Faster key compression for isogeny-based cryptosystems

Gustavo H. M. Zanon, Marcos A. Simplicio Jr,
Geovandro C. C. F. Pereira, Javad Doliskani, Paulo S. L. M. Barreto

Abstract—Supersingular isogeny-based cryptography is one of the more recent families of post-quantum proposals. An interesting feature is the comparatively low bandwidth occupation in key agreement protocols, which stems from the possibility of key compression. However, compression and decompression introduce a significant overhead to the overall processing cost despite recent progress. In this paper we address the main processing bottlenecks involved in key compression and decompression, and suggest substantial improvements for each of them. Some of our techniques may have an independent interest for other, more conventional areas of elliptic curve cryptography as well.

Index Terms—Post-quantum cryptography, Supersingular elliptic curves, Public-key compression, Pohlig-Hellman algorithm, Diffie-Hellman key exchange

1 Introduction

In the Supersingular Isogeny Diffie-Hellman (SIDH) protocol [1], the two parties need to exchange a representation of their public keys each consisting of an elliptic curve E together with two points P,Q on E. The curve E is supersingular and is defined over an extension field $\mathbb{F}_{p^2}$ for a prime of the form $p = \ell_A^m \ell_B^n - 1$ where ℓ_A, ℓ_B are small primes, usually equal to 2 and 3, respectively. Originally, this exchange was done using triples of the form (E, x_P, x_Q) where $E : y^2 = x^3 + ax + b$ and x_P, x_Q are the abscissas of P and Q. Two extra bits were also needed to recover the correct y-coordinates. Thus, the public keys are transferred using essentially the four elements $a, b, x_P, x_Q \in \mathbb{F}_{p^2}$ which require $8 \log p$ bits.

A different representation of the SIDH public keys was proposed by [2] that reduced the size to $4 \log p$ bits. The idea was to first represent the curve E using its j-invariant, which is an element of $\mathbb{F}_{p^2}$, rather than the coefficients a, b. This way E is represented using $2 \log p$ bits. The isomorphism class of an elliptic curve is uniquely determined by its j-invariant. Second, since the points P,Q are always in the torsion subgroups $E[\ell_A^m]$ or $E[\ell_B^n]$, they can be represented using elements of $\mathbb{Z}_t \oplus \mathbb{Z}_t$ where either $t = \ell_A^m$ or $t = \ell_B^n$. Since the parameters are such that $\ell_A^m \approx \ell_B^n$, a pair $(t_1, t_2) \in \mathbb{Z}_t \oplus \mathbb{Z}_t$ is represented using $2 \log p$ bits. This reduction of size of the public keys, however, comes with a rather high computational overhead. The conversion between the coefficients a, b of a curve E and its j-invariant is done efficiently; the expensive part is the conversion between elements of $\mathbb{Z}_t \oplus \mathbb{Z}_t$ and the points P,Q. As reported in [2], the compression phase for each party was slower than a full round of uncompressed key exchange by a factor of more than 10 times.

- *Gustavo Zanon and Marcos Simplicio Jr are with Escola Politécnica, University of São Paulo.*
  E-mail: {gzanon,msimplicio}@larc.usp.br
- *Geovandro Pereira and Javad Doliskani are with Institute for Quantum Computing, University of Waterloo.*
  E-mail: {geovandro.pereira,javad.doliskani}@uwaterloo.ca
- *Paulo Barreto is with University of Washington Tacoma.*
  E-mail: pbarreto@uw.edu

Costello *et al.* [3] further improved the key compression scheme by reducing the public key sizes to $3.5 \log p$ bits and decreasing the runtime by almost an order of magnitude. With this improvement, the key compression phase for each party is as fast as one full round of uncompressed key exchange. This certainly motivates the idea of including the compression and decompression phases as default parts of SIDH. However, compared to the currently deployed (classical) schemes, the compression/decompression runtime is unfavourable, requiring further research on speed-up techniques.

Our contributions: We propose new algorithms that further decrease the runtime of SIDH compression and decompression. In contrast to previous works that have deployed "known" algorithms to optimize the performance of key compression, some of the algorithms presented here are new and of broader interest than isogeny-based crypto. A summary of the improvements follows.

- Constructing torsion bases. Assuming the usual parameters $\ell_A = 2, \ell_B = 3$, we improve basis generation for both $E[2^m]$ and $E[3^n]$. To generate a basis for the 2^m-torsion, we propose an algorithm dubbed ***entangled basis*** generation. This algorithm is around 15.9× faster than the usual basis generation presented in [3] and has applications not only in key agreement but also in hash functions based on isogenies [4]. For the 3^n-torsion, we observed that the naive algorithm is more efficient (both in theory and practice) than the explicit 3-descent of [5] used by Costello *et al.* [3].
- In order to further speed up the torsion basis construction during decompression, we introduce the ***shared Elligator*** technique. This technique allows for a 1.5−2.8× faster ternary basis generation compared to the previous plain generation technique from [3]. When the new entangled basis generation is coupled with shared Elligator, the improvements are even more significant attaining a 29.9× speed up. For example, our implementation achieves 0.83M cycles for a 2^m-torsion basis generation, breaking the 1M cycles barrier for the first time for this

purpose. The previous 2-descent technique has a cost of 23.77M cycles according to our experiments.

- Computing discrete logarithms. Inspired by De Feo *et al.*'s ***optimal strategy*** method to compute smooth-degree isogenies [6], we propose an algorithm to compute discrete logarithms in the group μ_{ℓ^n} given an efficient method to compute discrete logarithms in μ_ℓ where ℓ is a small prime, or more generally, an algorithm to compute discrete logarithms in the group $\mu_{(\ell^w)^{n/w}}$ when $w \mid n$, given an efficient method to compute discrete logarithms in μ_{ℓ^w}. For instance, for $w = 6$ our algorithm is 3.9× and 4.6× faster than the algorithm used by [3] for the groups $\mu_{2^{372}}$ and $\mu_{3^{239}}$ respectively.
- We further describe how to compute Pohlig-Hellman in the group μ_{ℓ^n} from an adaptation of the optimal strategy traversal, given an efficient method to compute discrete logarithms in the group μ_{ℓ^w} when $w \nmid n$.
- Pairing computation. We exploit the special shapes of pairs of points generated as entangled bases and the existence of a subfield dismissed by [3] to optimize the Tate pairing. We achieve a speedup of 1.4× for the pairing phase over the algorithms used by [3] for both binary and ternary pairings.
- We evaluate the impact of our improved key compression on the Supersingular Isogeny Key Encapsulation (SIKE) scheme [7], a recently submitted candidate to NIST's call for standardization of post-quantum cryptography.
- Other improvements. We introduce *reverse basis decomposition*, which combined with the previous improvements, allows for further optimizations of compression and decompression. For example each party only needs to compute 4 pairings rather than 5. Also, two expensive cofactor multiplications by 3^n are saved during Bob's compression, and one cofactor multiplication by 3^n is saved during Alice's decompression.

We have implemented the above improvements on top of (the then-latest version of) the Microsoft SIDH library [8]. The library is designed for the specific prime $p = 2^{372}3^{239} - 1$ of size $\log p = 751$ bits.

Our software can be found at https://github.com/geovandro/PQCrypto-SIDH/releases/tag/1.1.0.

1.1 Notations and conventions

For simplicity, we assume that finite field arithmetic is carried out in a base field $\mathbb{F}_p$ and its quadratic extension $\mathbb{F}_{p^2}$ for a prime p of form $p := 2^m \cdot 3^n - 1$ for some $m > 2$ and $n > 1$, so that $p \equiv 3 \pmod 4$. The quadratic extension $\mathbb{F}_{p^2}/\mathbb{F}_p$ is represented as $\mathbb{F}_{p^2} = \mathbb{F}_p[i]/\langle i^2 + 1\rangle$, and arithmetic closely mimics that of the complex numbers.

All curves are represented using the Montgomery model unless otherwise specified. We follow the convention of using subscripts A and B for Alice and Bob, respectively. For example, the secret isogeny ϕ_A is computed by Alice and her public parameters are denoted by the points $P_\mathsf{A}, Q_\mathsf{A}$ and the curve E_A. Similarly, Bob's isogeny is denoted by ϕ_B, and his public parameters are $P_\mathsf{B}, Q_\mathsf{B}, E_\mathsf{B}$.

We denote by **i**, **c**, **m**, **s**, and **a** the costs of inverting, cubing, multiplying, squaring, and adding/subtracting/shifting in $\mathbb{F}_p$, respectively, and by **I**, **C**, **M**, **S**, and **A** the costs of the corresponding operations in $\mathbb{F}_{p^2}$. We disregard the cost of changing a sign (for instance, when handling the conjugate of a field element). The costs of the $\mathbb{F}_{p^2}$ operations relative to the costs of operations in $\mathbb{F}_p$ can be approximated by $1\mathbf{I} = 1\mathbf{i} + 2\mathbf{m} + 2\mathbf{s} + 1\mathbf{a}$, $1\mathbf{C} = 2\mathbf{m} + 1\mathbf{s} + 4\mathbf{a}$, $1\mathbf{M} = 3\mathbf{m} + 5\mathbf{a}$, $1\mathbf{S} = 2\mathbf{m} + 3\mathbf{a}$, and $1\mathbf{A} = 2\mathbf{a}$, by using the finite-field analogues to well-known Viète multiple-angle trigonometric identities [9, Formulas 5.68 and 5.69].

2 Reverse basis decomposition

In this section, we use *reverse basis decomposition* to speed up both Alice's and Bob's key compression by saving one pairing computation. Later in Section 3.1 we show that, when combined with an entangled basis generation, this technique will allow for avoiding two cofactor multiplications by 3^n in Bob's key compression and one in Alice's key decompression. We prove our results from Alice's point of view. The proofs for Bob are similar.

The main previous idea to achieve key compression [2], [3] is the following: instead of transmitting points $\phi_\mathsf{A}(P_\mathsf{B}), \phi_\mathsf{A}(Q_\mathsf{B}) \in E_\mathsf{A}[3^n]$, which are represented by two abscissas in $\mathbb{F}_{p^2}$ and consume $4\log p$ bits, Alice computes a canonical basis $R_1, R_2 \in E_\mathsf{A}[3^n]$ and expresses the expanded public key in that basis as $\phi_\mathsf{A}(P_\mathsf{B}) = a_0R_1 + b_0R_2$ and $\phi_\mathsf{A}(Q_\mathsf{B}) = a_1R_1 + b_1R_2$. In matrix notation,

$$\begin{bmatrix}\phi_\mathsf{A}(P_\mathsf{B})\\ \phi_\mathsf{A}(Q_\mathsf{B})\end{bmatrix} = \begin{bmatrix}a_0 & b_0\\ a_1 & b_1\end{bmatrix}\begin{bmatrix}R_1\\ R_2\end{bmatrix}. \tag{1}$$

This representation consists of four smaller integers $(a_0, b_0, a_1, b_1) \in (\mathbb{Z}/3^n\mathbb{Z})^4$ of total size $2\log p$ bits as suggested in [2]. This was improved in [3] by transmitting only the triple $(a_0^{-1}b_0, a_0^{-1}a_1, a_0^{-1}b_1) \in (\mathbb{Z}/3^n\mathbb{Z})^3$ or $(b_0^{-1}a_0, b_0^{-1}a_1, b_0^{-1}b_1) \in (\mathbb{Z}/3^n\mathbb{Z})^3$ depending on whether a_0 or b_0 is invertible. Therefore, only $(3/2)\log p$, plus one bit indicating the invertibility of a_0 or b_0 modulo 3^n, is needed. In the above mentioned techniques, the coefficients a_0, b_0, a_1, b_1 can be computed using five Tate pairings given by

$$\begin{aligned}
g_0 &= \mathsf{e}_{3^n}(R_1, R_2)\\
g_1 &= \mathsf{e}_{3^n}(R_1, \phi_\mathsf{A}(P_\mathsf{B})) = \mathsf{e}_{3^n}(R_1, a_0R_1 + b_0R_2) = g_0^{b_0}\\
g_2 &= \mathsf{e}_{3^n}(R_1, \phi_\mathsf{A}(Q_\mathsf{B})) = \mathsf{e}_{3^n}(R_1, a_1R_1 + b_1R_2) = g_0^{b_1}\\
g_3 &= \mathsf{e}_{3^n}(R_2, \phi_\mathsf{A}(P_\mathsf{B})) = \mathsf{e}_{3^n}(R_2, a_0R_1 + b_0R_2) = g_0^{-a_0}\\
g_4 &= \mathsf{e}_{3^n}(R_2, \phi_\mathsf{A}(Q_\mathsf{B})) = \mathsf{e}_{3^n}(R_2, a_1R_1 + b_1R_2) = g_0^{-a_1}.
\end{aligned} \tag{2}$$

From this, Alice can recover a_0, b_0, a_1, and b_1 by solving discrete logs in a multiplicative subgroup of smooth order 3^n using the Pohlig-Hellman algorithm.

