(\phi_A(Q_B), U_A\right) \\
&= \mathrm{e}_{3^{e_3}}\left(\phi_A(Q_B), c_0\phi_A(P_B) + d_0\phi_A(Q_B)\right) = h_0^{-c_0}, \\
h_4 &= \mathrm{e}_{3^{e_3}}\left(\phi_A(Q_B), V_A\right) \\
&= \mathrm{e}_{3^{e_3}}\left(\phi_A(Q_B), c_1\phi_A(P_B) + d_1\phi_A(Q_B)\right) = h_0^{-c_1}.
\end{aligned}$$

The superiority of reverse basis decomposition is that the value h_0 could be precomputed, i.e., powers of h_0 could be calculated in advance, bringing more efficiency for solution of four discrete logarithms.

Remark 3. Utilizing torus-based representation of cyclotomic subgroup elements and signed digit representation, Hutchinson et al. [13] reduced the memory requirements for computing discrete logarithms by a factor of 4.

2.3.3 Dual isogeny Naehrig and Renes [18] did a deeper study for public-key compression and made it possible to speed up pairing computation. They made use of the fact that

$$\begin{aligned}
h_1 &= e_{3^{e_3}}\left(P_B, \widehat{\phi}_A(U_A)\right), \; h_2 = e_{3^{e_3}}\left(P_B, \widehat{\phi}_A(V_A)\right), \\
h_3 &= e_{3^{e_3}}\left(Q_B, \widehat{\phi}_A(U_A)\right), \; h_4 = e_{3^{e_3}}\left(Q_B, \widehat{\phi}_A(V_A)\right),
\end{aligned}$$

where $\widehat{\phi}_A$ is the dual isogeny of ϕ_A.

The existence of $P_3 \in E_0(\mathbb{F}_p)[3^{e_3}]$ and the distortion map $\psi : (x, y) \mapsto (-x, iy)$ help construct a $3^{e_3}-$torsion group $\langle P_3, \psi(P_3)\rangle$. Let

$$P_B = \phi_0(P_3), Q_B = \phi_0(\psi(P_3)),$$

where ϕ_0 is the 2-isogeny of kernel $\langle(0, 0)\rangle$ form E_0 to E_6. Then $\langle P_B, Q_B\rangle$ is the $3^{e_3}-$torsion group of E_6. Same as above, pulling back pairing computation from E_6 to E_0 is feasible. In this case, computing h_1, h_2, h_3 and h_4 is efficient because of the special form of P_3 and $\psi(P_3)$.

Moreover, once P_3, Q_3 are fixed, one could precompute all the coefficients of Miller line functions to further speedup pairing computation. Nevertheless, it requires huge memory requirements and pairing computation is still the bottleneck of public-key compression.

Remark 4. The situation changes when Bob computes the four order-2^{e_2} pairings because it is impossible to seek out a point of order 2^{e_2} over $E_0(\mathbb{F}_p)$ [5], but the handling is analogous. Instead of pulling back to E_0, consider the isomorphism

$$\begin{aligned} \varphi : E_6 &\to E_{-11,14}, \\ (x, y) &\mapsto (x + 2, y), \end{aligned}$$

where $E_{-11,14} : y^2 = x^3 - 11x + 14$. It helps pull computations from E_B to $E_{-11,14}$. Since there does not exist a point of order 2^{e_2} over $E_6(\mathbb{F}_p)$, the best choice is to pick P_2 and Q_2 such that $[2]P_2 \in E_{-11,14}(\mathbb{F}_p)$ and $[2]Q_2 = (x, iy) \in E_{-11,14}(\mathbb{F}_{p^2})$, where $x, y \in \mathbb{F}_p$ [6]. The rest is similar to the case of Alice.

3 Pairing Optimization

As mentioned in Section 2.3.3, pairing computation over E_0 (or $E_{-11,14}$) dominants the time complexity of public-key compression. In this section we propose how to optimize it, according to the specific setting of SIKE, to make SIDH/SIKE more competitive.

Throught this section, let $L_{[\alpha]P,[\beta]P}$ be the line passing through $[\alpha]P$ and $[\beta]P$, where P is a rational point over an elliptic curve and $\alpha, \beta \in \mathbb{Z}$. Besides, use $g_{[\alpha]P,[\beta]P}$ and $v_{[\alpha]P}$ to denote the line functions that defines the lines $L_{[\alpha]P,[\beta]P}$ and $L_{[\alpha]P,-[\alpha]P}$, respectively. We abbreviate $g_{[\alpha]P,[\beta]P}$ and $v_{[\alpha]P}$ to $g_{\alpha,\beta}$ and v_α respectively when it does not cause ambiguous interpretation.

Remark 5. $L_{[\alpha]P,[\alpha]P}$ represents the line passing through the point $[\alpha]P$ twice. This means that $L_{[\alpha]P,[\alpha]P}$ is the tangent line of the curve at P, and, of course, it intersects the curve at $[-2\alpha]P$.

3.1 Handling the case of Alice

For Alice, she should compute four reduced Tate pairings of the form $e_{3^{e_3}}(R, S)$, where R is either P_3 or $\psi(P_3)$ while S is an indeterminate point. Each pairing computation consists of two stages: the Miller function construction and the final exponentiation. The latter is scarcely possible to improve for the low embedding degree, thus we focus on the former.

Since 3^{e_3} is a smooth number, it is best to compute the order-3^{e_3} pairing in a double-and-add like fashion by using Miller's algorithm [17], and each Miller iteration (except the final loop) is a tripling step of the form

$$\mathrm{div}\left(f_{3^{j+1},R}\right) = \mathrm{div}\left(f_{3^j,R}^3 \cdot \frac{g_{3^j,3^j} \cdot g_{3^j,2\cdot 3^j}}{v_{2\cdot 3^j} \cdot v_{3^{j+1}}}\right),$$

where $j = 0, 1, \cdots, e_3 - 2$. Use the notations $[3^j]R = (x_1^{(j)}, y_1^{(j)})$, $[2 \cdot 3^j]R = (x_2^{(j)}, y_2^{(j)})$, $[3^{j+1}]R = (x_3^{(j)}, y_3^{(j)})$, and let $\lambda_1^{(j)}$ and $\lambda_2^{(j)}$ be the slopes of $L_{[3^j]P,[3^j]P}$ and $L_{[3^j]P,[2\cdot 3^j]P}$ respectively, that is,

$$\lambda_1^{(j)} = \frac{3(x_1^{(j)})^2 + 1}{2y_1^{(j)}}, \lambda_2^{(j)} = \frac{y_2^{(j)} - y_1^{(j)}}{x_2^{(j)} - x_1^{(j)}}.$$

Then we have

$$\begin{aligned}
g_{3^j,3^j} &= \lambda_1^{(j)}(x - x_1^{(j)}) - (y - y_1^{(j)}), \\
g_{3^j,2\cdot 3^j} &= \lambda_2^{(j)}(x - x_2^{(j)}) - (y - y_2^{(j)}), \\
v_{2\cdot 3^j} &= x - x_2^{(j)}, \\
v_{3^{j+1}} &= x - x_3^{(j)}.
\end{aligned}$$

Eisenträger et al. [8] utilized the double-and-add trick with parabolas to speed up the evaluations of the Weil and Tate pairings. We are inspired by their work and take full advantage of this relation between $g_{3^j,3^j}$, $g_{3^j,2\cdot 3^j}$ and $v_{3^{j+1}}$ to optimize the implementation of the reduced Tate pairing.

