Quantum circuits for the CSIDH: optimizing quantum evaluation of isogenies

Daniel J. Bernstein[1], Tanja Lange[2], Chloe Martindale[2], Lorenz Panny[2]

[1] Department of Computer Science
University of Illinois at Chicago
Chicago, IL 60607–7045, USA
djb@cr.yp.to

[2] Department of Mathematics and Computer Science
Technische Universiteit Eindhoven
P.O. Box 513, 5600 MB Eindhoven, The Netherlands
tanja@hyperelliptic.org
chloemartindale@gmail.com
lorenz@yx7.cc

Abstract. Choosing safe post-quantum parameters for the new CSIDH isogeny-based key-exchange system requires concrete analysis of the cost of quantum attacks. The two main contributions to attack cost are the number of queries in hidden-shift algorithms and the cost of each query. This paper analyzes algorithms for each query, introducing several new speedups while showing that some previous claims were too optimistic for the attacker. This paper includes a full computer-verified simulation of its main algorithm down to the bit-operation level.

Keywords: Elliptic curves, isogenies, circuits, constant-time computation, reversible computation, quantum computation, cryptanalysis.

1 Introduction

Castryck, Lange, Martindale, Panny, and Renes recently introduced CSIDH [15], an isogeny-based key exchange that runs efficiently and permits non-interactive key exchange. Like the original CRS [20, 64, 68] isogeny-based cryptosystem, CSIDH has public keys and ciphertexts only about twice as large as traditional

[*] Author list in alphabetical order; see https://www.ams.org/profession/leaders/culture/CultureStatement04.pdf. This work was supported in part by the Commission of the European Communities through the Horizon 2020 program under project number 643161 (ECRYPT-NET), 645622 (PQCRYPTO), 645421 (ECRYPT-CSA), and CHIST-ERA USEIT (NWO project 651.002.004); the Netherlands Organisation for Scientific Research (NWO) under grants 628.001.028 (FASOR) and 639.073.005; and the U.S. National Science Foundation under grant 1314919. "Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation" (or other funding agencies). Permanent ID of this document: 9b88023d7d9ef3f55b11b6f009131c9f. Date of this document: 2018.10.31.

elliptic-curve keys and ciphertexts for a similar security level against all pre-quantum attacks known. CRS was accelerated recently by De Feo, Kieffer, and Smith [23]; CSIDH builds upon this and chooses curves in a different way, obtaining much better speed.

For comparison, the SIDH (and SIKE) isogeny-based cryptosystems [37, 22, 36] are somewhat faster than CSIDH, but they do not support non-interactive key exchange, and their public keys and ciphertexts are 6 times larger[3] than in CSIDH. Furthermore, there are concerns that the extra information in SIDH keys might allow attacks; see [58].

These SIDH disadvantages come from avoiding the commutative structure used in CRS and now in CSIDH. SIDH deliberately avoids this structure because the structure allows *quantum* attacks that asymptotically take subexponential time; see below. The CRS/CSIDH key size thus grows superlinearly in the post-quantum security level. For comparison, if the known attacks are optimal, then the SIDH key size grows linearly in the post-quantum security level.

However, even in a post-quantum world, it is not at all clear how much weight to put on these asymptotics. It is not clear, for example, how large the keys will have to be before the subexponential attacks begin to outperform the exponential-time non-quantum attacks or an exponential-time Grover search. It is not clear when the superlinear growth in CSIDH key sizes will outweigh the factor 6 mentioned above. For applications that need non-interactive key exchange in a post-quantum world, the SIDH/SIKE family is not an option, and it is important to understand what influence these attacks have upon CSIDH key sizes. The asymptotic performance of these attacks is stated in [15], but it is challenging to understand the concrete performance of these attacks for specific CSIDH parameters.

1.1. Contributions of this paper. The most important bottleneck in the quantum attacks mentioned above is the cost of evaluating a group action, a series of isogenies, in superposition. Each quantum attack incurs this cost many times; see below. The goals of this paper are to analyze and optimize this cost. We focus on CSIDH because CSIDH is much faster than CRS.

Our main result has the following shape: the CSIDH group action can be carried out in B nonlinear bit operations (counting ANDs and ORs, allowing free XORs and NOTs) with failure probability at most ϵ. (All of our algorithms know when they have failed.) This implies a reversible computation of the CSIDH group action with failure probability at most ϵ using at most $2B$ Toffoli gates (allowing free NOTs and CNOTs). This in turn implies a quantum computation of the CSIDH group action with failure probability at most ϵ using at most $14B$

[3] When the goal is for pre-quantum attacks to take 2^λ operations (without regard to memory consumption), CRS, CSIDH, SIDH, and SIKE all choose primes $p \approx 2^{4\lambda}$. The CRS and CSIDH keys and ciphertexts use (approximately) $\log_2 p \approx 4\lambda$ bits, whereas the SIDH and SIKE keys and ciphertexts use $6\log_2 p \approx 24\lambda$ bits for 3 elements of $\mathbb{F}_{p^2}$. There are compressed variants of SIDH that reduce $6\log_2 p$ to $4\log_2 p \approx 16\lambda$ (see [1]) and to $3.5\log_2 p \approx 14\lambda$ (see [19] and [75]), at some cost in run time.