Now since $\phi_\mathsf{A}(P_\mathsf{B})$ and $\phi_\mathsf{A}(Q_\mathsf{B})$ also form a basis for $E_\mathsf{A}[3^n]$, we see that the coefficient matrix in (1) is invertible modulo 3^n. So, we can write

$$\begin{bmatrix}R_1\\ R_2\end{bmatrix} = \begin{bmatrix}c_0 & d_0\\ c_1 & d_1\end{bmatrix}\begin{bmatrix}\phi_\mathsf{A}(P_\mathsf{B})\\ \phi_\mathsf{A}(Q_\mathsf{B})\end{bmatrix} \tag{3}$$

by inverting the matrix in (1). Changing the roles of the bases $\{R_1, R_2\}$ and $\{\phi_A(P_B), \phi_A(Q_B)\}$ in (2) we get

$$\begin{aligned} h_0 &= \mathrm{e}_{3^n}(\phi_A(P_B), \phi_A(Q_B)) \\ h_1 &= \mathrm{e}_{3^n}(\phi_A(P_B), R_1) \\ &= \mathrm{e}_{3^n}(\phi_A(P_B), c_0\phi_A(P_B) + d_0\phi_A(Q_B)) = h_0^{d_0} \\ h_2 &= \mathrm{e}_{3^n}(\phi_A(P_B), R_2) \\ &= \mathrm{e}_{3^n}(\phi_A(P_B), c_1\phi_A(P_B) + d_1\phi_A(Q_B)) = h_0^{d_1} \\ h_3 &= \mathrm{e}_{3^n}(\phi_A(Q_B), R_1) \\ &= \mathrm{e}_{3^n}(\phi_A(Q_B), c_0\phi_A(P_B) + d_0\phi_A(Q_B)) = h_0^{-c_0} \\ h_4 &= \mathrm{e}_{3^n}(\phi_A(Q_B), R_2) \\ &= \mathrm{e}_{3^n}(\phi_A(Q_B), c_1\phi_A(P_B) + d_1\phi_A(Q_B)) = h_0^{-c_1}. \end{aligned} \tag{4}$$

The first pairing in (4) is computed as $h_0 = \mathrm{e}_{3^n}(P_B, \hat{\phi}_A \circ \phi_A(Q_B)) = \mathrm{e}_{3^n}(P_B, [\deg \phi_A]Q_B) = \mathrm{e}_{3^n}(P_B, Q_B)^{2^m}$, which only depends on the public parameters P_B, Q_B and m. Therefore, it can be computed once and for all and be included in the public parameters. In particular, only the pairings h_1, h_2, h_3 and h_4 need to be computed at runtime. The discrete logs are computed as before using Pohlig-Hellman, yielding $c_0 = -\log_{h_0} h_3, d_0 = \log_{h_0} h_1, c_1 = -\log_{h_0} h_4$ and $d_1 = \log_{h_0} h_2$. Next, Alice inverts the computed coefficients matrix of (3) to obtain the coefficient matrix of (1). Explicitly,

$$\begin{bmatrix} a_0 & b_0 \\ a_1 & b_1 \end{bmatrix} = \frac{1}{D}\begin{bmatrix} d_1 & -d_0 \\ -c_1 & c_0 \end{bmatrix}$$

where $D = c_0 d_1 - c_1 d_0$. Notice that the extra inversion of D^{-1} does not need to be carried out when using the technique in [3]. More precisely, since at least one of d_0 and d_1, say d_1, is invertible modulo 3^n, Alice transmits the tuple

$$\begin{aligned} &(a_0^{-1}b_0, a_0^{-1}a_1, a_0^{-1}b_1) \\ &= (-d_1^{-1}DD^{-1}d_0, -d_1^{-1}DD^{-1}c_1, d_1^{-1}DD^{-1}c_0) \\ &= (-d_1^{-1}d_0, -d_1^{-1}c_1, d_1^{-1}c_0) \end{aligned}$$

which is independent of D.

3 Entangled basis generation

We now introduce a technique to create a complete basis of the 2^m-torsion from a single (albeit specific) point of order 2^m. In other words, the cost involved is essentially that of creating a generator for a single subgroup of order 2^m in $E[2^m]$: a generator for the linearly independent subgroup becomes immediately available almost for free. Consequently, the linear independence test consisting of two scalar multiplications by 2^{m-1} can be avoided. This is akin to distortion maps even though none is typically available for the curves involved in SIDH. We call the resulting bases "entangled" by analogy with the quantum phenomenon whereby the properties of one entity are entirely determined by the properties of another entity despite their separation[1].

In order to build an entangled basis $\langle P, Q\rangle = E[2^m]$ for $E : y^2 = x^3 + Ax^2 + x$, we somewhat "subvert" the original Elligator 2 formulas [10] under a different motivation than encoding points to random strings: obtaining two linearly independent points on E at once. Herein the value $t := u_0 r$, for $u_0 \in \mathbb{F}_{p^2} \setminus \mathbb{F}_p$ and $r \in \mathbb{F}_p^*$ s.t. $u := u_0^2 \in \mathbb{F}_{p^2} \setminus \mathbb{F}_p$, will be a square rather than a non-square. The new construction is proved in Theorem 1.

1. We stress, however, that here the naming is purely analogous: there is no quantum process involved in the construction.

Remark. As in [10] we assume that $A \neq 0$ in the Montgomery model. The case $A = 0$ corresponds to the curve $E : y^2 = x^3 + x$ which is used as the initial curve in most of the implementations. This does not pose a problem in our setting. This is because first, runtime basis generation does not happen for the initial curve E, and second, the probability that E is encountered in the middle of the key exchange is negligible. So the parties can avoid this issue by checking the j-invariant of their public keys.

Theorem 1. *Given a Montgomery supersingular elliptic curve $E_A/\mathbb{F}_{p^2} : y^2 = x(x^2 + Ax + 1)$ where $p = 2^m \cdot 3^n - 1$, $\#E_A(\mathbb{F}_{p^2}) = (p+1)^2$, and $A \neq 0$, let $t \in \mathbb{F}_{p^2}$ be a field element such that $t^2 \in \mathbb{F}_{p^2} \setminus \mathbb{F}_p$, and let $x_1 := -A/(1+t^2)$ be a quadratic non-residue that defines the abscissa of a point $P_1 \in E_A(\mathbb{F}_{p^2})$. Then $x_2 := -x_1 - A$ defines the abscissa of another point $P_2 \in E_A(\mathbb{F}_{p^2})$ such that $\langle [h]P_1, [h]P_2\rangle = E_A[2^m]$, where $h := 3^n$ is the cofactor of the 2^m-torsion group.*

Proof. Since $x_2 = t^2 x_1$, both abscissas are quadratic non-residues and by [11, Chapter 1 (§4), Theorem 4.1] the two points $P_1 = (x_1, y_1)$, $P_2 = (t^2 x_1, t y_1)$, with $x_1 + t^2 x_1 + A = 0$, are not in $[2]E_A$. So the points $[h]P_1$ and $[h]P_2$ are full 2^m-torsion points. To prove that $h \cdot P_1, h \cdot P_2$ generate $E_A[2^m]$ we have to prove that $[h \cdot 2^{m-1}](P_1 - P_2) \neq 0$, or equivalently that $(u, v) = P_1 + (-P_2) \notin [2]E_A$.

By the addition law [12, Algorithm 2.3] on E_A we get

$$\begin{aligned} \lambda &= \frac{y_2 - y_1}{x_2 - x_1} = \frac{-t y_1 - y_1}{t^2 x_1 - x_1} = \frac{-(t+1)y_1}{(t^2-1)x_1} = \frac{-y_1}{(t-1)x_1}, \\ \mu &= \frac{y_1 x_2 - y_2 x_1}{x_2 - x_1} = \frac{t(t+1)y_1 x_1}{(t+1)(t-1)x_1} = -\lambda t x_1, \\ u &= \lambda^2 - A - x_1 - x_2 = \lambda^2, \\ v &= -\lambda u - \mu = -\lambda u - (-\lambda t x_1) = -\lambda(u - t x_1). \end{aligned}$$

From the above equalities we see that $v^2 = \lambda^2(u - t x_1)^2 = u(u^2 + Au + 1)$ and hence $u^2 + Au + 1 = (u - t x_1)^2$. Let $w := u - t x_1 = \sqrt{u^2 + Au + 1}$. Then $1 - (u - w)^2 = 1 - t^2 x_1^2 = x_1^2 + A x_1 + 1$, which is a quadratic non-residue because x_1 is itself a quadratic non-residue while their product is obviously a square, $x_1(x_1^2 + A x_1 + 1) = y_1^2$. A straightforward calculation shows that $(1-(u+w)^2)(1-(u-w)^2) = u^2(A^2 - 4)$. But $A^2 - 4$ is a quadratic residue since E_A has the full 2-torsion over $\mathbb{F}_{p^2}$. Therefore, both $(u \pm w)^2 - 1$ have the same quadratic residuosity, that is, they are both quadratic non-residues by the above.

Now[2] assume by contradiction that $P_1 - P_2 \in [2]E_A$, i.e. there is a point $(x, y) \in E_A(\mathbb{F}_{p^2})$ such that $[2](x, y) = (u, v)$. From the doubling formula on E_A we get

$$u = \frac{(x^2 - 1)^2}{4x(x^2 + Ax + 1)}.$$

From this we get a quartic equation $(x^2-1)^2 - 4ux(x^2 + Ax + 1) = 0$. Since $x \neq 0$, we can divide both sides by x^2 and rearrange some terms to get

$$\left(x + \frac{1}{x}\right)^2 - 4u\left(x + \frac{1}{x}\right) - 4(Au + 1) = 0.$$

From this we obtain

$$x + \frac{1}{x} = \frac{4u \pm \sqrt{16(u^2 + Au + 1)}}{2} = \frac{4u \pm 4w}{2} = 2(u \pm w).$$

2. This part closely follows the idea behind [11, Chapter 1 (§4), Theorem 4.1].

In turn, from this we get $x^2 - 2(u \pm w)x + 1 = 0$. Again since $x \in \mathbb{F}_{p^2}$, the discriminant $4(u \pm w)^2 - 4$, and hence at least one of the $(u \pm w)^2 - 1$ must be a quadratic residue. But this contradicts the earlier observation that $(u \pm w)^2 - 1$ are both quadratic non-residues. Therefore $P_1 - P_2 \notin [2]E$, yielding the claim that $\langle [h]P_1, [h]P_2 \rangle = E_A[2^m]$. □

In practice, one can efficiently implement entangled basis generation as follows. Let $u_0 \in \mathbb{F}_{p^2} \backslash \mathbb{F}_p$ such that $u := u_0^2 \in \mathbb{F}_{p^2} \backslash \mathbb{F}_p$, e.g. $u_0 = 1 + i$ and $u = 2i$. Define two separate tables of pairs (r, v) with $v := 1/(1 + ur^2)$:

- table T_1 contains pairs (r, v) in which v is quadratic non-residue,
- table T_2 contains pairs (r, v) in which v is quadratic residue.

Performing one quadraticity test on A, only once per curve, and restricting table lookup to the table of opposite quadraticity ensures that $x := -Av$ is a non-square. Repeating quadraticity tests to ensure that a corresponding y exists, and completing one square root extraction in $\mathbb{F}_{p^2}$ to obtain y, one gets 2 points whose orders are multiples of 2^m at once. This is detailed in Algorithm 3.1.

Let us compare the number of operations required by the entangled basis algorithm with the plain basis generation algorithm used in Costello *et al.* [3].

Entangled basis: testing the quadraticity of A takes $(m + n + 1)\mathbf{s} + n\mathbf{m}$. The main loop runs twice on average at a cost $2(m + n + 1)\mathbf{s} + (2n + 22)\mathbf{m}$. The last stage is to complete a square root and costs $(m + n - 1)\mathbf{s} + (n + 1)\mathbf{m} + 1\mathbf{i}$. The total cost of the algorithm is then

$$(4m + 4n + 2)\mathbf{s} + (4n + 23)\mathbf{m} + 1\mathbf{i}.$$

Plain basis: To get the abscissa of a point on the curve takes $(2n+22)\mathbf{m}+2(m+n+1)\mathbf{s}$. Clearing the cofactor 3^n requires n point triplings at a cost $32n\mathbf{m}$. We also need to compute $m - 1$ point doublings for linear independence check that is required in the next steps. So obtaining the first basis point costs $(34n + 16m + 6)\mathbf{m} + 2(m + n + 1)\mathbf{s}$. The second basis point is obtained exactly the same way, except we also need a linear independence check. This is done in loop that runs twice on average. The expected cost of obtaining the second point is then twice the cost of obtaining the first point including the the $m - 1$ doublings step. The last stage of the algorithm is to recover the y coordinates of the points which costs $(4m + 4n)\mathbf{s} + (4n + 36)\mathbf{m} + 2\mathbf{i}$. Adding all these, the total cost of the algorithm is

$$(10m + 10n + 6)\mathbf{s} + (48m + 106n + 54)\mathbf{m} + 2\mathbf{i}.$$

For the values $m = 372$ and $n = 239$, and assuming $\mathbf{s} = 0.8\mathbf{m}$ and $\mathbf{i} = 100\mathbf{m}$, we get the performance ratio of 15.92.

3.1 Avoiding cofactor multiplication

Combining reverse basis decomposition and entangled basis generation enables us to further avoid two scalar multiplications by the large cofactor 3^n during Bob's public key compression, and one during Alice's decompression. First notice that Algorithm 3.1 already incorporates the mentioned optimization, i.e. the output points S_1 and S_2 satisfy $(R_1, R_2) := ([3^n]S_1, [3^n]S_2)$ such that $\langle R_1, R_2 \rangle = E[2^m]$. This is only possible because in reverse basis decomposition the Tate pairings

Algorithm 3.1 Entangled basis generation for $E[2^m](\mathbb{F}_{p^2})$: $y^2 = x^3 + Ax^2 + x$

INPUT: $A = a + bi \in \mathbb{F}_{p^2}$; $u_0 \in \mathbb{F}_{p^2} : u = u_0^2 \in \mathbb{F}_{p^2} \backslash \mathbb{F}_p$; tables T_1, T_2 of pairs $(r \in \mathbb{F}_p, v = 1/(1 + ur^2) \in \mathbb{F}_{p^2})$ of QNR and QR.

OUTPUT: $\{S_1, S_2\}$ such that $\langle [3^n]S_1, [3^n]S_2 \rangle = E[2^m](\mathbb{F}_{p^2})$.