As we mentioned in Remark 5, $L_{[3^j]P,[3^j]P}$ is a line passing through not only $[3^j]P$, but $[-2 \cdot 3^j]P$, which is the inverse of $[2 \cdot 3^j]P$, namely, $[-2 \cdot 3^j]P = (x_2^{(j)}, -y_2^{(j)})$. This implies that

$$g_{3^j,3^j} = \lambda_1^{(j)}(x - x_2^{(j)}) - (y + y_2^{(j)}).$$

Hence,

$$\begin{aligned}
&\frac{g_{3^j,3^j} \cdot g_{3^j,2\cdot 3^j}}{v_{2\cdot 3^j}} \\
&= \frac{[\lambda_1^{(j)}(x - x_2^{(j)}) - (y + y_2^{(j)})] \cdot [\lambda_2^{(j)}(x - x_2^{(j)}) - (y - y_2^{(j)})]}{x - x_2^{(j)}} \\
&= \lambda_1^{(j)}\lambda_2^{(j)}(x - x_2^{(j)}) - \lambda_1^{(j)}(y - y_2^{(j)}) - \lambda_2^{(j)}(y + y_2^{(j)}) + \frac{y^2 - (y_2^{(j)})^2}{x - x_2^{(j)}}.
\end{aligned} \tag{3}$$

Note E_0 is of the form: $y^2 = x^3 + x$. Similar with the denominator elimination method proposed in [4, 16, 25], we have

$$\begin{aligned} y^2 - (y_2^{(j)})^2 &= (x^3 + x) - ((x_2^{(j)})^3 + x_2^{(j)}) \\ &= (x^3 - (x_2^{(j)})^3) + (x - x_2^{(j)}) \\ &= (x - x_2^{(j)})(x^2 + x_2^{(j)}x + (x_2^{(j)})^2) + (x - x_2^{(j)}) \\ &= (x - x_2^{(j)})(x^2 + x_2^{(j)}x + (x_2^{(j)})^2 + 1). \end{aligned} \tag{4}$$

Thus we can further simplify the computation of Equation (3):

$$\begin{aligned} &\frac{g_{3^j,3^j} \cdot g_{3^j,2\cdot 3^j}}{v_{2\cdot 3^j}} \\ &= \lambda_1^{(j)}\lambda_2^{(j)}(x - x_2^{(j)}) - \lambda_1^{(j)}(y - y_2^{(j)}) - \lambda_2^{(j)}(y + y_2^{(j)}) + \frac{y^2 - (y_2^{(j)})^2}{x - x_2^{(j)}} \\ &= \lambda_1^{(j)}\lambda_2^{(j)}(x - x_2^{(j)}) - \lambda_1^{(j)}(y - y_2^{(j)}) - \lambda_2^{(j)}(y + y_2^{(j)}) + (x^2 + x_2^{(j)}x + (x_2^{(j)})^2 + 1) \\ &= x^2 + (x_2^{(j)} + \lambda_1^{(j)}\lambda_2^{(j)})x - (\lambda_1^{(j)} + \lambda_2^{(j)})y + C^{(j)}, \end{aligned}$$

where $C^{(j)} = (x_2^{(j)} - \lambda_1^{(j)}\lambda_2^{(j)})x_2^{(j)} + (\lambda_1^{(j)} - \lambda_2^{(j)})y_2^{(j)} + 1$. Furthermore,

$$\operatorname{div}\left(f_{3^{j+1},R}\right) = \operatorname{div}\left(f_{3^j,R}^3 \cdot \frac{x^2 + (x_2^{(j)} + \lambda_1^{(j)}\lambda_2^{(j)})x - (\lambda_1^{(j)} + \lambda_2^{(j)})y + C^{(j)}}{x - x_3^{(j)}}\right).$$

Obviously, each Miller loop would be more efficient if we precompute all the following values:

$$\begin{aligned} t_0^{(j)} &= x_3^{(j)}, \\ t_1^{(j)} &= x_2^{(j)} + \lambda_1^{(j)}\lambda_2^{(j)}, \\ t_2^{(j)} &= \lambda_1^{(j)} + \lambda_2^{(j)}, \\ t_3^{(j)} &= (x_2^{(j)} - \lambda_1^{(j)}\lambda_2^{(j)})x_2^{(j)} + (\lambda_1^{(j)} - \lambda_2^{(j)})y_2^{(j)} + 1. \end{aligned} \tag{5}$$

For the final loop we only need to precompute $x_1^{(e_3-1)}$ and $c = \lambda_1^{(e_3-1)}x_1^{(e_3-1)} - y_1^{(e_3-1)}$ since at the moment,

$$\operatorname{div}\left(f_{3^{e_3},R}\right) = \operatorname{div}\left(f_{3^{e_3-1},R}^3 \cdot g_{3^{e_3-1},3^{e_3-1}}\right) = \operatorname{div}\left(f_{3^{e_3-1},R}^3 \cdot [\lambda_1^{(e_3-1)}x - y - c]\right). \tag{6}$$

In general, there are $4(e_3 - 1) + 2$ elements in $\mathbb{F}_p$ required to precompute for the evaluation of $f_{P_3}(S)$.

Realizing that in the case of $R = \psi(P_3)$, we are able to make full use of the precomputed value for P_3:

$$\operatorname{div}\left(f_{3^{j+1},\psi(P_3)}\right) = \operatorname{div}\left(f_{3^j,\psi(P_3)}^3 \cdot \frac{x^2 - t_1^{(j)}x + it_2^{(j)}y + t_3^{(j)}}{x + t_0^{(j)}}\right),$$

and the final loop is also no exception. As a result, it is only necessary to store $4e_3 - 2$ elements in $\mathbb{F}_p$.

In Algorithm 1 we present pseudocode for $f_{3^{e_3},R}(U_k)$, where R is either P_3 or $\psi(P_3)$ and U_k $(k = 0, 1)$ are points of order 3^{e_3} over E_0. Denote by m and a the cost of one multiplication and one addition in $\mathbb{F}_p$, respectively. Then each Miller loop (except the final loop) needs a computational cost of $2(26m+49a) = 52m + 98a$. Compared with previous work [18], we save $16m + 36a$ per step.

Remark 6. It seems that for each Miller iteration one inversion operation in $\mathbb{F}_{p^2} = \mathbb{F}_p(i)$ is required. However, for all element $a + bi \in \mathbb{F}_p(i)$,

$$\left(\frac{1}{a+bi}\right)^{\frac{p^2-1}{2^{e_2}}} = \left(\left(\frac{1}{a+bi}\right)^{(p-1)}\right)^{3^{e_3}} = \left(\frac{(a-bi)^{(p-1)}}{(a^2+b^2)^{(p-1)}}\right)^{3^{e_3}} = (a-bi)^{\frac{p^2-1}{2^{e_2}}},$$

where $a, b \in \mathbb{F}_p$. Therefore, the inversion of an element could be replaced by its conjugate thanks to the final exponentiation. In the case of Bob, one can also utilize this trick to make algorithms more efficient.

Remark 7. In Algorithm 1, the notation of the form x_P represents the x-coordinate of P. We use $t_0^{(j)}$, $t_1^{(j)}$, $t_2^{(j)}$ and $t_3^{(j)}$ to denote precomputed values for P_3 mentioned in Equation (5). Also, $\lambda_1^{(e_3-1)}$ and c are precomputed values mentioned in Equation (6).

3.2 Handling the case of Bob

Bob needs to compute four pairing evaluations of the form $e_{2^{e_2}}(R, S)$, where R is either P_2 or Q_2 while S is undetermined. Analogously, it is hard to speed up the final exponentiation, thus we are still concerned with the Miller iteration.

It seems that the trick used in Section 3.1 is difficult to operate because each Miller iteration is too simple. However, if we take a bigger step — that is, combining two Miller iterations into one step, we find that the above trick is still able to work. We give the explanation in Appendix A. Here we present another way to make the pairing evaluations perform better. Furthermore, this method consumes less memory, compared to the former one.

We first investigate the diviosrs of $f_{4^{j+1},R}$ and $f_{4^j,R}$, and try to deduce the relation between $f_{4^{j+1},R}$ and $f_{4^j,R}$.

Since

$$\mathrm{div}(f_{4^{j+1},R}) = 4^{j+1}(R) - ([4^{j+1}]R) - (4^{j+1} - 1)(\mathcal{O}),$$

and

$$\mathrm{div}(f_{4^j,R}) = 4^j(R) - ([4^j]R) - (4^j - 1)(\mathcal{O}).$$

Therefore, we have

$$\mathrm{div}(f_{4^{j+1},R}) = 4\mathrm{div}(f_{4^j,R}) + 4([4^j]R) - ([4^{j+1}]R) - 3(\mathcal{O}).$$

Algorithm 1 Computation of four Miller evaluations in the case of Alice

Input: U_0, $U_1 \in E_0(\mathbb{F}_{p^2})$
Output: $f_{P_3}(U_0)$, $f_{P_3}(U_1)$, $f_{\psi(P_3)}(U_0)$ **and** $f_{\psi(P_3)}(U_1)$

for each $k \in \{0, 1\}$ **do**
  $f_k \leftarrow 1,\ f_{k+2} \leftarrow 1,\ s_k \leftarrow x_{U_k}^2$
end for
for each $j \in \{0, 1, \cdots, e_3 - 1\}$ **do**
  for each $k \in \{0, 1\}$ **do**
    $temp_1 \leftarrow x_{U_k} \cdot t_1^{(j)},\ temp_2 \leftarrow y_{U_k} \cdot t_2^{(j)},$
    $g \leftarrow s_{U_k} + temp_1,\ g \leftarrow g - temp_2,\ g \leftarrow g + t_3^{(j)},$
    $h \leftarrow x_{U_k} - t_0^{(j)},\ h \leftarrow h^*,$
    $temp_3 \leftarrow g \cdot h,$
    $temp_4 \leftarrow f_k,\ f_k \leftarrow f_k^2,\ f_k \leftarrow f_k \cdot temp_4,\ f_k \leftarrow f_k \cdot temp_3,$
    $g \leftarrow s_{U_k} - temp_1,\ g \leftarrow g + i \cdot temp_2,\ g \leftarrow g + t_3^{(j)},$
    $h \leftarrow x_{U_k} + t_0^{(j)},\ h \leftarrow h^*,$
    $temp_3 \leftarrow g \cdot h,$
    $temp_4 \leftarrow f_{k+2},\ f_{k+2} \leftarrow f_{k+2}^2,\ f_{k+2} \leftarrow f_{k+2} \cdot temp_4,\ f_{k+2} \leftarrow f_{k+2} \cdot temp_3.$
  end for
end for
for each $k \in \{0, 1\}$ **do**
  $temp_1 \leftarrow \lambda_1^{(e_3-1)} \cdot x_{U_k},$
  $g \leftarrow temp_1 - y_{U_k}^{(e_3-1)},\ g \leftarrow g + c,$
  $temp_2 \leftarrow f_k,\ f_k \leftarrow f_k^2,\ f_k \leftarrow f_k \cdot temp_2,\ f_k \leftarrow f_k \cdot g,$
  $g \leftarrow -i \cdot temp_1 - y_{U_k}^{(e_3-1)},\ g \leftarrow g - i \cdot c,$
  $temp_2 \leftarrow f_{k+2},\ f_{k+2} \leftarrow f_{k+2}^2,\ f_{k+2} \leftarrow f_{k+2} \cdot temp_2,\ f_{k+2} \leftarrow f_{k+2} \cdot g.$
end for
return $f_0,\ f_1,\ f_2,\ f_3.$

It remains to find a rational function $\ell^{(j)}$, whose divisor is

$$\mathrm{div}(\ell^{(j)}) = 4([4^j]R) - ([4^{j+1}]R) - 3(\mathcal{O}).$$

Note that

$$\begin{aligned}
\mathrm{div}(\ell^{(j)}) &= 4([4^j]R) - ([4^{j+1}]R) - 3(\mathcal{O}) \\
&= 4([4^j]R) + 2([-2 \cdot 4^j]R) - 2([-2 \cdot 4^j]R) - ([4^{j+1}]R) - 3(\mathcal{O}) \\
&= 4([4^j]R) + 2([-2 \cdot 4^j]R) - 6(\mathcal{O}) - 2([-2 \cdot 4^j]R) - ([4^{j+1}]R) + 3(\mathcal{O}) \\
&= 2\left(2([4^j]R) + ([-2 \cdot 4^j]R) - 3(\mathcal{O})\right) - \left(2([-2 \cdot 4^j]R) + ([4^{j+1}]R) - 3(\mathcal{O})\right).
\end{aligned}$$

That is,

$$\mathrm{div}(\ell^{(j)}) = 2\mathrm{div}(g_{4^j,4^j}) - \mathrm{div}(g_{-2\cdot 4^j,-2\cdot 4^j}),$$

where

$$\begin{aligned}
\mathrm{div}(g_{4^j,4^j}) &= 2([4^j]R) + ([-2 \cdot 4^j]R) - 3(\mathcal{O}), \\
\mathrm{div}(g_{-2\cdot 4^j,-2\cdot 4^j}) &= 2([-2 \cdot 4^j]R) + ([4^{j+1}]R) - 3(\mathcal{O}).
\end{aligned} \tag{7}$$

As a summary, we claim the following lemma.

Lemma 1. *For $j \in \mathbb{N}$, we have*

$$\mathrm{div}(f_{4^{j+1},R}) = \mathrm{div}\left(f^4_{4^j,R} \cdot \frac{g^2_{4^j,4^j}}{g_{-2\cdot 4^j,-2\cdot 4^j}}\right). \tag{8}$$

Use $(x_1^{(j)}, y_1^{(j)})$ and $(x_2^{(j)}, y_2^{(j)})$ to respectively denote the affine coordinates of $[4^j]R$ and $[-2\cdot 4^j]R$. Let λ_1 and λ_2 be the slopes of $L_{[4^j]R,[4^j]R}$, $L_{[-2\cdot 4^j]R,[-2\cdot 4^j]R}$:

$$\lambda_1^{(j)} = \frac{(3x_1^{(j)})^2 - 11}{2y_1^{(j)}}, \lambda_2^{(j)} = \frac{(3x_2^{(j)})^2 - 11}{2y_2^{(j)}}.$$

Then all the straight line functions mentioned in Equation (7) can be represented as

$$\begin{aligned} g_{4^j,4^j} &= \lambda_1^{(j)}(x - x_1^{(j)}) - (y - y_1^{(j)}), \\ g_{2\cdot 4^j,2\cdot 4^j} &= \lambda_2^{(j)}(x - x_2^{(j)}) - (y - y_2^{(j)}). \end{aligned}$$