$z \leftarrow a^2 + b^2$, $s \leftarrow z^{(p+1)/4}$
$T \leftarrow (s^2 \stackrel{?}{=} z)\ T_1 : T_2$ // select proper table by testing quadraticity of A
repeat
  lookup next entry (r, v) from T
  $x \leftarrow -A \cdot v$ // NB: x nonsquare
  $t \leftarrow x \cdot (x^2 + A \cdot x + 1)$ // test quadraticity of $t = c + di$
  $z \leftarrow c^2 + d^2$, $s \leftarrow z^{(p+1)/4}$
until $s^2 = z$ // compute $y \leftarrow \sqrt{x^3 + A \cdot x^2 + x}$
$z \leftarrow (c + s)/2$, $\alpha \leftarrow z^{(p+1)/4}$, $\beta \leftarrow d \cdot (2\alpha)^{-1}$
$y \leftarrow (\alpha^2 \stackrel{?}{=} z)\ \alpha + \beta i : -\beta - \alpha i$ // compute basis
return $S_1 \leftarrow (x, y)$, $S_2 \leftarrow (ur^2 x, u_0 r y)$

h_i take the points S_i in their second argument which does not need to be necessarily cofactor-reduced. In this case, for $R_1 = c_0\phi_B(P_A) + d_0\phi_B(Q_A)$ and $R_2 = c_1\phi_B(P_A) + d_1\phi_B(Q_A)$, the respective pairing computations are

$$\begin{aligned}
k_0 &= \mathrm{e}_{2^m}(\phi_B(P_A), \phi_B(Q_A)) \\
k_1 &= \mathrm{e}_{2^m}(\phi_B(P_A), S_1) \\
&= \mathrm{e}_{2^m}(\phi_B(P_A), [3^{-n}]R_1) = k_0^{3^{-n}d_0} \\
k_2 &= \mathrm{e}_{2^m}(\phi_B(P_A), S_2) \\
&= \mathrm{e}_{2^m}(\phi_B(P_A), [3^{-n}]R_2) = k_0^{3^{-n}d_1} \\
k_3 &= \mathrm{e}_{2^m}(\phi_B(Q_A), S_1) \\
&= \mathrm{e}_{2^m}(\phi_B(Q_A), [3^{-n}]R_1) = k_0^{-3^{-n}c_0} \\
k_4 &= \mathrm{e}_{2^m}(\phi_B(Q_A), S_2) \\
&= \mathrm{e}_{2^m}(\phi_B(Q_A), [3^{-n}]R_2) = h_0^{-3^{-n}c_1}.
\end{aligned}$$

Thus, the discrete logarithms are the desired ones up to a factor 3^{-n}, and given by $\hat{c}_0 = -\log_{k_0} k_3 = 3^{-n}c_0$, $\hat{d}_0 = \log_{k_0} k_1 = 3^{-n}d_0$, $\hat{c}_1 = -\log_{k_0} k_4 = 3^{-n}c_1$, and $\hat{d}_1 = \log_{k_0} k_2 = 3^{-n}d_1$. Notice that $3^{-n} \bmod 2^m$ must be odd which implies that $\hat{c}_0$ or $\hat{d}_0$ is invertible if and only if c_0 or d_0 is invertible. Similar to the situation in Section 2, when using the compression with only 3 coefficients as in [3] Bob transmits exactly the original coefficients: assuming $\hat{c}_0$ is invertible, then

$$\begin{aligned}
&(\hat{c}_0^{-1}\hat{d}_0, \hat{c}_0^{-1}\hat{c}_1, \hat{c}_0^{-1}\hat{d}_1) \\
&= (c_0^{-1}3^n3^{-n}d_0, c_0^{-1}3^n3^{-n}c_1, c_0^{-1}3^n3^{-n}d_1) \\
&= (c_0^{-1}d_0, c_0^{-1}c_1, c_0^{-1}d_1)
\end{aligned}$$

The derivation when d_0 is invertible is analogous.

To decompress Bob's public key, Alice needs to perform a single cofactor multiplication by 3^n as follows. Assume that a_0 is invertible modulo 2^m so that Alice receives the triple $(a_0^{-1}b_0, a_0^{-1}a_1, a_0^{-1}b_1)$. She needs to compute the kernel $\ker(\phi_{AB}) = \langle \phi_B(P_A) + sk_A \cdot \phi_B(Q_A) \rangle$ which can be written as $\langle a_0R_1 + b_0R_2 + sk_A \cdot (a_1R_1 + b_1R_2) \rangle = \langle (a_0 + sk_Aa_1)R_1 + (b_0 + sk_Ab_1)R_2 \rangle$ As noted in [3], one computes $\ker(\phi_{AB})$ as $a_0^{-1}\ker(\phi_{AB}) = \langle (1+sk_Aa_0^{-1}a_1)R_1+(a_0^{-1}b_0+sk_Aa_0^{-1}b_1)R_2 \rangle$, which can be done with one scalar multiplication and one point addition by writing $\ker(\phi_{AB}) = \langle R_1 + (1 + sk_Aa_0^{-1}a_1)^{-1}(a_0^{-1}b_0 + sk_Aa_0^{-1}b_1)R_2 \rangle$. Now if Alice uses Algorithm 3.1, she obtains an entangled basis $\{S_1, S_2\}$ such that $(R_1, R_2) = ([3^n]S_1, [3^n]S_2)$.

She can then compute $T = \langle S_1 + (1 + sk_A a_0^{-1} a_1)^{-1}(a_0^{-1} b_0 + sk_A a_0^{-1} b_1) S_2$ first and then recover the correct kernel $\ker(\phi_{AB}) = \langle [3^n]T \rangle$ by performing one cofactor scalar multiplication.

4 On basis generation for $E[3^n]$

The entangled basis approach introduced in Section 3 does not immediately generalize to the ternary case. There is no clear way to simultaneously choose two linearly independent points. As a consequence, to generate bases for $E[3^n]$ we adopted the naïve approach of randomly picking candidate points and testing them for the correct order and linear independence.

Costello *et al.* suggest the use of a 3-descent approach based on a result by Schaefer and Stoll [5], and claim significant performance gains. However, we were unable to reproduce and thus verify their claims. On the contrary, the naïve method is observed to be always faster than 3-descent, with a cost ratio $\mathscr{C}_{\text{naïve}}/\mathscr{C}_{\text{3-descent}} \approx 0.89$ that runs against their claim. The following detailed analysis appears to corroborate this observed cost ratio.

In Costello *et al.*'s 3-descent approach, the claimed gains only apply to *testing* whether a point is in $E \setminus [3]E$. We note that the cost of generating candidates for testing has to be taken into account as well.

Thus, on the one hand, naïve testing involves:

- one Elligator construction at a cost **L** per attempt,
- m doublings at a cost **D** each per attempt,
- $n-1$ triplings at a cost **T** each per attempt,
- 9/8 attempts on average at a cost **P** each to get a point of right order,
- 4/3 point constructions and checks at a cost **C** each on average to get a second, linearly independent point.

Hence the naïve cost to get (the x-coordinates of) the base points is $(1+4/3)(9/8)\mathbf{P} + (4/3)\mathbf{C}$, complemented by two curve equation solvings at a cost **E** each to complete the point coordinates, or $(1+4/3)(9/8)\mathbf{P} + (4/3)\mathbf{C} + 2\mathbf{E}$ overall.

Estimating $\mathbf{L} = (0.8m + 1.8n + 9.8)\mathbf{m} + 20\mathbf{a}$, $\mathbf{D} = 13\mathbf{m} + 29\mathbf{a}$, $\mathbf{T} = 27\mathbf{m} + 61\mathbf{a}$, $\mathbf{C} = 2(3\mathbf{m} + 5\mathbf{a})$, $\mathbf{E} = 1\mathbf{i} + (1.6m + 3.6n + 27.6)\mathbf{m} + 46\mathbf{a}$, and noticing that $\mathbf{P} = \mathbf{L} + m\mathbf{D} + (n-1)\mathbf{T}$, we conclude that the naïve cost is $\mathscr{C}_{\text{naïve}} \approx 2\mathbf{i} + (39.425m + 82.8n + 18.05)\mathbf{m} + (76.125m + 160.125n - 3.708)\mathbf{a}$.

On the other hand, the 3-descent method initially involves:

- one Elligator construction at a cost **L**,
- m doublings at a cost **D**,
- $n-1$ triplings at a cost **T**,
- one curve equation solving at a cost **E**,
- one filter function construction at a cost **F** to get a point P_3 of order 3 and possibly the first base point.

Note that a more expensive doubling formula at a cost $\mathbf{D}' = 16\mathbf{m} + 34\mathbf{a}$ (instead of $\mathbf{D} = 13\mathbf{m} + 29\mathbf{a}$) was employed by [3] which takes the curve coeficients in projective form. This projective formula is useful for computing projective 2^m-isogenies but we note that it is not necessary in the context of basis generation and one can simply stick to the usual more efficient Montgomery doubling [13]. We consider the faster version in our estimates.

The 3-descent method will require (with probability 1/9) an extra, filtered point construction at a cost **Z** to get the first base point, plus 4/3 filtered point constructions and checks on average (since the probability of check success is 3/4) to get the second base point, and finally two curve equation solvings.

Estimating $\mathbf{F} = 1\mathbf{i} + 12.6\mathbf{m} + 29\mathbf{a}$, $\mathbf{Z} = 3\mathbf{i} + (17.8m + 37.8n + 39.3)\mathbf{m} + (29m + 61n + 57.5)\mathbf{a}$, and keeping the same remaining estimates as before, we conclude that the 3-descent cost is $\mathscr{C}_{\text{3-descent}} := (3(4/3) + 3(1/9) + 4)\mathbf{i} + ((17.8m + 37.8n + 45.3)(4/3) + (17.8m + 37.8n + 39.3)(1/9) + 31.2m + 39.6n + 78.2)\mathbf{m} + ((29m + 61n + 67.5)(4/3) + (29m + 61n + 57.5)(1/9) + 29m + 61n + 126)\mathbf{a}$.

This yields a cost ratio $\mathscr{C}_{\text{naïve}}/\mathscr{C}_{\text{3-descent}} \approx 0.89$, which is what we observe experimentally. This runs against the claim in [3] on "the significant speed advantage that is obtained by the use of the result of Schaefer and Stoll [i.e. the 3-descent method]:" the naïve method is observed to be always faster than 3-descent.

4.1 Shared Elligator and faster decompression

Although shown to be faster than 3-descent in the previous section, the naïve approach for basis generation of $E[3^n]$ incurs a substantial cost that seems unavoidable at key compression. Interestingly, the knowledge gained in the process (in the form of the actual counters r that specify the points in the Elligator 2 construction) could be then shared between Alice and Bob, speeding up the latter's work at key decompression. For a very modest increase in Alice's public key size (for instance, a single extra byte for each of the two basis points would provide space that is only exceeded with probability well below 2^{-400}), Bob's $E[3^n]$ basis generation would get about 32% faster, and his full decompression of Alice's key would become about 24% faster.

As discussed in Section 4, naïve $E[3^n]$ basis generation requires 9/8 construction attempts on average to get each point of right order individually, and the second point construction and testing has to be repeated 4/3 times on average to ensure the pair constitutes a basis. This means that, if the cost of generating each point candidate is **P**, the actual expected cost to obtain a basis is roughly $(21/8)\mathbf{P} + (4/3)\mathbf{C} + 2\mathbf{E}$. In case Alice shares the actual Elligator 2 counters that lead to a basis, the cost for Bob would become $2\mathbf{P} + \mathbf{C} + 2\mathbf{E}$ since both the Elligator computation and the linear independence test would become deterministic. This represents a speed up of about 1.47× (or 32% faster) on basis generation alone and 1.4× for the overall decompression compared to the 3-descent method suggested in [3].

Remark. Notice that if Bob trusts Alice, i.e., the Alice's public key is authenticated, then $n-1$ triplings can be avoided since linear independence test is not necessary (the Elligator counters are supposed to give a genuine basis). In this case the improvement is more drastic, the cost to generate a basis for $E[3^n]$ becomes $2\mathbf{P}' + 2\mathbf{E}$ where $\mathbf{P}' = \mathbf{L} + m\mathbf{D}$ which represents a speedup of 2.86× over the previous 3-descent approach. In this case, the task of decompression would get about 2× faster.

Remark. Note that the public key encryption SIKE.PKE and the key encapsulation mechanism SIKE.KEM use slightly different settings with respect to processing the uncompressed public key. However, this does not impact the compression and decompression process. More precisely, the public key is always compressed in the key generation function and decompressed before the key encapsulation function. Also, since the counter in the shared Elligator is computed from public information, it does not impact the security of the above two protocols.

Moreover, the Elligator 2 counters tend to be very small. Specifically, if the probability that a point candidate is rejected at a certain attempt is $1/t$, then the expected number of attempts is $t/(t-1)$ and the probability that the number of attempts exceeds N is the probability of failing N times, i.e., t^{-N}. For the first point, $t = 9$ (only one of the 9 points of 3-torsion causes rejection when computing $[3^{n-1}]R$), while for the second point $t = 3$ (because the probability of a candidate being accepted at a certain attempt is $(8/9)(3/4) = 2/3$, and hence the probability of rejection is $1/3$). Therefore, for the first point the expected number of attempts is $9/8$ and the probability that the number of attempts exceeds N is 9^{-N}, while for the second point the expected number of attempts is $3/2$ and the probability that the number of attempts exceeds N is 3^{-N}. If we use one byte for each counter (interpreted as an index to a table of squares or a table of non-squares), the probabilities that the available counter space ($N = 255$) is exceeded are a whooping $2^{-811.5}$ and $2^{-404.2}$ respectively. This means that Alice needs to transmit just two more bytes with her key.

This modest increase in Alice's key size is compensated by worthwhile speed-ups for Bob. For $m = 372$ and $n = 239$, Alice's extended key size becomes 330 bytes rather than 328 (see Table 3 in [3]), matching Bob's plain key size. While the 3^n-torsion generation time for Alice is that indicated on Table 5, Bob's time decreases to about 13.63 Mcycles yielding a ratio 1.47 rather than 1.15. Furthermore, Alice's compression ratio to the previous state of the art stays the same as well, but Bob's ratio improves from 1.09 to 1.30.

4.2 The Shared Elligator on Entangled Bases

Interestingly, the binary entangled basis generation technique not only allows for a shared Elligator decompression but in addition requires less extra bandwidth compared to the ternary case, i.e., 1 instead of 2 extra bytes. In addition, the resulting 50% faster decompression is considerably more noticeable compared to the ternary counterpart. The reason for this bigger improvement is that the Step 1 of the entangled basis Algorithm 3.1 consisting of a relatively expensive quadraticity test can be avoided. In this case, Bob who is responsible for compressing his key in the 2^m-torsion can transmit the quadraticity of A through a single bit b. Since Bob's public key size is not an exact multiple of 32 bytes for a 751-bit prime, a few unused bits in the byte-oriented representation of the key are available and one of those can be used for transmitting this information about A.