Now we analyze the precomputed values that should be stored for each Miller iteration. Since

$$\begin{aligned} g_{4^j,4^j} &= \lambda_1^{(j)}(x - x_1^{(j)}) - (y - y_1^{(j)}) = \lambda_1^{(j)}x - y + (-\lambda_1^{(j)}x_1^{(j)} + y_1^{(j)}), \\ g_{-2\cdot 4^j,-2\cdot 4^j} &= \lambda_2^{(j)}(x - x_2^{(j)}) - (y + y_2^{(j)}) = \lambda_2^{(j)}x - y + (-\lambda_2^{(j)}x_2^{(j)} - y_2^{(j)}), \end{aligned}$$

we can only precompute the following four values to speed up the Miller loop:

$$\begin{aligned} t_0^{(j)} &= \lambda_1^{(j)}, \\ t_1^{(j)} &= -\lambda_1^{(j)}x_1^{(j)} + y_1^{(j)}, \\ t_2^{(j)} &= \lambda_2^{(j)}, \\ t_3^{(j)} &= -\lambda_2^{(j)}x_2^{(j)} - y_2^{(j)}. \end{aligned} \tag{9}$$

In the implementation we still execute a doubling step for $P_2, Q_2 \in \mathbb{F}_{p^2}$, while the rest are essentially operated in $\mathbb{F}_p$, and we take quadrupling steps as possible. The final loop is relatively easy:

$$\mathrm{div}(f_{2^{e_2},R}) = \begin{cases} \mathrm{div}\left(f^2_{2^{e_2-1},R} \cdot \left(x - x_{[2^{e_2-1}]R}\right)\right), \text{if } e_2 \text{ is even}, \\ \mathrm{div}\left(f^4_{2^{e_2-2},R} \cdot \dfrac{[\lambda_{[2^{e_2-2}]R}\left(x - x_{[2^{e_2-1}]R}\right) - y]^2}{x - x_{[2^{e_2-1}]R}}\right), \text{otherwise}, \end{cases} \tag{10}$$

where $\lambda_{[2^{e_2-2}]R}$ is the slope of the line $L_{[2^{e_2-2}]R,[2^{e_2-2}]R}$, and $x_{[2^{e_2-1}]R}$ is the x-coordinate of $[2^{e_2-1}]R$.

Different from P_3 and Q_3, there is no notable relation between P_2 and Q_2, so we have to precompute 8 values over $\mathbb{F}_p$ for each loop of four Miller evaluations, and the last step requires extra 4 values when e_2 is odd. To sum up, there are $4e_2 + 2$ or $4e_2 + 4$ elements to be precomputed and stored with respect to the parity of e_2.

In Algorithms 2 and 3 we present pseudocode for evaluating $f_{2^{e_2},R}(U_k)$, where R is either P_2 or Q_2, and U_k $(k = 1, 2)$ are two points of order 2^{e_2} over $E_{-11,14}$. The computational cost for each Miller iteration (except the first loop and the final loop) is $2(14m + 25a) + 2(14m + 23a) = 56m + 96a$. Compared with the work of M. Naehrig and J.Renes [18], we save $24m + 56a$ per quadrupling step, i.e, $12m + 28a$ per doubling step on average.

Remark 8. In Algorithms 2 and 3, the notations of the form x_P, y_P represent the x-coordinate and y-coordinate of P, while the notation λ_P is the slope of the tangent line passing through P. We use $t_0^{(j)}$, $t_1^{(j)}$, $t_2^{(j)}$ and t_3^j to denote precomputed values for P_2 (or Q_2) mentioned in Equation (18).

Algorithm 2 Computation of the Miller evaluations when $R = P_2$ in the case of Bob

Input: U_0, $U_1 \in E_0(\mathbb{F}_{p^2})$
Output: $f_{P_2}(U_0)$, $f_{P_2}(U_1)$

for each $k \in \{0, 1\}$ **do**
  $h \leftarrow x_{U_k} - x_{[2]P_2}$,
  $g \leftarrow h$, $g \leftarrow \lambda_{P_2} \cdot g$, $g \leftarrow g - y_{U_k}$, $g \leftarrow g - y_{[2]P_2}$,
  $h \leftarrow h^*$, $g \leftarrow g \cdot h$,
  $f_k \leftarrow g$.
end for
for each $j \in \{0, 1, \cdots, \frac{\lfloor e_2 - 1 \rfloor}{2}\}$ **do**
  for each $k \in \{0, 1\}$ **do**
    $g \leftarrow t_0^{(j)} \cdot x_{U_k}$, $g \leftarrow g - y_{U_k}$, $g \leftarrow g + t_1^{(j)}$,
    $h \leftarrow t_2^{(j)} \cdot x_{U_k}$, $h \leftarrow h - y_{U_k}$, $h \leftarrow h + t_3^{(j)}$, $h \leftarrow h^*$,
    $f_k \leftarrow f_k^2$, $f_k \leftarrow f_k \cdot g$, $f_k \leftarrow f_k^2$, $f_k \leftarrow f_k \cdot h$.
  end for
end for
if e_2 is even **then**
  for each $k \in \{0, 1\}$ **do**
    $g \leftarrow x - x_{[2^{e_2-1}]P_2}$,
    $f_k \leftarrow f_k^2$, $f_k \leftarrow f_k \cdot g$.
  end for
else
  for each $k \in \{0, 1\}$ **do**
    $h \leftarrow x_{U_k} - x_{[2^{e_2-1}]P_2}$,
    $g \leftarrow \lambda_{[2^{e_2-2}]P_2} \cdot h$, $g \leftarrow g - y_{U_k}$,
    $h \leftarrow h^*$,
    $f_k \leftarrow f_k^2$, $f_k \leftarrow f_k \cdot g$, $f_k \leftarrow f_k^2$, $f_k \leftarrow f_k \cdot g$.
  end for
end if
return f_0, f_1.

Algorithm 3 Computation of the Miller evaluations when $R = Q_2$ in the case of Bob

Input: U_0, $U_1 \in E_0(\mathbb{F}_{p^2})$
Output: $f_{Q_2}(U_0)$, $f_{Q_2}(U_1)$

for each $k \in \{0, 1\}$ **do**
  $h \leftarrow x_{U_k} - x_{[2]Q_2}$,
  $g \leftarrow h$, $g \leftarrow \lambda_{Q_2} \cdot g$, $g \leftarrow g - y_{U_k}$, $g \leftarrow g - i \cdot y_{[2]Q_2}$,
  $h \leftarrow h^*$, $g \leftarrow g \cdot h$,
  $f_k \leftarrow g$.
end for
for each $j \in \{0, 1, \cdots, \frac{\lfloor e_2 - 1 \rfloor}{2}\}$ **do**
  for each $k \in \{0, 1\}$ **do**
    $g \leftarrow i \cdot t_0^{(j)} \cdot x_{U_k}$, $g \leftarrow g - y_{U_k}$, $g \leftarrow g + i \cdot t_1^{(j)}$,
    $h \leftarrow i \cdot t_2^{(j)} \cdot x_{U_k}$, $h \leftarrow h - y_{U_k}$, $h \leftarrow h + i \cdot t_3^{(j)}$,
    $f_k \leftarrow f_k^2$, $f_k \leftarrow f_k \cdot g$, $f_k \leftarrow f_k^2$, $f_k \leftarrow f_k \cdot h$.
  end for
end for
if e_2 is even **then**
  for each $k \in \{0, 1\}$ **do**
    $g \leftarrow x - x_{[2^{e_2 - 1}]Q_2}$,
    $f_k \leftarrow f_k^2$, $f_k \leftarrow f_k \cdot g$.
  end for
else
  for each $k \in \{0, 1\}$ **do**
    $h \leftarrow x_{U_k} - x_{[2^{e_2 - 1}]Q_2}$,
    $g \leftarrow i \cdot \lambda_{[2^{e_2 - 2}]R} \cdot h$, $g \leftarrow g - y_{U_k}$,
    $h \leftarrow h^*$,
    $f_k \leftarrow f_k^2$, $f_k \leftarrow f_k \cdot g$, $f_k \leftarrow f_k^2$, $f_k \leftarrow f_k \cdot h$.
  end for
end if
return f_0, f_1.

4 Cost Estimates and Implementation

In this section we present our concrete cost estimates for four Miller evaluations and show the implementation of key generation by utilizing our techniques.

4.1 Cost estimates

In Section 3.1 and 3.2 we have analyzed the computational cost of each tripling and quadrupling step. Indeed, it is the main cost gap between previous work [18] and ours since the calculation of the Miller iteration is the main process of pairings. Tables 1 and 2 show our cost estimates for the computation of four Miller evaluations for all SIKE primes.

The estimates given in Tables 1 and 2 show that in the same condition, no matter what the characteristic of the finite field is given, our algorithms to

Table 1. Cost estimates (over the base field) of the previous work and ours (Algorithm 1) to compute four Miller evaluations of order-3^{e_3} pairings.

Parameters	Previous work [18]		Our work	
	Multiplication	Addition	Multiplication	Addition
SIKEp434	9314	18344	7112	13400
SIKEp503	10810	21292	8256	15556
SIKEp610	13054	25714	9972	18790
SIKEp751	16114	31744	12312	23200

Table 2. Cost estimates (over the base field) of the previous work and ours (Algorithms 2 and 3) to compute four Miller evaluations of order-2^{e_2} pairings.

Parameters	Previous work [18]		Our work	
	Multiplication	Addition	Multiplication	Addition
SIKEp434	8624	16404	6036	10368
SIKEp503	9984	18988	6988	12000
SIKEp610	12184	23168	8528	14638
SIKEp751	14864	28260	10404	17856

compute Miller evaluations can provide an acceleration. We can see that when computing Miller evaluations of order-3^{e_3} pairings, we save nearly one fourth of additions and multiplications. As for four Miller evaluations of order-2^{e_2} pairings, the ratio of the amount of multiplications and additions of our algorithms to those of the previous work are about 70% and 63%, respectively.

4.2 Implementation

Our code[1] is based on the SIDH C library[2]. We benchmarked our code on the Intel(R) Core(TM) i5-10210U CPU processor at 2.11 GHz running on 64-bit Linux. Since the implementation of computing discrete logarithms is not in constant time because of the randomness of the algorithms, we execute public-key compression 10^5 times and take the average cycle counts to make the data more reliable. The performance results are presented in Table 3. As expected, compared to the previous work, we reduce the computational cost for public-key compression.

Table 4 shows the storage comparison between the previous work and ours. It can be seen that in both cases we save about one-third memory. Therefore, some platforms, such as memory-constrained IoT devices, may benefit from our new algorithms.

[1] `https://github.com/LinKaizhan/SIDH_Faster_Comp`
[2] `https://github.com/Microsoft/PQCrypto-SIDH`

Table 3. Average computational cost (in millions of clock cycles) of key generation.

Parameters	Key generation of Alice			Key generation of Bob		
	Previous [18]	Ours	Speedup	Previous [18]	Ours	Speedup
SIKEp434	5.35	5.06	5.73%	5.34	4.98	7.23%
SIKEp503	7.44	7.15	4.06%	7.29	6.90	5.65%
SIKEp610	14.19	13.56	4.65%	13.36	12.81	4.29%
SIKEp751	22.13	21.28	3.99%	22.08	20.90	5.65%

Table 4. Storage requirements (in KiB) for pairing computation.

Parameters	Previous work [18]			This work		
	Pairings of order		Total	Pairings of order		Total
	3^{e_3}	2^{e_2}		3^{e_3}	2^{e_2}	
SIKEp434	44.8	70.7	115.5	29.9	47.4	77.2
SIKEp503	59.5	93.5	153.0	39.6	62.6	102.3
SIKEp610	89.8	142.7	232.5	59.8	95.3	155.2
SIKEp751	134.3	208.9	343.1	89.4	139.7	229.1

5 Conclusion

In this paper we mainly focused on the Miller evaluations calculated in public-key compression in the SIDH C library and presented several algorithms to improve the efficiency of pairing computation, which is the main bottleneck of compression. It is worth noting that we not only reduce the computational cost, but save nearly one-third memory to store the pre-computed tables. We believe that our work could make SIDH and SIKE more competitive among the post-quantum cryptography.

Acknowledgment

We thank the anonymous reviewers for their detailed and helpful comments. The work of Chang-An Zhao is partially supported by NSFC under Grant No. 61972428, by the Major Program of Guangdong Basic and Applied Research under Grant No. 2019B030302008 and by the Open Fund of State Key Laboratory of Information Security (Institute of Information Engineering, Chinese Academy of Sciences, Beijing 100093) under grant No. 2020-ZD-02.

References

1. Azarderakhsh, R., Campagna, M., Costello, C., De Feo, L., Hess, B., Hutchinson, A., Jalali, A., Jao, D., Karabina, K., Koziel, B., LaMacchia, B., Longa, P., Naehrig, M., Pereira, G., Renes, J., Soukharev, V., Urbanik, D.: Supersingular Isogeny Key Encapsulation (2020), `http://sike.org`