The other information Bob can share with Alice is the counter r computed in Step 4 that leads to a point on the curve for the first candidate. This can be done with one single extra byte. The probability of exceeding one byte is the probability of the first abscissa failing to be on the curve $N = 255$ times, i.e., $2^{-N} = 2^{-255}$. Moreover, because the second point on an entangled basis is determined by the first point, no second extra byte is necessary in this case. Algorithm 4.1 describes the entangled basis generation with shared Elligator for decompression.

Entangled basis + shared Elligator: The resulting operation count for the entangled basis generation coupled with shared Elligator involves only one single iteration of the loop in Step 3 of Algorithm 3.1 amounting to $(m+n+1)\mathbf{s}+(n+9)\mathbf{m}$ and the final part (Steps 8 to 10) amounting to $(m+n-1)\mathbf{s}+(n+6)\mathbf{m}+\mathbf{i}$. Adding up the above costs, the new basis generation will cost

$$2(m+n)\mathbf{s}+(2n+15)\mathbf{m}+\mathbf{i}.$$

Assuming $\mathbf{i} = 100\mathbf{m}$ and $\mathbf{s} = 0.8\mathbf{m}$, this represents a speed up of 29.9× faster basis generation compared to the plain basis generation described in Section 3.1.

Algorithm 4.1 Entangled basis generation coupled with shared Elligator for $E[2^m](\mathbb{F}_{p^2}) : y^2 = x^3 + Ax^2 + x$

INPUT: $A = a + bi \in \mathbb{F}_{p^2}$; $u_0 \in \mathbb{F}_{p^2} : u = u_0^2 \in \mathbb{F}_{p^2} \backslash \mathbb{F}_p$; tables T_1, T_2 of pairs $(r \in \mathbb{F}_p, v = 1/(1+ur^2) \in \mathbb{F}_{p^2})$ of QNR and QR. A bit bit for A's quadraticity and $r \in \mathbb{F}_p$.
OUTPUT: $\{S_1, S_2\}$ such that $\langle [3^n]S_1, [3^n]S_2 \rangle = E[2^m](\mathbb{F}_{p^2})$.

1: $T \leftarrow (bit \stackrel{?}{=} 1)\ T_1 : T_2$ // select proper table according to A's quadraticity
2: $x \leftarrow -A \cdot T[r]$ // NB: x nonsquare
3: $t \leftarrow x \cdot (x^2 + A \cdot x + 1)$ // test quadraticity of $t = c + di$
4: $z \leftarrow c^2 + d^2$, $s \leftarrow z^{(p+1)/4}$
5: **if** $s^2 \neq z$ **then**
6: Abort // incorrect parameters (b, r) received
7: **end if** // compute $y \leftarrow \sqrt{x^3 + A \cdot x^2 + x}$
8: $z \leftarrow (c+s)/2$, $\alpha \leftarrow z^{(p+1)/4}$, $\beta \leftarrow d \cdot (2\alpha)^{-1}$
9: $y \leftarrow (\alpha^2 \stackrel{?}{=} z)\ \alpha + \beta i : -\beta - \alpha i$ // compute basis
10: **return** $S_1 \leftarrow (x, y)$, $S_2 \leftarrow (ur^2 x, u_0 r y)$

5 Pairing computation

The pairing computation techniques by Costello *et al.* [3] are based on curves in a variant of the Montgomery model, with projective coordinates (X^2, XZ, Z^2, YZ), which turned out to be the best setting among several models they assessed. We will argue that the older and today less favoured short Weierstraß model leads to more efficient pairing algorithms.

Interestingly, Costello *et al.* dismiss the technique of denominator elimination [14] and keep numerators and denominators separate during pairing evaluation. We point out, however, that pairing values are defined over $\mathbb{F}_{p^2}$ and the inverse of a field element $a + bi$ is $(a - bi)/(a^2 + b^2)$. Hence, rather than keeping a separate denominator $a + bi$ one can simply and immediately multiply the pairing value by the conjugate $a - bi$ instead; the result only differs from the original one by a denominator consisting of the norm $a^2 + b^2 \in \mathbb{F}_p$, and this denominator does get eliminated by the final exponentiation in the reduced Tate pairing computation. This leaves the cost of pairing computation unchanged, but it simplifies the implementation as it entirely does away with separate numerators and denominators.

Let $r \geq 0$ be the pairing order. For embedding degree $k = 1$, $r \mid \Phi_1(p^2) = p^2 - 1 = 2^m \cdot 3^n \cdot (p-1)$, and by construction r is always either 2^m or 3^n. We will be interested in computing reduced Tate pairings of order r, whose first argument must have that order as well. In the case of compressed SIDH keys, pairings of the following forms are computed together (recall that a fifth pairing $e_0 := \mathsf{e}_r(P,Q) = \mathsf{e}_r(P_0,Q_0)^{\deg\phi}$ is readily available through precomputation):

$$e_1 := \mathsf{e}_r(P,R_1),\ e_2 := \mathsf{e}_r(P,R_2),$$
$$e_3 := \mathsf{e}_r(Q,R_1),\ e_4 := \mathsf{e}_r(Q,R_2)$$

where the first two pairings share the same first argument P, and next two pairings share the same first argument Q.

From now on, we will split the discussion into two cases: binary-order pairings, $r = 2^m$, and ternary-order pairings, $r = 3^n$. The curve equation in the short Weierstraß model is $E_W : v^2 = u^3 + au + b$. Given a Montgomery curve $E_M : y^2 = x^3 + Ax^2 + x$, the corresponding short Weierstraß model is obtained via $a = 1 - A^2/3$, $b = (2A^3 - 9A)/27$, and a point $(x, y) \in E_M$ maps to a point $(u, v) \in E_W$ by setting $u = x + A/3$, $v = y$. For convenience, we extend Jacobian coordinates $[X : Y : Z]$ with a fourth component, $[X : Y : Z : T]$ with $T = Z^2$.

5.1 Binary-order pairings

The computation of the reduced Tate pairing $\mathsf{e}_r(P,Q)$ of order $r = 2^m$ proceeds as described in Algorithm 5.1, which requires doubling a point $V \in E(\mathbb{F}_{p^2})$. The doubling formulas in Jacobian coordinates have a single exception, that occurs when the point being doubled has order 2. That is, when $y = 0$, since the angular coefficient of the tangent to the curve at that point becomes undefined. That exception, however, can only occur deterministically in the scenario contemplated here, namely at the last step of the Miller loop; since by definition the first pairing argument is always a point of order 2^m, chosen by the very entity that is computing the pairing.

Besides, the difference in runtime reveals no private information, since the pairing arguments are either already public for being part of a conventional torsion basis, or else are about to be made public for being part of a public key.

Algorithm 5.1 Tate2(P,Q): basic reduced Tate pairing of order $r = 2^m$:

INPUT: points P, Q.
OUTPUT: $\mathsf{e}_r(P,Q)$.

1: $f \leftarrow 1, \quad V \leftarrow P$
2: **for** $i \leftarrow 0$ **to** $m-1$ **do**
3: $\quad f \leftarrow f^2 \cdot g_{V,V}(Q)/g_{[2]V}(Q), \quad V \leftarrow [2]V$
4: **end for**
5: **return** $\mathsf{e}_r(P,Q) \leftarrow f^{(p^2-1)/r}$

In Algorithm 5.1, the function $g_{U,V}$ is defined to be the lines through U and V. If $U = V$ then $g_{U,V}$ is the tangent at U, and if either $U = \infty$ or $V = \infty$ then $g_{U,V}$ is the vertical line at the other point. Also we denote by g_U the value $g_{U,-U}$. The most efficient doubling algorithm for (either plain or modified) Jacobian coordinates appears to be one devised by Bernstein and Lange [15], which maintains an additional coordinate $U := aZ^4$. However, that algorithm does not directly compute the value of $T = Z^2$ that will be needed for pairing calculation. We will thus modify the Bernstein-Lange formulas so that the cost of recovering T is less than that of squaring Z.

Let $V = [X : Y : Z : U : T]$ and $[2]V = [X' : Y' : Z' : U' : T']$ in the extended coordinate system defined above. Then, initializing $T \leftarrow Z^2$ and $U \leftarrow aT^2$:

$$X_2 \leftarrow X^2;\ Y_2 \leftarrow Y^2;\ W \leftarrow 2Y_2;\ W_2 \leftarrow W^2;$$
$$M \leftarrow 3X_2 + U;\ S \leftarrow (X+W)^2 - X_2 - W_2;$$
$$X' \leftarrow M^2 - 2S;\ Y' \leftarrow M \cdot (S - X') - 2W_2;$$
$$Z' \leftarrow (Y+Z)^2 - Y_2 - T;\ T' \leftarrow (Z')^2;\ U' \leftarrow 4W_2 \cdot U;$$

The cost is $2\mathbf{M} + 7\mathbf{S} + 16\mathbf{A} = 20\mathbf{m} + 63\mathbf{a}$. This is only $1\mathbf{m} + 5\mathbf{a}$ more than the cost $3\mathbf{M} + 5\mathbf{S} + 14\mathbf{A} = 19\mathbf{m} + 58\mathbf{a}$ of doubling without yielding Z^2 as a by-product.

This algorithm yields the intermediate values $\lambda_N := M$, $\lambda_D := Z' = 2YZ$, W, and Y_2, besides the point coordinates. These values, together with $L \leftarrow Z' \cdot T$, $R \leftarrow Z' \cdot \bar{T}$, are useful in the calculation of a function equivalent to $g_{V,V}(Q)/g_{[2]V}(Q)$, namely $\tilde{g}_2(V,Q) := (M \cdot (T \cdot x - X) + W - L \cdot y) \cdot R \cdot (T' \cdot x - X')^-$ when $V \neq O$ and $[2]V \neq O$ (i.e. $Z \neq 0$ and $Z' \neq 0$), $\tilde{g}_2(V,Q) := (T \cdot x - X) \cdot \bar{T}$ when $V = -V \neq O$ (i.e. $Z \neq 0$ and $Z' = 0$), or simply $\tilde{g}_2(V,Q) := 1$ when $Z = O$. Denominators in the base field, namely $|Z^2 \cdot (T' \cdot x - X')|^2 \in \mathbb{F}_p$ in the first case and $|Z^2|^2 \in \mathbb{F}_p$ in the second case, are eliminated.

Since the scenario where the pairing computations take place only involve bases of the 2^m-torsion group, and hence points of full order 2^m, the exceptional formula is indeed never invoked until the end of Miller's loop. The difference in processing time is irrelevant for security here, since the computations only involve information that is meant to be public.

However, one can further optimize the computation of (a function equivalent to) $\tilde{g}_2(V,Q)$. First, the expression $(T' \cdot x - X')$ that occurs at a certain step will play the role of $(T \cdot x - X)$ at the next step, so one can simply store it from one step to the next and thus save $1\mathbf{M}$. Second, one can show that all R and $\bar{T}$ factors that appear in the definition of $\tilde{g}_2(V,Q)$ are irrelevant to the pairing value, and can be omitted. We provide the details in the Appendix B.

Consequently, initializing $h \leftarrow T \cdot x - X$ before Miller's loop at a cost of $1\mathbf{M}$ per pairing, the line function value g can be evaluated as

$$g \leftarrow M \cdot h + W - L \cdot y;\ h \leftarrow T' \cdot x - X';\ g \leftarrow g \cdot \bar{h}$$

at a cost of $4\mathbf{M} + 3\mathbf{A} = 12\mathbf{m} + 26\mathbf{a}$ per step of Miller's loop. The cost of computing $L \leftarrow Z' \cdot T$ alone is $1\mathbf{M} = 3\mathbf{m} + 5\mathbf{a}$. This completes the construction of a line function $\hat{g}_2(V,Q)$ equivalent to $\tilde{g}_2(V,Q)$.

The updating of f at each step as $f \leftarrow f^2 \cdot \hat{g}_2(V,Q)$ incurs $2\mathbf{m} + 3\mathbf{a}$ to compute the complex square f^2 plus $3\mathbf{m} + 5\mathbf{a}$ to compute $f^2 \cdot \hat{g}_2(V,Q)$ from f^2 and $\hat{g}_2(V,Q)$, totaling $5\mathbf{m} + 8\mathbf{a}$. Therefore, the proposed variant has the following overall cost per step:

- (shared) cost of point doubling and line function construction: $20\mathbf{m} + 63\mathbf{a} + 3\mathbf{m} + 5\mathbf{a} = 23\mathbf{m} + 68\mathbf{a}$;
- (individual) cost of line function evaluation and accumulation: $12\mathbf{m} + 26\mathbf{a} + 5\mathbf{m} + 8\mathbf{a} = 17\mathbf{m} + 34\mathbf{a}$.

By comparison, the Costello *et al.* [3] technique has the following costs:

- (shared) cost of point doubling and line function construction: $9\mathbf{M} + 5\mathbf{S} + 1\mathbf{s} + 7\mathbf{a} = 37\mathbf{m} + 1\mathbf{s} + 67\mathbf{a}$;
- (individual) cost of line function evaluation and accumulation: $5\mathbf{M} + 2\mathbf{S} + 2\mathbf{s} + 1\mathbf{a} = 19\mathbf{m} + 2\mathbf{s} + 32\mathbf{a}$;

Therefore in the present case, where one has to compute pairs of pairings that share the same first argument, our technique costs a fraction $\approx (23 + 2 \cdot 17)/(37.8 + 2 \cdot 20.6) = 57/79 \approx$ 72% of the Costello *et al.* method, assuming $1\mathbf{s} \approx 0.8\mathbf{m}$ and essentially ignoring $\mathbf{a}$.

TABLE 1: Cost of the binary Miller loop (ratio assumes $\mathbf{s} \approx 0.8\mathbf{m}$ and ignores $\mathbf{a}$).

pairings	Costello *et al.*	ours	ratio
1	$56\mathbf{m}+3\mathbf{s}+99\mathbf{a}$	$40\mathbf{m}+102\mathbf{a}$	≈ 0.685
2	$75\mathbf{m}+5\mathbf{s}+131\mathbf{a}$	$57\mathbf{m}+136\mathbf{a}$	≈ 0.722
$2^\dagger$	$75\mathbf{m}+5\mathbf{s}+131\mathbf{a}$	$55\mathbf{m}+126\mathbf{a}$	≈ 0.696

† Simultaneous pairings on entangled bases

5.1.1 Pairings on an entangled basis

If two pairings $e(P,R_1)$, $e(P,R_2)$ sharing the same first argument P are computed on an entangled basis $R_1 = (x_1, y_1)$, $R_2 = (x_2, y_2)$ with $x_2 = t^2 \cdot x_1$, $y_2 = t \cdot y_1$, one can slightly improve the line function evaluation and accumulation, exploiting the fact that multiplication by carefully chosen t or t^2 given the values of $T' \cdot x_1$ or $L \cdot y_1$ is less expensive than the full multiplications $T' \cdot x_2$ or $L \cdot y_2$ for generic (x_2, y_2).

Specifically, for $t = (1+i)r$ and $t^2 = 2ir^2$ with some small $r \in \mathbb{F}_p$, the cost of a dedicated implementation of simultaneous pairings on entangled bases drops by $2\mathbf{m}+10\mathbf{a}$, thus becoming only $55\mathbf{m}+126\mathbf{a}$, or less than 70% the cost of the Costello *et al.* method. The performance improvements brought about by the techniques we proposed are summarized on Table 1. Our proposed variant of the simultaneous reduced Tate pairing is shown in full detail as Algorithm A.1 in the Appendix A.

5.2 Ternary-order pairings

The computation of the reduced Tate pairing $\mathsf{e}_r(P,Q)$ of order $r = 3^n$ proceeds as described in Algorithm 5.2. Again, the tripling formulas in Jacobian coordinates have an exception when $y = 0$, but this can be handled in a similar fashion to the binary case. The difference in runtime reveals no private information for the same reason, namely only public data is involved in the pairing computations.

Algorithm 5.2 Tate3(P,Q): basic reduced Tate pairing of order $r = 3^n$:

INPUT: points P, Q.
OUTPUT: $\mathsf{e}_r(P,Q)$.

1: $f \leftarrow 1, \quad V \leftarrow P$
2: **for** $i \leftarrow 0$ **to** $n-1$ **do**
3: $\quad f \leftarrow f^3 \cdot g_{V,V}(Q) \cdot g_{V,[2]V}(Q)/(g_{[2]V}(Q) \cdot g_{[3]V}(Q))$,
4: $\quad V \leftarrow [3]V$
5: **end for**
6: **return** $\mathsf{e}_r(P,Q) \leftarrow f^{(p^2-1)/r}$

The most efficient tripling algorithm known for Jacobian coordinates appears to be one devised by Bernstein and Lange [15]. Its cost is $6\mathbf{M}+10\mathbf{S}+25\mathbf{A} = 38\mathbf{m}+110\mathbf{a}$ for a curve with generic equation coefficients. We present a variant for the same modified Jacobian coordinates used in the binary case, with cost $5\mathbf{M}+11\mathbf{S}+31\mathbf{A} = 37\mathbf{m}+120\mathbf{a}$.

Let $V = [X:Y:Z:T:U]$ and $[3]V = [X':Y':Z':T':U']$ in the extended coordinate system as before. Then, initializing $T \leftarrow Z^2$ and $U \leftarrow aT^2$:

$$
\begin{aligned}
&X_2 \leftarrow X^2;\ Y_2 \leftarrow Y^2;\ Y_4 \leftarrow Y_2^2;\\
&M \leftarrow 3X_2 + U;\ M_2 \leftarrow M^2;\\
&D \leftarrow (X+Y_2)^2 - X_2 - Y_4;\ F \leftarrow 6D - M_2;\\
&F_2 \leftarrow F^2;\ W \leftarrow 2Y_2;\ W' \leftarrow 2W;\ S \leftarrow 16Y_4;\\
&G \leftarrow (M+F)^2 - M_2 - F_2 - S;\ G' \leftarrow S - G;\\
&H \leftarrow 2F_2;\ H_2 \leftarrow H^2;\ H' \leftarrow 4G;\ F' \leftarrow 2F;\\
&X' \leftarrow (X+H)^2 - X_2 - H_2 - W' \cdot H';\\
&Y' \leftarrow 2Y \cdot (H' \cdot G' - F' \cdot H);\ Z' \leftarrow (Z+F)^2 - T - F_2;\\
&T' \leftarrow (Z')^2;\ U' \leftarrow 4H_2 \cdot U
\end{aligned}
$$

This algorithm yields intermediate values F, F', G', W, W', and M, besides the point coordinates. These values, together with $L \leftarrow ((Y+Z)^2 - Y_2 - T) \cdot T$ and $R \leftarrow F \cdot \bar{T}$, are useful in the calculation of a function equivalent to $g_{V,V}(Q) \cdot g_{V,[2]V}(Q)/(g_{[2]V}(Q) \cdot g_{[3]V}(Q))$, namely $\tilde{g}_3(V,Q) := (M \cdot h + d) \cdot (G' \cdot h + F' \cdot d) \cdot (W' \cdot h + F)^- \cdot R \cdot \bar{h}_3$ when $[3]V \neq O$; $\tilde{g}_3(V,Q) := (M \cdot h + d) \cdot \bar{L}$ when $[3]V = O$ but $V \neq O$; or simply $\tilde{g}_3(V,Q) := 1$ when $V = O$, where $h := T \cdot x - X$, $d := W - L \cdot y$, $h_3 := T' \cdot x - X'$.

One can further optimize the computation of a function $\hat{g}_3(V,Q)$ equivalent to $\tilde{g}_3(V,Q)$ in a similar fashion to what was done for the binary case. First, the expression $T' \cdot x - X'$ that occurs at a certain step will play the role of $T \cdot x - X$ at the next step, so one can simply store it from one step to the next and thus save $1\mathbf{M}$. Second, one can show that all R and $\bar{L}$ factors that appear in the definition of $\tilde{g}_3(V,Q)$ are irrelevant to the pairing value, and can be omitted. We give more details in the Appendix B.

Consequently, the parabola function construction can be completed by computing only L as above at a cost $1\mathbf{M}+1\mathbf{S}+3\mathbf{A}$. After initializing $h \leftarrow T \cdot x - X$ before Miller's loop at a cost of $1\mathbf{M}$ per pairing, the value g of the parabola function can be evaluated as

$$
\begin{aligned}
d &\leftarrow W - L \cdot y;\\
g &\leftarrow (M \cdot h + d) \cdot (G' \cdot h + F' \cdot d) \cdot (W' \cdot h + F);\\
h &\leftarrow T' \cdot x - X';\\
g &\leftarrow g \cdot \bar{h};
\end{aligned}
$$

at a cost $9\mathbf{M}+5\mathbf{A}$ per step of Miller's loop, except at the final step, when it is simply $g \leftarrow M \cdot h + d$. This completes the construction of a parabola function $\hat{g}_3(V,Q)$ equivalent to $\tilde{g}_3(V,Q)$.

The updating of f at each step as $f \leftarrow f^3 \cdot \hat{g}_3(V,Q)$ incurs a cost $1\mathbf{C}$ to compute the complex cube f^3, plus $1\mathbf{M}$ to compute $f^3 \cdot \hat{g}_3(V,Q)$ from f^3 and $\hat{g}_3(V,Q)$. Therefore, the proposed variant has the following overall cost per step, where again the shared part is amortized among simultaneous pairings that share the same first argument:

- (shared) cost of point tripling and parabola function construction: $5\mathbf{M}+11\mathbf{S}+31\mathbf{A}+1\mathbf{M}+1\mathbf{S}+3\mathbf{A} = 40\mathbf{m}+134\mathbf{a}$;
- (individual) cost of parabola function evaluation and accumulation: $9\mathbf{M}+5\mathbf{A}+1\mathbf{C} = 32\mathbf{m}+2\mathbf{s}+66\mathbf{a}$.

By comparison, the Costello *et al.* [3] technique has the following costs:

TABLE 2: Cost of the ternary Miller loop (ratio assumes $\mathbf{s} \approx 0.8\mathbf{m}$ and ignores $\mathbf{a}$).

pairings	Costello *et al.*	ours	ratio
1	$103\mathbf{m}+6\mathbf{s}+188\mathbf{a}$	$72\mathbf{m}+2\mathbf{s}+200\mathbf{a}$	≈ 0.683
2	$137\mathbf{m}+6\mathbf{s}+248\mathbf{a}$	$104\mathbf{m}+2\mathbf{s}+266\mathbf{a}$	≈ 0.756

- (shared) cost of point tripling and construction of the parabola functions: $19\mathbf{M}+6\mathbf{S}+6\mathbf{s}+15\mathbf{a} = 69\mathbf{m}+6\mathbf{s}+128\mathbf{a}$;
- (individual) cost of evaluating the parabola functions and accumulating the results: $10\mathbf{M}+2\mathbf{S}+4\mathbf{a} = 34\mathbf{m}+60\mathbf{a}$.

Therefore in the present case, where one has to compute pairs of pairings that share the same first argument, our technique costs a fraction $\approx (40+2\cdot 33.6)/(73.8+2\cdot 34) = 107.2/141.8 \approx 76\%$ of the Costello *et al.* method, assuming $1\mathbf{s} \approx 0.8\mathbf{m}$ and essentially ignoring $\mathbf{a}$.

The performance improvements brought about by the techniques we propose are summarized on Table 2. Our proposed variant of the simultaneous reduced Tate pairing is shown in full detail in the Appendix as Algorithm A.2.

6 Discrete logarithm computation

Let $L := \ell^w$ for some integer $w > 0$, and let $\mu_{L^e} \subset \mathbb{F}_{p^2}$ be the set of L^e-th roots of unity in $\mathbb{F}_{p^2}$, i.e. $\mu_{L^e} := \{v \in \mathbb{F}_{p^2} \mid v^{L^e} = 1\}$. Inverting in μ_{L^e} is a mere conjugation, $(a+bi)^{-1} = a-bi$ since the norm is 1. The Pohlig-Hellman method (Algorithm 6.1) [16], which computes the discrete logarithm of $c \in \mu_{L^e}$, requires solving an equation of the form

$$r_k^{L^{e-1-k}} = s^{d_k}$$

where $s = g^{L^{e-1}}$ has order L and, for $k = 0,\dots,e-1$, $d_k \in \{0,\dots,L-1\}$ is an L-ary digit, $r_0 = c$, and r_{k+1} depends on r_k and d_k.

Algorithm 6.1 Basic Pohlig-Hellman discrete logarithm algorithm

INPUT: generator $g \in \mu_{L^e}$, challenge $c \in \mu_{L^e}$.
OUTPUT: $d := \log_g c$, i.e. $g^d = c$.

$s \leftarrow g^{L^{e-1}}$ // NB: $s^L = 1$
$d \leftarrow 0$, $r_0 \leftarrow c$
for $k \leftarrow 0$ **to** $e-1$ **do**
  $v_k \leftarrow r_k^{L^{e-1-k}}$
  find $d_k \in \{0,\dots,L-1\}$ such that $v_k = s^{d_k}$
  $d \leftarrow d + d_k L^k$, $r_{k+1} \leftarrow r_k \cdot g^{-L^k d_k}$
end for // NB: $g^d = c$
return d

Assuming that g^{-L^k} is precomputed and stored for all k as a by-product of the computation of s, the naive strategy to obtain the discrete logarithm requires repeatedly computing the exponential $r_k^{L^{e-1-k}}$ at the cost of $e-1-k$ raisings to the L, then solving a small discrete logarithm instance in a subgroup of order L to get one L-ary digit, then clearing that digit in the exponent of r_k at a cost not exceeding L multiplications to obtain r_{k+1}. The overall cost is thus $O(e^2)$.

It turns out that this strategy is far from optimal, as pointed out by Shoup [17, Chapter 11]. The crucial task is to obtain the sequence $r_0^{L^{e-1}}, r_1^{L^{e-2}}, r_2^{L^{e-3}}, \dots, r_{e-1}^{L^0}$ in this order, since each r_k depends on the previous one. We can visualize this task using a directed acyclic graph Δ strikingly similar to De Feo *et al.*'s T_n graph, which they call a "discrete equilateral triangle", that models the construction of smooth-degree isogenies [6, Section 4.2.2].

In our case, the set of vertices is $\{\Delta_{j,k} \mid j+k \le e-1\}$ where $\Delta_{j,k} := r_k^{L^j}$. Each vertex has either two downward outgoing edges, or no edges at all. Vertices $\Delta_{j,k}$ with $j+k > e-1$ have two edges: a left edge $\Delta_{j,k} \to \Delta_{j+1,k}$ that models raising the source vertex to the L-th power to yield the destination vertex, $r_k^{L^{j+1}} \leftarrow (r_k^{L^j})^L$, and a right edge $\Delta_{j,k} \to \Delta_{j,k+1}$ that models clearing the $(j+k)$-th digit in the exponent of the source vertex, $r_{k+1}^{L^j} \leftarrow r_k^{L^j} \cdot g^{-L^{(j+k)} d_k}$. Vertices $\Delta_{j,k}$ with $j+k = e-1$ are leaves since they have no outgoing edges.

De Feo *et al.* [6, Equation 5] describe an $O(e^2)$ dynamic programming algorithm that computes the cost of an optimal subtree of Δ with root at Δ_{00} and covering all leaves. If the cost of traversing a left or right edge is p or q respectively, and the cost of an optimal subtree of k edges is $C_{p,q}(k)$, their algorithm is based on the relations $C_{p,q}(1) = 0$ and $C_{p,q}(k) = \min_{1\le j<k}\big(C_{p,q}(j) + C_{p,q}(k-j) + (k-j)p + jq\big)$ for $k > 1$.

The naive dynamic programming approach is to store the values of $C_{p,q}(k)$ for $k = 1\dots e$, invoking the above relation $k-1$ times at each step to find the corresponding minimum, for a total $e(e-1)/2$ invocations, hence the $O(e^2)$ cost. However, because $C_{p,q}(k)$ has no local minimum other than the single global minimum (or two adjacent, equivalent copies of the global minimum at worst), one can find that minimum with a variant of binary search that compares two consecutive values near the middle of the search interval $[1\dots k-1]$ and then halves that interval. This yields the $O(e \log e)$ Algorithm 6.2, which computes $C_{p,q}(k)$ and the structure of the optimal traversal strategy by storing the values of j above that attain the minimum at each step.