2. Azarderakhsh, R., Jao, D., Kalach, K., Koziel, B., Leonardi, C.: Key Compression for Isogeny-Based Cryptosystems. In: Proceedings of the 3rd ACM International Workshop on ASIA Public-Key Cryptography. pp. 1–10 (2016)
3. Costello, C., Jao, D., Longa, P., Naehrig, M., Renes, J., Urbanik, D.: Efficient Compression of SIDH Public Keys. In: Coron, J.S., Nielsen, J.B. (eds.) Advances in Cryptology – EUROCRYPT 2017. pp. 679–706. Springer International Publishing, Cham (2017)
4. Costello, C., Lange, T., Naehrig, M.: Faster Pairing Computations on Curves with High-Degree Twists. In: Nguyen, P.Q., Pointcheval, D. (eds.) Public Key Cryptography – PKC 2010. pp. 224–242. Springer Berlin Heidelberg, Berlin, Heidelberg (2010)
5. Costello, C., Longa, P., Naehrig, M.: Efficient Algorithms for Supersingular Isogeny Diffie-Hellman. In: Robshaw, M., Katz, J. (eds.) Advances in Cryptology – CRYPTO 2016. pp. 572–601. Springer Berlin Heidelberg, Berlin, Heidelberg (2016)
6. Costello, C., Longa, P., Naehrig, M., Renes, J., Virdia, F.: Improved Classical Cryptanalysis of SIKE in Practice. In: Kiayias, A., Kohlweiss, M., Wallden, P., Zikas, V. (eds.) Public-Key Cryptography – PKC 2020. pp. 505–534. Springer International Publishing, Cham (2020)
7. Dobson, S., Galbraith, S.D., LeGrow, J., Ti, Y.B., Zobernig, L.: An Adaptive Attack on 2-SIDH. International Journal of Computer Mathematics: Computer Systems Theory **5**(4), 282–299 (2020)
8. Eisenträger, K., Lauter, K., Montgomery, P.L.: Fast Elliptic Curve Arithmetic and Improved Weil Pairing Evaluation. In: Joye, M. (ed.) Topics in Cryptology — CT-RSA 2003. pp. 343–354. Springer Berlin Heidelberg, Berlin, Heidelberg (2003)
9. Enge, A.: Bilinear pairings on elliptic curves. L'Enseignement Mathématique **61**(1), 211–243 (2016)
10. Faz-Hernández, A., López, J., Ochoa-Jiménez, E., Rodríguez-Henríquez, F.: A Faster Software Implementation of the Supersingular Isogeny Diffie-Hellman Key Exchange Protocol. IEEE Transactions on Computers **67**(11), 1622–1636 (2018)
11. Frey, G., Rück, H.G.: A Remark Concerning m-Divisibility and the Discrete Logarithm in the Divisor Class Group of Curves. Math. Comput. **62**(206), 865–874 (1994)
12. Galbraith, S.D., Petit, C., Shani, B., Ti, Y.B.: On the Security of Supersingular Isogeny Cryptosystems. In: Cheon, J.H., Takagi, T. (eds.) Advances in Cryptology – ASIACRYPT 2016. pp. 63–91. Springer Berlin Heidelberg, Berlin, Heidelberg (2016)
13. Hutchinson, A., Karabina, K., Pereira, G.: Memory Optimization Techniques for Computing Discrete Logarithms in Compressed SIKE. In: Cheon, J.H., Tillich, J.P. (eds.) Post-Quantum Cryptography. pp. 296–315. Springer International Publishing, Cham (2021)
14. Jao, D., De Feo, L.: Towards Quantum-Resistant Cryptosystems from Supersingular Elliptic Curve Isogenies. In: Yang, B.Y. (ed.) Post-Quantum Cryptography. pp. 19–34. Springer Berlin Heidelberg, Berlin, Heidelberg (2011)
15. Jaques, S., Schanck, J.M.: Quantum Cryptanalysis in the RAM Model: Claw-Finding Attacks on SIKE. In: Boldyreva, A., Micciancio, D. (eds.) Advances in Cryptology – CRYPTO 2019. pp. 32–61. Springer International Publishing, Cham (2019)
16. Lin, X., Zhao, C.A., Zhang, F., Wang, Y.: Computing the Ate Pairing on Elliptic Curves with Embedding Degree $k = 9$. IEICE Trans. Fundam. Electron. Commun. Comput. Sci. **E91-A**(9), 2387–2393 (2008)

17. Miller, V.S.: The Weil Pairing, and Its Efficient Calculation. Journal of Cryptology **17**(4), 235–261 (2004)
18. Naehrig, M., Renes, J.: Dual Isogenies and Their Application to Public-Key Compression for Isogeny-Based Cryptography. In: Galbraith, S.D., Moriai, S. (eds.) Advances in Cryptology – ASIACRYPT 2019. pp. 243–272. Springer International Publishing, Cham (2019)
19. Pereira, G.C.C.F., Barreto, P.S.L.M.: Isogeny-Based Key Compression Without Pairings. In: Garay, J.A. (ed.) Public-Key Cryptography – PKC 2021. pp. 131–154. Springer International Publishing, Cham (2021)
20. Pereira, G.C.C.F., Doliskani, J., Jao, D.: x-only point addition formula and faster compressed SIKE. Journal of Cryptographic Engineering **11**, 57–69 (2021)
21. Pohlig, S., Hellman, M.: An Improved Algorithm for Computing Logarithms over GF(p) and Its Cryptographic Significance (Corresp.). IEEE Trans. Inf. Theor. **24**(1), 106–110 (2006)
22. Shor, P.: Algorithms for quantum computation: discrete logarithms and factoring. In: Proceedings 35th Annual Symposium on Foundations of Computer Science. pp. 124–134 (1994)
23. Vélu, J.: Isogénies entre courbes elliptiques. C. R. Acad. Sci., Paris, Sér. A **273**, 238–241 (1971)
24. Zanon, G.H.M., Simplicio, M.A., Pereira, G.C.C.F., Doliskani, J., Barreto, P.S.L.M.: Faster Key Compression for Isogeny-Based Cryptosystems. IEEE Transactions on Computers **68**(5), 688–701 (2019)
25. Zhang, X., Lin, D.: Analysis of Optimum Pairing Products at High Security Levels. In: Galbraith, S., Nandi, M. (eds.) Progress in Cryptology - INDOCRYPT 2012. pp. 412–430. Springer Berlin Heidelberg, Berlin, Heidelberg (2012)

A Another approach for pairing evaluation in the case of Bob

In the prior version of the preprint, we utilized the trick mentioned in Section 3.1 to optimize pairing evaluation in the case of Bob. Up to now, this method is not as efficient as the method mentioned in Section 3.2. Besides, applying this method requires more memory. We present the main idea of this method here for interested readers.

To begin with, we claim the below lemma, which gives the relation between $f_{4^{j+1},R}$ and $f_{4^j,R}$.

Lemma 2. *For $j \in \mathbb{N}$, we have*

$$\operatorname{div}\left(f_{4^{j+1},R}\right) = \operatorname{div}\left(f_{4^j,R}^4 \cdot \frac{g_{4^j,4^j}^2 \cdot g_{2\cdot 4^j,2\cdot 4^j}}{v_{2\cdot 4^j}^2 \cdot v_{4^{j+1}}}\right). \tag{11}$$

Proof. Note that

$$\operatorname{div}\left(f_{2\cdot 4^j,R}\right) = \operatorname{div}\left(f_{4^j,R}^2 \cdot \frac{g_{4^j,4^j}}{v_{2\cdot 4^j}}\right). \tag{12}$$

Therefore,

$$\operatorname{div}\left(f_{4^{j+1},R}\right) = \operatorname{div}\left(f_{2\cdot 4^j,R}^2 \cdot \frac{g_{2\cdot 4^j,2\cdot 4^j}}{v_{4^{j+1}}}\right). \tag{13}$$

Combining Equations (12) and (13), we have

$$\begin{aligned}\mathrm{div}\left(f_{4^{j+1},R}\right) &= \mathrm{div}\left(\left(f_{4^j,R}^2 \cdot \tfrac{g_{4^j,4^j}}{v_{2\cdot 4^j}}\right)^2 \cdot \tfrac{g_{2\cdot 4^j,2\cdot 4^j}}{v_{4^{j+1}}}\right)\\ &= \mathrm{div}\left(f_{4^j,R}^4 \cdot \tfrac{g_{4^j,4^j}^2 \cdot g_{2\cdot 4^j,2\cdot 4^j}}{v_{2\cdot 4^j}^2 \cdot v_{4^{j+1}}}\right).\end{aligned}$$

This completes the proof of the lemma. ∎

Use $(x_1^{(j)}, y_1^{(j)})$, $(x_2^{(j)}, y_2^{(j)})$, $(x_4^{(j)}, y_4^{(j)})$ to respectively denote the affine coordinates of $[4^j]R$, $[2\cdot 4^j]R$, and $[4^{j+1}]R$. Let λ_1 and λ_2 be the slopes of $L_{[4^j]R,[4^j]R}$, $L_{[2\cdot 4^j]R,[2\cdot 4^j]R}$:

$$\lambda_1^{(j)} = \frac{(3x_1^{(j)})^2 - 11}{2y_1^{(j)}}, \lambda_2^{(j)} = \frac{(3x_2^{(j)})^2 - 11}{2y_2^{(j)}}.$$