Algorithm 6.2 OptPath(p,q,e): optimal subtree traversal path

INPUT: p,q: left and right edge traversal cost; e: number of leaves of Δ.
OUTPUT: P: optimal traversal path

Define $C[1\dots e]$ as an array of costs and $P[1\dots e]$ as an array of indices.
$C[1] \leftarrow 0$, $P[1] \leftarrow 0$
for $k \leftarrow 2$ **to** e **do**
  $j \leftarrow 1$, $z \leftarrow k-1$
  while $j < z$ **do**
    $m \leftarrow j + \lfloor (z-j)/2 \rfloor$, $w \leftarrow m+1$
    $t_1 \leftarrow C[m] + C[k-m] + (k-m)\cdot p + m \cdot q$
    $t_2 \leftarrow C[w] + C[k-w] + (k-w)\cdot p + w \cdot q$
    if $t_1 \le t_2$ **then**
      $z \leftarrow m$
    else
      $j \leftarrow w$
    end if
  end while
  $C[k] \leftarrow C[j] + C[k-j] + (k-j)\cdot p + j \cdot q$, $P[k] \leftarrow j$
end for
return P

6.1 Discrete logarithm computation cost

The cost of an optimal strategy depends on the individual costs of traversing a left edge and a right edge. We now show that, because of our proposed reverse basis decomposition technique, the total cost of discrete logarithm computation is drastically reduced. Recall that $L := \ell^w$, where ℓ is a small prime and $w > 0$ is a small integer. A left edge traversal represents the computation $r_k^{L^{j+1}} \leftarrow (r_k^{L^j})^L$ at a cost $w\mathbf{S} \approx 1.6w\mathbf{m}$ in the binary case and $w\mathbf{C} = w(2\mathbf{m} + 1\mathbf{s}) \approx 2.8w\mathbf{m}$ in the ternary case.

A right edge traversal represents the computation $r_{k+1}^{L^j} \leftarrow r_k^{L^j} \cdot g^{-L^{(j+k)}d_k}$, which can be performed via table lookup $r_{k+1}^{L^j} \leftarrow r_k^{L^j} \cdot T[j+k][d_k]$ where $T[u][d] := g^{-L^u \cdot d}$. Since $j+k \le e-1$, the table size is $e \cdot L$ field elements. This enables a tradeoff when computing discrete logarithms in μ_{ℓ^m} with $w \mid m$, namely, by computing discrete logarithms in $\mu_{(\ell^w)^{m/w}}$, which coincides with μ_{L^e} for $e = m/w$. In that case the table size can be written $(m/w) \cdot \ell^w$ to show more clearly the size dependence on these parameters.

However, no more than a single multiplication is incurred regardless of ℓ, e, or w, namely, $1\mathbf{M} \approx 3\mathbf{m}$. When w is very small, avoiding the multiplication for $d_k = 1$ noticeably reduces the running time and requires fewer table entries. Moreover, the table is *fixed* with the reverse basis decomposition technique, because $g = \mathsf{e}(P_\mathsf{B}, Q_\mathsf{B})^{\deg \phi_\mathsf{A}}$, or $g = \mathsf{e}(P_\mathsf{A}, Q_\mathsf{A})^{\deg \phi_\mathsf{B}}$, thus incurring no table building cost at runtime for each newly generated key. Even the simple discrete logarithm instances at the leaves only incur $O(L)$ lookups on the same table, since $s^{d_k} = \overline{T[e-1][d_k]}$.

Algorithm 6.3 summarizes the proposed technique, combining Shoup's RDL algoritm [17, Section 11.2.3] with the optimal divide-and-conquer strategy of De Feo *et al.* and the efficient table lookup enabled by reverse basis decomposition. Following Shoup's analysis as indicated, and assuming that the optimal strategy is close to balanced (a reasonable assumption, according to De Feo *et al.*), we obtain an asymptotic cost $O(e \log e)$ multiplications if $\mathbb{F}_p$. This turns out to be quite close to the experimentally observed costs (see below). Notice that, contrary to Shoup, we do not need the baby-step giant-step algorithm to compute elementary logarithms, since in our case they correspond to groups of order 2 or 3.

Algorithm 6.3 Traverse(r, j, k, z, L, P, T, d)

INPUT: r: value of root vertex Δ_{jk}, i.e. $r := r_k^{L^j}$; j, k: coordinates of root vertex Δ_{jk}; z: number of leaves in subtree rooted at Δ_{jk}; P: traversal path; T: lookup table.
OUTPUT: d: digits (base L) of $\log_g r_0$.
REMARK: initial call is Traverse$(r_0, 0, 0, e, L, P, T, d)$.

1: **if** $z > 1$ **then**
2: $t \leftarrow P[z]$ // z leaves: t to the left exp, $z-t$ to the right
3: $r' \leftarrow r^{L^{z-t}}$ // go left $(z-t)$ times
4: Traverse$(r', j+(z-t), k, t, L, P, T, d)$
5: $r' \leftarrow r \cdot \prod_{h=k}^{k+t-1} T[j+h][d_h]$ // go right t times
6: Traverse$(r', j, k+t, z-t, L, P, T, d)$
7: **else** // leaf
8: find $t \in \{0, \ldots, L-1\}$ such that $r = \overline{T[e-1][t]}$
9: $d_k \leftarrow t$ // recover k-th digit d_k of the discrete logarithm from $r = s^{d_k}$
10: **end if**

The resulting improvements are substantial. For discrete logs in $\mu_{2^{372}}$, the optimal cost is $\approx 4958.4\mathbf{m}$ for $w = 1$, $\approx 3127.9\mathbf{m}$ for $w = 3$, and $\approx 2103.7\mathbf{m}$ for $w = 6$. For discrete logarithms in $\mu_{3^{239}}$, the optimal cost is $\approx 4507.6\mathbf{m}$ for $w = 1$, $\approx 2638.1\mathbf{m}$ for $w = 3$, and $\approx 1739.8\mathbf{m}$ for $w = 6$.

Tradeoffs are also possible. Instead of being a matrix of size $(m/w) \cdot \ell^w$, the lookup table could be restricted to a single array $T_1[u] := g^{-\ell^{wu}}$ of (m/w) entries, by computing $T_1[u]^d = g^{-\ell^{wu} \cdot d}$ on demand using an optimal multiplication chain for cyclotomic exponentiation. For instance, discrete logs in $\mu_{2^{372}}$ with $w = 3$ would require a table of size 124 at an average cost $\approx 4453.9\mathbf{m}$. For comparison, the best results reported in [3, Section 5] are $2150\mathbf{m} + 7652\mathbf{s} \approx 8271.6\mathbf{m}$ for discrete logs in $\mu_{2^{372}}$ and $5320\mathbf{m} + 3349\mathbf{s} \approx 7999.2\mathbf{m}$ for discrete logs in $\mu_{3^{239}}$, both with $w = 3$, which is optimal in that technique; increasing w is observed to actually cause a cost increase.

Table 3 summarizes the gains our technique makes possible and compares them against the results of [3], in terms of both the raw number of multiplications in the base field and the ratio between our results and theirs. We recall that no side-channel security concern arises from this technique, since all information involved in the processing is public.

6.2 Improved Pohlig-Hellman for generic w

Rewriting the Pohlig-Hellman algorithm 6.1 to compute discrete logarithms in μ_{ℓ^m} while representing the exponent d in base ℓ^w with $d = \sum_{i=0}^{\lceil m/w \rceil} d_i \ell^{wi}$, thus recovering more bits of the exponent per digit, is straightforward when $w \mid m$. On the other hand, the case where $w \nmid m$ requires special handling. The worst case happens when m is a prime number and hence not divisible by any $1 < w < m$. This is the case for the SIDH prime in our implementation where the ternary discrete logarithm is defined over a subgroup of order 3^{239}, since 239 is a prime.

An algorithm that deals with the cases where $w \nmid m$ will be derived next. Denote the challenge by $r_0 := g^d \in \mu_{\ell^m}$ where $d = \sum_{i=0}^{\lceil m/w \rceil - 1} d_i \ell^{wi}$ and let r_0 be the root $\Delta_{00} := r_0$ of the graph Δ. Note that Δ here is a discrete equilateral triangle of side $s = \lceil m/w \rceil$ vertices. In the usual divisible-exponent version, a left edge traversal is equivalent to raising the current element Δ_{jk} to the ℓ^w-power, i.e., $r_k^{\ell^{jw+w}} \leftarrow r_k^{\ell^{jw}}$. When $w \mid m$, the value r_k is raised to the ℓ^w-power $(s-k)-1$ times so that the resulting leaf element $\Delta_{j,k} = r_k^{\ell^{(s-k-1)w}}$ for $j+k = s-1$ belongs to a subgroup of order ℓ^w. The corresponding k-th digit is then simply recovered by solving the logarithm $d_k := \log_{g^{m-w}} \Delta_{j,k}$.

When $w \nmid m$, a crucial observation is that when the element r_k is raised to the $\ell^{(s-k-1)w}$-th power, the corresponding leaf element $\Delta_{j,k}$ belongs to a subgroup of order $\ell^{m \bmod w}$ (instead of ℓ^w) since by construction r_k has order ℓ^{m-kw} and

$$\begin{aligned} \Delta_{j,k} &= r_k^{\ell^{(s-k-1)w}} \\ &= r_k^{\ell^{sw-kw-w}} \\ &= r_k^{\ell^{(m-m \bmod w+w)-kw-w}} \\ &= r_k^{\ell^{m-kw-m \bmod w}}. \end{aligned}$$

Thus, solving this smaller logarithm would only allow for recovering partial information of the digit and it would not be possible to retrieve the full exponent in the end of the process. In order to correct the orders of the leaf elements a few modifications of the traversal strategy (Algorithm 6.3) are needed. The key idea for obtaining leaves of desired order

TABLE 3: Discrete logarithm computation costs (assuming $\mathbf{s} \approx 0.8\mathbf{m}$)

group	Costello *et al.* [3]	ours, $w=1$ (ratio)	ours, $w=3$ (ratio)	ours, $w=6$ (ratio)
$\mu_{2^{372}}$	8271.6**m**	4958.4**m** (0.60)	3127.9**m** (0.39)	2103.7**m** (0.25)
$\mu_{3^{239}}$	7999.2**m**	4507.6**m** (0.56)	2638.1**m** (0.33)	1739.8**m** (0.22)

ℓ^w consists of modifying the first left edge traversal from each rightmost vertex $\Delta_{0,k}$. The first left traversals will be given by $\Delta_{1,k} \leftarrow \Delta_{0,k}^{\ell^{m \bmod w}}$ and $\Delta_{j+1,k} \leftarrow \Delta_{j,k}^{\ell^w}$ for $j > 0$, or equivalently, the traversal from $\Delta_{j,k}$ to $\Delta_{j+1,k}$ is defined as $r_k^{\ell^{m \bmod w+jw}} \leftarrow r_k^{\ell^{m \bmod w+(j-1)w}}$. This leads to the following redefinition of $\Delta_{j,k}$. Let the set of vertices be $\{\Delta_{j,k} \mid j+k \le \lceil m/w \rceil - 1\}$, then $\Delta_{j,k} := r_k$ for $j = 0$ and $\Delta_{j,k} := r_k^{\ell^{m \bmod w+(j-1)w}}$ for $j > 0$. By applying this modification, all the leaves will have order ℓ^w except the rightmost one which has order $\ell^{m \bmod w}$ as proven by Lemma 2 in the Appendix C.

The right edge traversals that are responsible for removing the digits of the exponent are also modified. They are given by $\Delta_{j,k+1} \leftarrow \Delta_{j,k} \cdot g^{-d_k \cdot \ell^{m \bmod w+(j-1)w}}$. Since the non-rightmost vertices $\Delta_{j,k}$ are computed as $\Delta_{0,k}^{\ell^{m \bmod w+(j-1)w}}$, the digits in the exponent will have coefficients displaced by $\ell^{w-m \bmod w}$, and therefore a suitable digit-removal table can be defined by $T_2[0][d_i] := g^{-d_i}$ for $0 \le d_i < \ell^w - 1$ and $T_2[u][d_i] := g^{-d_i \cdot \ell^{m \bmod w+(u-1)w}}$ for $0 < u < s$. This table is also used to compute discrete logarithms at leaves $0, 1, \cdots, s-2$ because its last row will have elements of order ℓ^w. A distinct table will be needed for dealing with both non-rightmost right edge traversals and discrete log computation through the rightmost edges $\Delta_{0,k}$. This table is defined by $T_1[v][d_i] := g^{-d_i \cdot \ell^{vw}}$ for $0 \le v < s$.

The TraverseW Algorithm 6.4 details the full procedure to compute the version of Pohlig-Hellman for general w combined with an optimal traversal strategy.

The optimal path P is retrieved by invoking $OptPath(w \cdot p, q, \lceil m/w \rceil)$ where p is the cost of raising to ℓ and q is the cost of a multiplication in the quadratic extension field.

Algorithm 6.4 TraverseW$(r, j, k, z, \ell, w, P, T_1, T_2, d)$

INPUT: r: value of root vertex $\Delta_{j,k}$, i.e. $r := r_k^{\ell^{m \bmod w+(j-1)w}}$; j, k: coordinates of root vertex $\Delta_{j,k}$; z: number of leaves in subtree rooted at $\Delta_{j,k}$; P: traversal path; T_1, T_2: lookup tables; w: base power not dividing m.
OUTPUT: d: digits (base ℓ^w) of $\log_g r_0$.
REMARK: initial call is TraverseW$(r_0, 0, 0, m, \ell, w, P, T_1, T_2, d)$.

1: **if** $z > 1$ **then**
2: $t \leftarrow P[z]$ // z leaves
3: **if** $j > 0$ **then** // go left $(z-t)$ times
4: $r' \leftarrow r^{\ell^{w(z-t)}}$
5: **else**
6: $r' \leftarrow r^{\ell^{m \bmod w+w(z-t-1)}}$
7: **end if**
8: TraverseW$(r', j+(z-t), k, t, \ell, w, P, T_1, T_2, d)$
9: **if** $j = 0$ **then** // go right t times
10: $r' \leftarrow r \cdot \prod_{h=k}^{k+t-1} T_1[j+h][d_h]$
11: **else**
12: $r' \leftarrow r \cdot \prod_{h=k}^{k+t-1} T_2[j+h][d_h]$
13: **end if**
14: TraverseW$(r', j, k+t, z-t, \ell, w, P, T_1, T_2, d)$
15: **else** // leaf
16: **if** $j = 0$ **and** $k = \lceil m/w \rceil - 1$ **then**
17: find $0 \le t < \ell^{m \bmod w}$ s.t. $r = \overline{T_1[\lceil m/w \rceil - 1][t]}$
18: **else**
19: find $0 \le t < \ell^w$ s.t. $r = \overline{T_2[\lceil m/w \rceil - 1][t]}$
20: **end if**
21: $d_k \leftarrow t$ // recover the k-th digit d_k from $r = s^{d_k}$
22: **end if**

7 Point tripling on Montgomery curves

Multiplication by 3^n, be it as a cofactor in the case of the 2^m torsion or as a tool to test linear independence in the 3^n torsion, is a computationally expensive operation. We describe in Algorithm 7.1 an improved method for point tripling on Montgomery curves that, though modest, directly addresses this bottleneck.

The cost of our tripling is $5\mathbf{M} + 6\mathbf{S} + 7\mathbf{A}$ (or one less multiplication in scenarios where the curve coefficient A can be carefully chosen and fixed) with 4 ancillary variables, not counting the left shift (multiplication by 2) which costs no more than an addition but can be precomputed for a given curve. It is less expensive than the previously best tripling algorithm in the literature, which only attains $6\mathbf{M} + 5\mathbf{S} + 7\mathbf{A}$ with 8 ancillary variables [18, Appendix B]. Note that this tripling algorithm can be employed in the key (de)compression operations since they do not require the curve coefficient A to be in projective form. The projective version is only required in the computation of 3^n-isogenies, where field inversions can be avoided if the projective form is adopted. That is the case of the tripling formula by Faz *et al.* [19], which costs $7\mathbf{M} + 5\mathbf{S} + 9\mathbf{A}$.

Algorithm 7.1 Improved tripling on the Montgomery curve $By^2 = x^3 + Ax^2 + x$

INPUT: $P = (x, z)$: a point in xz representation.
OUTPUT: $[3]P = (x', z')$.

1: $t_1 \leftarrow x^2$; $t_2 \leftarrow z^2$;
2: $t_3 \leftarrow t_1 + t_2$
3: $t_4 \leftarrow 2A \cdot ((x+z)^2 - t_3) + t_3$
4: $t_3 \leftarrow (t_1 - t_2)^2$;
5: $t_1 \leftarrow (t_1 \cdot t_4 - t_3)^2$; $t_2 \leftarrow (t_2 \cdot t_4 - t_3)^2$;
6: $x' \leftarrow x \cdot t_2$; $z' \leftarrow z \cdot t_1$;
7: **return** (x', z')

8 Implementation and experimental results

Our improved key compression and decompression techniques have been implemented on top of the SIDH C library [8] to make a full-fledge key exchange available. We left the previous (de)compression functions in the new version so that the experiments can be replicated for comparisons.

Since we only process public information (compression and decompression of public keys), side-channel attacks are not an issue. This enabled, for instance, the adoption of a simple, fast version of the extended Euclidean algorithm.

TABLE 4: Benchmarks in Mcycles on an Intel Core i5-6267U clocked at 2.9 GHz (clang compiler with `-O3` flag, and $\mathbf{s} = \mathbf{m}$ in this implementation). Note also that decompression incorporates the shared Elligator optimization.

	2^m-torsion ($w = 2$)			3^n-torsion ($w = 6$)		
operation	SIDH v2.0 [3]	ours	ratio	SIDH v2.0 [3]	ours	ratio
one discrete logarithm	5.88	2.57	**2.3**	4.71	1.17	**4.0**
pairing phase	33.23	25.37	**1.3**	37.72	29.04	**1.3**
compression	75.49	37.07	**2.0**	79.33	54.14	**1.5**
decompression	28.76	8.97	**3.2**	25.95	12.91	**2.0**

The benchmark methodology consisted of taking the average of individual operations in key (de)compression such as basis generation, discrete log and pairing on 20 thousand executions for random inputs and 800 executions for more expensive operations as key (de)compression.

The initial public curve is the usual supersingular curve $E_0 : y^2 = x^3 + x$ defined over $\mathbb{F}_{p^2}$ where $p = 2^{372}3^{239} - 1$. It is worth mentioning that before applying our (de)compression techniques, the SIDH v2.0 library was first modified to perform Alice's key generation with both points P_A and Q_A defined over the extension $E_0(\mathbb{F}_{p^2})\backslash E_0(\mathbb{F}_p)$ instead of defining P_A in the base field as suggested in [3]. The approach in [3] starts with point $P_A = (x, y) \in E_0(\mathbb{F}_p)$ over the base field and then applies the distortion map τ to get a linearly independent point $Q_A = \tau(P_A) = (-x, iy)$ lying in the trace zero group. This optimization cannot be combined with our techniques because using distortion maps on binary torsions only gives a basis $\langle P_A, \tau(P_A)\rangle = E_0[2^{m-1}]$ of a smaller group of order $2^{2(m-1)}$, and in this case the images of P_A and $Q_A = \tau(P_A)$ under Bob's isogeny consequently generate a smaller torsion as well, i.e. $\langle \phi_B(P_A), \phi_B(Q_A)\rangle = E_B[2^{m-1}]$. In particular, the reverse basis decomposition technique combined with entangled basis would not work since an entangled basis generates the full 2^m-torsion, and this basis cannot be converted to a basis of a smaller torsion, i.e. the change of basis matrix in Equation 3 would not exist. Therefore, the points

$$P_A := 3^{239} \cdot (5 + i, \sqrt{(5 + i)^3 + 5 + i}) \in E_0(\mathbb{F}_{p^2})\backslash E_0(\mathbb{F}_p)$$

and $Q_A := \tau(P_A) \in E_0(\mathbb{F}_{p^2})\backslash E_0(\mathbb{F}_p)$ are selected. Points P_B and Q_B are the ones in [3] since for ℓ^n torsions with ℓ odd, distortion maps do generate the full group $E_0[\ell^n]$ and P_B can be kept over the base field. For the discrete logarithms we set $w = 2$ for the binary case and $w = 6$ for the ternary one. The latter decision is to compensate the fact that we still do not have an entangled basis for the ternary torsion.

The experiments for the 3^n-torsion basis generation are shown in Table 5. The three improvements against the previous 3-descent from SIDH library v2.0 are given. They consist of the naive (although faster than 3-descent), naive + shared Elligator and the naive + shared Elligator without linear independence test basis generation techniques.

Regarding the 2^m-torsion basis generation coupled with the shared Elligator for decompression, our benchmark results show that the 1M cycles barrier could be broken for the first time, requiring only 0.83M cycles to generate a basis which compares extremely well against the previous 2-descent with cost 23.77M cycles in the SIDH v2.0 library. Note that in this case linear independence check is implicitly given (due to Theorem 1). The results are shown in Table 6.

TABLE 5: Benchmark of the **3^n-torsion basis generation** in Mcycles on an Intel Core i5-6267U clocked at 2.9 GHz (clang compiler with `-O3` flag, and $\mathbf{s} = \mathbf{m}$ in this implementation).

technique	source	Mcycles	ratio
3-descent	SIDH v2.0 [3]	19.98	–
naive basis generation	this work	17.33	**1.2**
shared Elligator	this work	13.63	**1.5**
shared ell. + no LI check	this work	7.26	**2.8**

TABLE 6: Benchmark of the **2^m-torsion basis generation** in Mcycles on an Intel Core i5-6267U clocked at 2.9 GHz (clang compiler with `-O3` flag, and $\mathbf{s} = \mathbf{m}$ in this implementation).

technique	source	Mcycles	ratio
2-descent	SIDH v2.0 [3]	23.77	–
entangled basis	this work	1.60	**14.9**
ent. basis + shared ell.	this work	0.83	**28.6**

Table 4 summarizes our experimental results for the high-level operations of key (de)compression. The small differences between the theoretical estimates and the practical results are basically due to the cost of squaring in the base field $\mathbb{F}_p$ that is implemented in SIDH v2.0 library by reusing the modular multiplication instead of adopting an optimized Montgomery squaring in lower level. In this case, the finite field squaring does not take a cost of 0.6 to 0.8**m** although this is possible to be obtained in practice.

Table 7 illustrates the impact of (de)compression algorithms on SIKE. Incorporating key compression and decompression into SIKE.KEM is straightforward; the compression is only done during key generation which is based on the 2^m-torsion of the curves. The upside of the choice of 2^m-torsion is that we have much faster algorithms such as entangled basis generation. The decompression is done immediately before encapsulation. Note that there is no (de)compression on the 3^n-torsion. The much smaller overhead for the prime p964 is because the same compression algorithm is used along with a non-optimized implementation (the only available option) for this prime. For example, for p964 the isogeny computation is a multiplication-based algorithm with quadratic complexity while it is a quasi-linear time algorithm for the other primes.

9 Conclusion

In this paper we have proposed a range of new algorithms and techniques to speed up the supersingular isogeny Diffie-

TABLE 7: Benchmark for the Supersingular Isogeny Key Encapsulation in Mcycles on an Intel Core i5-6267U clocked at 2.9 GHz (clang compiler with `-O3` flag, and $\mathbf{s} = \mathbf{m}$ in this implementation). The entries marked with * denote the runtimes on an actual machine while the ones marked with ** denote estimated runtimes based on operation counts over $\mathbb{F}_p$.

prime	SIKE KeyGen	SIKE encaps.	2^m compression	2^m decompression	overhead on KeyGen	overhead on encaps.
p503	9.1 *	15.1 *	10.2 ($w = 6$) **	3.06 **	112%	20%
p751	27.5 *	44.5 *	31.7 ($w = 6$) *	8.97 *	115%	20%
p964	13570.8 *	19406.8 *	294.7 ($w = 6$) **	84.45 **	2.2%	0.4%

Hellman. For example, in the 2^m-torsion using $w = 2$ for the discrete logarithms, the key compression is about 2× faster than the SIDH library and decompression achieves a factor of 3×, while the basis generation itself is nearly 15× faster during compression and about 29× faster in decompression. The main bottleneck now, by far, is the pairing phase, that takes about 25.4M cycles against 1.6M for basis generation and $4 \times 2.57 \approx 10.3$M for the discrete logarithm phase.

Moreover, the combination of the entangled basis generation and the shared Elligator techniques allowed for a basis generation algorithm that runs in less than the 1M cycles barrier for the first time, where the best previous known algorithm takes 23.77M cycles for the same task.

It is worthwhile to point out that the techniques of entangled basis generation, and the optimal strategy applied to solve smooth-order discrete logarithms, not only set up new speed records for those tasks, but might find new applications in different contexts in cryptography. We leave the possibility of extending the new entangled basis generation technique to non-binary torsions as an open problem.

Acknowledgement

The authors thank the anonymous reviewers for their valuable and enriching comments. J. Doliskani and G. Pereira were supported by NSERC, CryptoWorks21, and Public Works and Government Services Canada. M. Simplicio was supported by Brazilian National Council for Scientific and Technological Development (CNPq) under grant 301198/2017-9. M. Simplicio, P. Barreto and G. Zanon were partially supported by the joint São Paulo Research Foundation (FAPESP) / Intel Research grant 2015/50520-6 "Efficient Post-Quantum Cryptography for Building Advanced Security Applications."

References

[1] D. Jao and L. De Feo, "Towards quantum-resistant cryptosystems from supersingular elliptic curve isogenies," in *Post-Quantum Cryptography – PQCrypto 2011*, ser. Lecture Notes in Computer Science, no. 7071. Taipei, Taiwan: Springer, 2011, pp. 19–34.

[2] R. Azarderakhsh, D. Jao, K. Kalach, B. Koziel, and C. Leonardi, "Key compression for isogeny-based cryptosystems," in *Proceedings of the 3rd ACM International Workshop on ASIA Public-Key Cryptography*. ACM, 2016, pp. 1–10.

[3] 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*, ser. Lecture Notes in Computer Science, no. 10210. Paris, France: Springer, 2017, pp. 679–706.

[4] J. Doliskani, G. C. C. F. Pereira, and P. S. L. M. Barreto, "Faster cryptographic hash function from supersingular isogeny graphs," Cryptology ePrint Archive, Report 2017/1202, 2017, http://eprint.iacr.org/2017/1202.

[5] E. Schaefer and M. Stoll, "How to do a p-descent on an elliptic curve," *Transactions of the American Mathematical Society*, vol. 356, no. 3, pp. 1209–1231, 2004.

[6] 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, 2014.

[7] S. team, "Supersingular isogeny key encapsulation," 2017, https://sike.org.

[8] Microsoft SIDH team, "SIDH v2.0," 2017, https://www.microsoft.com/en-us/research/project/sidh-library/.

[9] M. R. Spiegel and J. Liu, *Mathematical Handbook of Formulas and Tables*, 2nd ed., ser. Schaum's Outline Series. New York, USA: McGraw-Hill, 1999.

[10] D. J. Bernstein, M. Hamburg, A. Krasnova, and T. Lange, "Elligator: Elliptic-curve points indistinguishable from uniform random strings," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM, 2013, pp. 967–980.

[11] D. Husemöller, *Elliptic Curves*, 2nd ed., ser. Graduate Texts in Mathematics. New York, USA: Springer, 2004, vol. 111.

[12] J. H. Silverman, *The Arithmetic of Elliptic Curves*, 2nd ed., ser. Graduate Texts in Mathematics. New York, USA: Springer, 2009, vol. 106.

[13] P. L. Montgomery, "Speeding the Pollard and elliptic curve methods of factorization," *Mathematics of computation*, vol. 48, no. 177, pp. 243–264, 1987.

[14] P. S. L. M. Barreto, H. Y. Kim, B. Lynn, and M. Scott, "Efficient algorithms for pairing-based cryptosystems," in *Advances in Cryptology – Crypto 2002*, ser. Lecture Notes in Computer Science, no. 2442. Santa Barbara (CA), USA: Springer, 2002, pp. 354–368.

[15] D. J. Bernstein and T. Lange, "Analysis and optimization of elliptic-curve single-scalar multiplication," in *Finite Fields and Applications: Proceedings of Fq8*, no. 461. Providence (RI), USA: American Mathematical Society, 2008, pp. 1–18.