Then all the straight line functions mentioned in Equation (11) can be represented as

$$\begin{aligned}g_{4^j,4^j} &= \lambda_1^{(j)}(x - x_1^{(j)}) - (y - y_1^{(j)}) = \lambda_1^{(j)}(x - x_2^{(j)}) - (y + y_2^{(j)}),\\ g_{2\cdot 4^j,2\cdot 4^j} &= \lambda_2^{(j)}(x - x_2^{(j)}) - (y - y_2^{(j)}),\\ v_{2\cdot 4^j} &= x - x_2^{(j)},\\ v_{4^{j+1}} &= x - x_4^{(j)}.\end{aligned}$$

Now we try to simplify Equation (11). Similar with the trick used in Equation (4),

$$\begin{aligned}y^2 - (y_2^{(j)})^2 &= (x^3 - 11x + 14) - ((x_2^{(j)})^3 - 11x_2^{(j)} + 14)\\ &= (x^3 - (x_2^{(j)})^3) - 11(x - x_2^{(j)})\\ &= (x - x_2^{(j)})(x^2 + x_2^{(j)}x + (x_2^{(j)})^2) - 11(x - x_2^{(j)})\\ &= (x - x_2^{(j)})(x^2 + x_2^{(j)}x + (x_2^{(j)})^2 - 11).\end{aligned} \tag{14}$$

This implies that

$$\begin{aligned}\ell_1^{(j)} &= \frac{g_{4^j,4^j} \cdot g_{2\cdot 4^j,2\cdot 4^j}}{v_{2,4^j}}\\ &= \frac{[\lambda_1^{(j)}(x - x_2^{(j)}) - (y + y_2^{(j)})] \cdot [\lambda_2^{(j)}(x - x_2^{(j)}) - (y - y_2^{(j)})]}{x - x_2^{(j)}}\\ &= \lambda_1^{(j)}\lambda_2^{(j)}(x - x_2^{(j)}) - \lambda_1^{(j)}(y - y_2^{(j)}) - \lambda_2^{(j)}(y + y_2^{(j)}) + \frac{y^2 - (y_2^{(j)})^2}{x - x_2^{(j)}}\\ &= \lambda_1^{(j)}\lambda_2^{(j)}(x - x_2^{(j)}) - \lambda_1^{(j)}(y - y_2^{(j)}) - \lambda_2^{(j)}(y + y_2^{(j)}) + (x^2 + x_2^{(j)}x + (x_2^{(j)})^2 - 11)\\ &= x^2 + (x_2^{(j)} + \lambda_1^{(j)}\lambda_2^{(j)})x - (\lambda_1^{(j)} + \lambda_2^{(j)})y + C_1^{(j)},\end{aligned} \tag{15}$$

where $C_1^{(j)} = (x_2^{(j)} - \lambda_1^{(j)}\lambda_2^{(j)})x_2^{(j)} + (\lambda_1^{(j)} - \lambda_2^{(j)})y_2^{(j)} - 11$. Realizing that

$$\begin{aligned}&\mathrm{div}(g_{4^j,4^j}) = 2([4^j]R) + ([-2\cdot 4^j]R) - 3(\mathcal{O}),\\ &\mathrm{div}(g_{2\cdot 4^j,2\cdot 4^j}) = 2([2\cdot 4^j]R) + ([-4^{j+1}]R) - 3(\mathcal{O}),\\ &\mathrm{div}(v_{2\cdot 4^j}) = ([2\cdot 4^j]R) + ([-2\cdot 4^j]R) - 2(\mathcal{O}),\end{aligned}$$

we can deduce

$$\mathrm{div}\left(\ell_1^{(j)}\right) = 2([4^j]R) + ([2\cdot 4^j]R) + ([-4^{(j+1)}]R) - 4(\mathcal{O}). \tag{16}$$

This implies $\ell_1^{(j)}$ has a zero at $[2\cdot 4^j]R$, but not $[-2\cdot 4^j]R$. Therefore, the function $\ell_1^{(j)}$ takes the form

$$\ell_1^{(j)} = (x + 2x_2^{(j)} + \lambda_1^{(j)}\lambda_2^{(j)})(x - x_2^{(j)}) - (\lambda_1^{(j)} + \lambda_2^{(j)})(y - y_2^{(j)}). \tag{17}$$

Next, consider $\ell_2^{(j)} = \frac{g_{4^j,4^j}\cdot \ell_1^{(j)}}{v_{2\cdot 4^j}}$. Utilizing the same trick above,

$$\begin{aligned}
\ell_2^{(j)} &= \frac{g_{4^j,4^j}\cdot \ell_1}{v_{2\cdot 4^j}}\\
&= \frac{[\lambda_1^{(j)}(x-x_2^{(j)})-(y+y_2^{(j)})]\cdot[(x+2x_2^{(j)}+\lambda_1^{(j)}\lambda_2^{(j)})(x-x_2^{(j)})-(\lambda_1^{(j)}+\lambda_2^{(j)})(y-y_2^{(j)})]}{x-x_2^{(j)}}\\
&= \lambda_1^{(j)}(x + 2x_2^{(j)} + \lambda_1^{(j)}\lambda_2^{(j)})(x - x_2^{(j)}) - \lambda_1^{(j)}(\lambda_1^{(j)} + \lambda_2^{(j)})(y - y_2^{(j)}) - (x+\\
&\quad 2x_2^{(j)} + \lambda_1^{(j)}\lambda_2^{(j)})(y + y_2^{(j)}) + (\lambda_1^{(j)} + \lambda_2^{(j)})\frac{y^2 - (y_2^{(j)})^2}{x - x_2^{(j)}}\\
&= \lambda_1^{(j)}(x + 2x_2^{(j)} + \lambda_1^{(j)}\lambda_2^{(j)})(x - x_2^{(j)}) - \lambda_1^{(j)}(\lambda_1^{(j)} + \lambda_2^{(j)})(y - y_2^{(j)}) - (x+\\
&\quad 2x_2^{(j)} + \lambda_1^{(j)}\lambda_2^{(j)})(y + y_2^{(j)}) + (\lambda_1^{(j)} + \lambda_2^{(j)})\frac{[x^3 - (x_2^{(j)})^3] - 11[x - x_2^{(j)}]}{x - x_2^{(j)}}\\
&= \lambda_1^{(j)}(x + 2x_2^{(j)} + \lambda_1^{(j)}\lambda_2^{(j)})(x - x_2^{(j)}) - \lambda_1^{(j)}(\lambda_1^{(j)} + \lambda_2^{(j)})(y - y_2^{(j)}) - (x+\\
&\quad 2x_2^{(j)} + \lambda_1^{(j)}\lambda_2^{(j)})(y + y_2^{(j)}) + (\lambda_1^{(j)} + \lambda_2^{(j)})(x^2 + x_2^{(j)}x + (x_2^{(j)})^2 - 11)\\
&= (2\lambda_1^{(j)} + \lambda_2^{(j)})x^2 - xy - (2x_2^{(j)} + (\lambda_1^{(j)})^2 + 2\lambda_1^{(j)}\lambda_2^{(j)})y + ((\lambda_1^{(j)})^2\lambda_2^{(j)} -\\
&\quad y_2^{(j)} + 2\lambda_1^{(j)}x_2^{(j)} + \lambda_2^{(j)}x_2^{(j)})x + C_2^{(j)},
\end{aligned}$$

where $C_2^{(j)} = -\lambda_1^{(j)}x_2^{(j)}(2x_2^{(j)} + \lambda_1^{(j)}\lambda_2^{(j)}) + \lambda_1^{(j)}y_2^{(j)}(\lambda_1^{(j)} + \lambda_2^{(j)}) - y_2^{(j)}(2x_2^{(j)} + \lambda_1^{(j)}\lambda_2^{(j)}) + (\lambda_1^{(j)} + \lambda_2^{(j)})((x_2^{(j)})^2 - 11)$.

Defining

$$\begin{aligned}
t_0^{(j)} &= x_4^{(j)},\\
t_1^{(j)} &= 2\lambda_1^{(j)} + \lambda_2^{(j)},\\
t_2^{(j)} &= 2x_2^{(j)} + (\lambda_1^{(j)})^2 + 2\lambda_1^{(j)}\lambda_2^{(j)},\\
t_3^{(j)} &= (\lambda_1^{(j)})^2(\lambda_2^{(j)}) - y_2^{(j)} + t_1^{(j)}(x_2^{(j)} - t_2^{(j)}),\\
t_4^{(j)} &= -(t_1^{(j)}x_4^{(j)} + y_4^{(j)} + t_3^{(j)})(x_4^{(j)} + t_2^{(j)}).
\end{aligned} \tag{18}$$

we have

$$\ell_2^{(j)} = (t_1^{(j)}x - y + t_3^{(j)})\cdot(x + t_2^{(j)}) + t_4^{(j)}, \tag{19}$$

and therefore,

$$\operatorname{div}\left(f_{4^{j+1},R}\right) = \operatorname{div}\left((f_{4^j,R})^4 \cdot \frac{(t_1^{(j)}x - y + t_3^{(j)}) \cdot (x + t_2^{(j)}) + t_4^{(j)}}{x - t_0^{(j)}}\right).$$

The last step is similar to the handling in the last subsection, as we can see in Equation (10).

For each loop of four Miller evaluations, we need to precompute 10 values over $\mathbb{F}_p$, and the last step requires extra 4 values when e_2 is odd and 2 when e_2 is even. In general, there are totally $5e_2-2$ and $5e_2+2$ elements to be precomputed and store with respect to e_2.

In Algorithms 4 and 5 we present pseudocode for evaluating $f_{2^{e_2},R}(U_k)$, where R is either P_2 or Q_2, and U_k $(k = 1, 2)$ are two points of order 2^{e_2} over $E_{-11,14}$. The computational cost for each Miller iteration (except the first loop and the final loop) is $4(15m + 29a) = 60m + 116a$. In comparison, M. Naehrig and J.Renes [18] state the cost $40m + 76a$ for a doubling step, in other words, we save $20m+36a$ per quadrupling step, i.e, $10m+18a$ per doubling step on average.

Algorithm 4 Computation of the Miller evaluations when $R = P_2$ in the case of Bob

Input: $U_0, U_1 \in E_0(\mathbb{F}_{p^2})$

Output: $f_{P_2}(U_0), f_{P_2}(U_1)$

for each $k \in \{0,1\}$ **do**
    $h \leftarrow x_{U_k} - x_{[2]P_2}$,
    $g \leftarrow h$, $g \leftarrow \lambda_{P_2} \cdot g$, $g \leftarrow g - y_{U_k}$, $g \leftarrow g - y_{[2]P_2}$,
    $h \leftarrow h^*$, $g \leftarrow g \cdot h$,
    $f_k \leftarrow g$.
end for
for each $j \in \{0, 1, \cdots, \frac{\lfloor e_2 - 1 \rfloor}{2}\}$ **do**
    for each $k \in \{0,1\}$ **do**
        $h \leftarrow x_{U_k} - t_0^{(j)}$, $h \leftarrow h^*$,
        $temp_1 \leftarrow t_1^{(j)} \cdot x_{U_k}$, $temp_1 \leftarrow temp_1 - y_{U_k}$,$temp_1 \leftarrow temp_1 + t_3^{(j)}$,
        $g \leftarrow x_{U_k} + t_2^{(j)}$, $g \leftarrow g \cdot temp_1$, $g \leftarrow g + t_4^{(j)}$,
        $g \leftarrow g \cdot h$,
        $f_k \leftarrow f_k^2$, $f_k \leftarrow f_k^2$, $f_k \leftarrow f_k \cdot g$.
    end for
end for
if e_2 is even **then**
    for each $k \in \{0,1\}$ **do**
        $g \leftarrow x - t_0^{(\frac{e_2}{2} - 1)}$,
        $f_k \leftarrow f_k^2$, $f_k \leftarrow f_k \cdot g$.
    end for
else
    for each $k \in \{0,1\}$ **do**
        $h \leftarrow x_{U_k} - x_{[2^{e_2-1}]P_2}$,
        $g \leftarrow \lambda_{[2^{e_2-2}]P_2} \cdot h$, $g \leftarrow g - y_{U_k}$, $g \leftarrow g^2$,
        $h \leftarrow h^*$,
        $f_k \leftarrow f_k^2$, $f_k \leftarrow g \cdot f_k$, $f_k \leftarrow f_k^2$, $f_k \leftarrow f_k \cdot h$.
    end for
end if
return f_0, f_1.

Algorithm 5 Computation of the Miller evaluations when $R = Q_2$ in the case of Bob

Input: U_0**,** $U_1 \in E_0(\mathbb{F}_{p^2})$
Output: $f_{P_2}(U_0)$**,** $f_{P_2}(U_1)$

for each $k \in \{0, 1\}$ **do**
    $h \leftarrow x_{U_k} - x_{[2]Q_2}$,
    $g \leftarrow h$, $g \leftarrow \lambda_{Q_2} \cdot g$, $g \leftarrow g - y_{U_k}$, $g \leftarrow g - i \cdot y_{[2]Q_2}$,
    $h \leftarrow h^*$, $g \leftarrow g \cdot h$,
    $f_k \leftarrow g$.
end for
for each $j \in \{0, 1, \cdots, \frac{\lfloor e_2 - 1 \rfloor}{2}\}$ **do**
    for each $k \in \{0, 1\}$ **do**
        $h \leftarrow x_{U_k} - t_0^{(j)}$, $h \leftarrow h^*$,
        $temp_1 \leftarrow i \cdot t_1^{(j)} \cdot x_{U_k}$, $temp_1 \leftarrow temp_1 - y_{U_k}$,$temp_1 \leftarrow temp_1 + i \cdot t_3^{(j)}$,
        $g \leftarrow x_{U_k} + t_2^{(j)}$, $g \leftarrow g \cdot temp_1$, $g \leftarrow g + i \cdot t_4^{(j)}$,
        $g \leftarrow g \cdot h$,
        $f_k \leftarrow f_k^2$, $f_k \leftarrow f_k^2$, $f_k \leftarrow f_k \cdot g$.
    end for
end for
if e_2 is even **then**
    for each $k \in \{0, 1\}$ **do**
        $g \leftarrow x - t_0^{(\frac{e_2}{2} - 1)}$,
        $f_k \leftarrow f_k^2$, $f_k \leftarrow f_k \cdot g$.
    end for
else
    for each $k \in \{0, 1\}$ **do**
        $h \leftarrow x_{U_k} - x_{[2^{e_2 - 1}]Q_2}$,
        $g \leftarrow i \cdot \lambda_{[2^{e_2 - 2}]R} \cdot h$, $g \leftarrow g - y_{U_k}$, $g \leftarrow g^2$,
        $h \leftarrow h^*$,
        $f_k \leftarrow f_k^2$, $f_k \leftarrow g \cdot f_k$, $f_k \leftarrow f_k^2$, $f_k \leftarrow f_k \cdot h$.
    end for
end if
return f_0, f_1.