[16] S. C. Pohlig and M. E. Hellman, "An improved algorithm for computing logarithms over $GF(p)$ and its cryptographic significance function," *IEEE Transactions on information Theory*, vol. 24, no. 1, pp. 106–110, 1978.

[17] V. Shoup, *A computational introduction to number theory and algebra*. Cambridge University Press, 2005.

[18] S. R. S. 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*, ser. Lecture Notes in Computer Science, no. 9646. Fes, Morocco: Springer, 2016, pp. 84–106.

[19] A. Faz-Hernández, J. López, E. Ochoa-Jiménez, and F. Rodríguez-Henríquez, "A faster software implementation of the supersingular isogeny diffie-hellman key exchange protocol," *IEEE Transactions on Computers*, 2017.

Appendix A
Pairing algorithms

Algorithm A.1 Tate2($P,[Q_j],m$): reduced Tate pairing of order $r=2^m$

INPUT: Curve $E: y^2=x^3+ax+b$
- Point $P=[X_P:Y_P:Z_P]$ on E of order 2^m
- t points $Q_j=[X_{Q_j}:Y_{Q_j}:Z_{Q_j}]$ on E, $Z_{Q_j}\in\{0,1\}$

OUTPUT: List of t values $e_{2^m}(P,Q_j)$

$X\leftarrow X_P$; $Y\leftarrow Y_P$; $Z\leftarrow Z_P$; $T\leftarrow Z^2$; $U\leftarrow a\cdot T^2$
▷ NB: the following operations are in $\mathbb{F}_{p^2}$
for $j\leftarrow 0$ **to** $t-1$ **do**
  $f_j\leftarrow 1$; $h_j\leftarrow T\cdot X_{Q_j}-X$
end for
for $k\leftarrow 0$ **to** $m-1$ **do**
▷ *point doubling and line function construction:*
  $X_2\leftarrow X^2$; $Y_2\leftarrow Y^2$; $W\leftarrow 2Y_2$; $W_2\leftarrow W^2$
  $M\leftarrow 3X_2+U$; $S\leftarrow (X+W)^2-X_2-W_2$
  $X'\leftarrow M^2-2S$; $Y'\leftarrow M\cdot(S-X')-2W_2$
  $Z'\leftarrow (Y+Z)^2-Y_2-T$; $T'\leftarrow (Z')^2$
  $U'\leftarrow 4W_2\cdot U$; $L\leftarrow Z'\cdot T$
  if $Z'=0$ **then** // *exception for points in* $[2]E$
    $X'\leftarrow 1$; $Y'\leftarrow 1$
  end if
▷ *line function evaluation and accumulation:*
  for $j\leftarrow 0$ **to** $t-1$ **do**
    if $Z'\neq 0$ **then**
      $g\leftarrow M\cdot h_j+W-L\cdot Y_{Q_j}$
      $h_j\leftarrow T'\cdot X_{Q_j}-X'$
      $g\leftarrow g\cdot\bar{h}_j$
    else // *exception for points in* $[2]E$
      $g\leftarrow h_j$
    end if
    $f_j\leftarrow f_j^2$; $f_j\leftarrow f_j\cdot g$
  end for
  $X\leftarrow X'$; $Y\leftarrow Y'$; $Z\leftarrow Z'$; $T\leftarrow T'$; $U\leftarrow U'$
end for
▷ *a dedicated final exponentiation should be used next:*
return $[(Z_{Q_j}\stackrel{?}{\neq}0)f_j^{(p^2-1)/r}:1\mid j=0\ldots t-1]$

Algorithm A.2 Tate3($P,[Q_j],n$): reduced Tate pairing of order $r=3^n$

INPUT: Curve $E: y^2=x^3+ax+b$
- Point $P=[X_P:Y_P:Z_P]$ on E of order 3^n
- t points $Q_j=[X_{Q_j}:Y_{Q_j}:Z_{Q_j}]$ on E, $Z_{Q_j}\in\{0,1\}$

OUTPUT: List of t values $e_{3^n}(P,Q_j)$

$X\leftarrow X_P$; $Y\leftarrow Y_P$; $Z\leftarrow Z_P$; $T\leftarrow Z^2$; $U\leftarrow a\cdot T^2$
▷ NB: the following operations are in $\mathbb{F}_{p^2}$
for $j\leftarrow 0$ **to** $t-1$ **do**
  $f_j\leftarrow 1$; $h_j\leftarrow T\cdot X_{Q_j}-X$
end for
for $k\leftarrow 0$ **to** $n-1$ **do**
▷ *point tripling and parabola function construction:*
  $X_2\leftarrow X^2$; $Y_2\leftarrow Y^2$; $Y_4\leftarrow Y_2^2$
  $M\leftarrow 3X_2+U$; $M_2\leftarrow M^2$
  $D\leftarrow (X+Y_2)^2-X_2-Y_4$; $F\leftarrow 6D-M_2$
  $F_2\leftarrow F^2$; $W\leftarrow 2Y_2$; $W'\leftarrow 2W$; $S\leftarrow 16Y_4$
  $G\leftarrow (M+F)^2-M_2-F_2-S$; $G'\leftarrow S-G$
  $H\leftarrow 2F_2$; $H_2\leftarrow H^2$; $H'\leftarrow 4G$; $F'\leftarrow 2F$
  $X'\leftarrow (X+H)^2-X_2-H_2-W'\cdot H'$
  $Y'\leftarrow 2Y\cdot(H'\cdot G'-F'\cdot H)$
  $Z'\leftarrow (Z+F)^2-T-F_2$
  $T'\leftarrow (Z')^2$; $U'\leftarrow 4H_2\cdot U$
  $L\leftarrow ((Y+Z)^2-Y_2-T)\cdot T$
  if $Z'=0$ **then** // *exception for points in* $[3]E$
    $X'\leftarrow 1$; $Y'\leftarrow 1$
  end if
▷ *parabola function evaluation and accumulation:*
  for $j\leftarrow 0$ **to** $t-1$ **do**
    $d\leftarrow W-L\cdot Y_{Q_j}$
    if $Z'\neq 0$ **then**
      $g\leftarrow (M\cdot h_j+d)(G'\cdot h_j+F'\cdot d)(W'\cdot h_j+F)^-$
      $h_j\leftarrow T'\cdot X_{Q_j}-X'$; $g\leftarrow g\cdot\bar{h}_j$
    else // *exception for points in* $[3]E$
      $g\leftarrow (M\cdot h_j+d)$
    end if
    $f\leftarrow f^3$; $f\leftarrow f\cdot g$
  end for
  $X\leftarrow X'$; $Y\leftarrow Y'$; $Z\leftarrow Z'$; $T\leftarrow T'$; $U\leftarrow U'$
end for
▷ *a dedicated final exponentiation should be used next:*
return $[(Z_{Q_j}\stackrel{?}{\neq}0)f_j^{(p^2-1)/r}:1\mid j=0\ldots t-1]$

Appendix B
Irrelevant factors in pairing computation

Consider the function $\tilde{g}_2(V,Q)$ defined in Section 5.1. Let Z_j denote the computed z-coordinate of $[2^j]P$, let $T_j:=Z_j^2$ and let $R_j:=Z_{j+1}\cdot\bar{T}_j=Z_{j+1}\cdot(\bar{Z}_j)^2$ denote the contribution of the R factor above at the j-th step in Miller's loop for $0\le j<m-1$, with $R_{m-1}:=1$, rather than 0 as the general expression would yield, for convenience. One can show by induction that the contribution of all R factors to the pairing value before the final exponentiation is

$$\hat{R}:=\prod_{j=0}^{m-1}R_j^{2^{m-1-j}}=\prod_{j=0}^{m-1}\left(Z_{j+1}\cdot\bar{T}_j\right)^{2^{m-1-j}},$$

which can be rearranged as

$$\hat{R}=(\bar{T}_0)^{2^{m-1}}Z_{m-1}^2\prod_{j=1}^{m-2}\left(Z_j^{2^{m-j}}(\bar{T}_j)^{2^{m-1-j}}\right)$$
$$=(\bar{Z}_0)^{2^m}T_{m-1}\prod_{j=1}^{m-2}\left(Z_j\bar{Z}_j\right)^{2^{m-j}}.$$

But the final exponentiation will erase the first factor as $((\bar{Z}_0)^{2^m})^{(p^2-1)/2^m}=(\bar{Z}_0)^{(p^2-1)}=1$, and also the last product above, which only involves norms in $\mathbb{F}_p$. Hence the actual contribution is simply $\hat{R}^{(p^2-1)/2^m}=T_{m-1}^{(p^2-1)/2^m}$, but the line function at the last step of Miller's loop contributes a factor $(T\cdot x-X)\cdot\bar{T}$ to the pairing value before the final exponentiation, so $\hat{R}$ could be incorporated there as $(T\cdot x-X)\cdot\bar{T}_{m-1}\cdot T_{m-1}\sim T\cdot x-X$.

An analogous situation arises for the ternary case. Consider the function $\tilde{g}_3(V,Q)$ defined in Section 5.2. Let Z_j denote

the computed z-coordinate of $[3^j]P$, let $T_j := Z_j^2$, and let $R_j := 2|Z_j|^2 \cdot R_j = (2F_j \cdot Z_j) \cdot (\bar{Z}_j \cdot \bar{T}_j) = Z_{j+1} \cdot (\bar{Z}_j)^3$ denote a contribution equivalent to that of the R factor above at the j-th step in Miller's loop for $0 \leq j < n-1$, with $R_{n-1} := 1$ for convenience. One can show by induction and term rearrangement that the contribution of all R factors to the pairing value before the final exponentiation is

$$\hat{R} := (\bar{Z}_0)^{3^n} \cdot Z_{n-1}^3 \cdot \prod_{j=1}^{n-2} \left(Z_j \cdot \bar{Z}_j\right)^{3^{n-j}}.$$

Therefore, the actual contribution after the final exponentiation is simply $\hat{R}^{(p^2-1)/3^n} = (Z_{n-1}^3)^{(p^2-1)/3^n}$, but the parabola function at the last step of Miller's loop contributes a factor $((M \cdot h + d) \cdot \bar{L})^3$ to the pairing value before the final exponentiation, so $\hat{R}$ could be incorporated to that expression as $(M \cdot h + d)^3 \cdot (\bar{L})^3 \cdot Z_{n-1}^3 = ((M \cdot h + d) \cdot \bar{Y})^3 \cdot (2\bar{Z}_{n-1} \cdot Z_{n-1})^3 \sim ((M \cdot h + d) \cdot \bar{Y})^3$.

Yet, the remaining factor $(\bar{Y})^3$, or indeed Y itself by virtue of the tripling algorithm is a 2^m-th root of an element from the base field, Y_0. This means that the final exponentiation will erase it, since $(p^2-1)/3^n = (p-1) \cdot 2^m$ and $Y^{(p^2-1)/3^n} = (Y^{2^m})^{p-1} = Y_0^{p-1} = 1$. Hence, it can be simply omitted, leaving only $M \cdot h + d$.

In both the binary and the ternary case, a corresponding simplified formula can be used without making any reference at all to the R factors.

Appendix C
Proof for the discrete log traversal when w does not divide m

We prove the following lemma about the order of the leaf elements in the graph Δ as defined in Section 6.2.

Lemma 2. *Let Δ be the acyclic graph defined in Section 6 with root $\Delta_{00} := r_0$ where $r_0 = g^d \in \mu_{\ell^m}$ and $d = \sum_{i=0}^{\lceil m/w \rceil - 1} d_i \ell^{iw}$ where $0 \leq d_i < \ell^w$. Let w be an integer that does not divide m. Then, if the first left walk from the elements $\Delta_{0,k}$ for $0 \leq k < \lceil m/w \rceil - 1$ is defined by raising to the power of $\ell^{m \pmod w}$ and the subsequent left walks by raising to ℓ^w, then all the leaves except the rightmost one will have order ℓ^w. In addition, the rightmost $\lceil m/w \rceil$-th leaf has order $\ell^{m \bmod w}$.*

Proof. By construction, the leaves of Δ can be computed as $\Delta_{j,k} := r_k^{\ell^{([m/w]-1-(k+1))w+m \bmod w}}$ for $j + k = \lceil m/w \rceil - 1$ and $k < \lceil m/w \rceil - 1$ by simply using the raise-based strategy, i.e., leaves are reached by always traversing to the left, except the rightmost leaf which is reached by a sequence of $\lceil m/w \rceil - 1$ walks to the right. Let u be the integral part of the division e/w and $t := m \bmod w$ the respective remainder. Thus, we have $\lceil m/w \rceil = u + 1$. Rewrite $\Delta_{j,k}$ as

$$\begin{aligned}
\Delta_{j,k} &= r_k^{\ell^{([m/w]-1-(k+1))w+t}} \\
&= r_k^{\ell^{(u-1-k)w+t}} \\
&= r_k^{\ell^{uw-kw-w+t}} \\
&= r_k^{\ell^{(m-t)-kw-w+t}} \\
&= r_k^{\ell^{m-w-kw}}
\end{aligned}$$

Recall that r_k has order ℓ^{m-kw} for $0 \leq k < u-1$ by construction and therefore $\Delta_{j,k}$ has the promised order ℓ^w.

With respect to the order of the rightmost leaf, notice that it can be defined as the first root of r_0 followed by the elimination of the first $(\lceil m/w \rceil - 1)$-th digits. Therefore it is explicitly given by

$$\begin{aligned}
r_0 \cdot g^{-\sum_{i=0}^{\lceil m/w \rceil - 2} d_i \ell^{iw}} &= r_0^{d_{\lceil m/w \rceil - 1} \cdot \ell^{(\lceil m/w \rceil - 1)w}} \\
&= r_0^{d_{\lceil m/w \rceil - 1} \cdot \ell^{w\lceil m/w \rceil - w}} \\
&= r_0^{d_{\lceil m/w \rceil - 1} \cdot \ell^{(m-t+w)-w}} \\
&= r_0^{d_{\lceil m/w \rceil - 1} \cdot \ell^{m-t}}
\end{aligned}$$

which has the desired order since r_0 has order ℓ^m. □