T-gates (allowing free Clifford gates). Appendix A reviews these cost metrics and their relationships.

We explain how to compute pairs (B, ϵ) for any given CSIDH parameters. For example, we show how to compute CSIDH-512 for uniform random exponent vectors in $\{-5, \ldots, 5\}^{74}$ using

- $1118827416420 \approx 2^{40}$ nonlinear bit operations using the algorithm of Section 7, or
- $765325228976 \approx 0.7 \cdot 2^{40}$ nonlinear bit operations using the algorithm of Section 8,

in both cases with failure probability below 2^{-32}. CSIDH-512 is the smallest parameter set considered in [15]. For comparison, computing the same action with failure probability 2^{-32} using the Jao–LeGrow–Leonardi–Ruiz-Lopez algorithm [38], with the underlying modular multiplications computed by the same algorithm as in Roetteler–Naehrig–Svore–Lauter [63], would use approximately 2^{51} nonlinear bit operations.

We exploit a variety of algorithmic ideas, including several new ideas pushing beyond the previous state of the art in isogeny computation, with the goal of obtaining the best pairs (B, ϵ). We introduce a new constant-time variable-degree isogeny algorithm, a new application of the Elligator map, new ways to handle failures in isogeny computations, new combinations of the components of these computations, new speeds for integer multiplication, and more.

1.2. Impact upon quantum attacks. Kuperberg [46] introduced an algorithm using $\exp\big((\log N)^{1/2+o(1)}\big)$ queries and $\exp\big((\log N)^{1/2+o(1)}\big)$ operations on $\exp((\log N)^{1/2+o(1)})$ qubits to solve the order-N dihedral hidden-subgroup problem. Regev [61] introduced an algorithm using only a polynomial number of qubits, although with a worse $o(1)$ for the number of queries and operations. A followup paper by Kuperberg [47] introduced further algorithmic options.

Childs, Jao, and Soukharev [17] pointed out that these algorithms could be used to attack CRS. They analyzed the asymptotic cost of a variant of Regev's algorithm in this context. This cost is dominated by queries, in part because the number of queries is large but also because the cost of each query is large. Each query evaluates the CRS group action using a superposition of group elements.

We emphasize that computing the exact attack costs for any particular set of CRS or CSIDH parameters is complicated and requires a lot of new work. The main questions are (1) the exact number of queries for various dihedral-hidden-subgroup algorithms, not just asymptotics; and (2) the exact cost of each query, again not just asymptotics.

The first question is outside the scope of our paper. Some of the simpler algorithms were simulated for small sizes in [46], [10], and [11], but Kuperberg commented in [46, page 5] that his "experiments with this simulator led to a false conjecture for algorithm's precise query complexity".

Our paper addresses the second question for CSIDH: the concrete cost of quantum algorithms for evaluating the action of the class group, which means computing isogenies of elliptic curves in superposition.

1.3. Comparison to previous claims regarding query cost. Bonnetain and Schrottenloher claim in [11, online versions 4, 5, and 6] that CSIDH-512 can be broken in "only" 2^{71} quantum gates, where each query uses 2^{37} quantum gates ("Clifford+T" gates; see Appendix A.4).

We work in the same simplified model of counting operations, allowing any number of qubits to be stored for free. We further simplify by counting only T-gates. We gain considerable performance from optimizations not considered in [11]. We take the best possible distribution of input vectors, disregarding the 2^2 overhead estimated in [11]. Our final gate counts for each query are nevertheless much higher than the 2^{37} claimed in [11]. Even assuming that [11] is correct regarding the number of queries, the cost of each query pushes the total attack cost above 2^{80}.

The query-cost calculation in [11] is not given in enough detail for full reproducibility. However, some details are provided, and given these details we conclude that costly parts of the computation are overlooked in [11] in at least three ways. First, to estimate the number of quantum gates for multiplication in $\mathbb{F}_p$, [11] uses a count of nonlinear bit operations for multiplication in $\mathbb{F}_2[x]$, not noticing that all known methods for multiplication in $\mathbb{Z}$ (never mind reduction modulo p) involve many more nonlinear bit operations than multiplication in $\mathbb{F}_2[x]$. Second, at a higher level, the strategy for computing an ℓ-isogeny requires first finding a point of order ℓ, an important cost not noticed in [11]. Third, [11] counts the number of operations in a *branching* algorithm, not noticing the challenge of building a *non-branching* (constant-time) algorithm for the same task, as required for computations in superposition. Our analysis addresses all of these issues and more.

1.4. Memory consumption. We emphasize that our primary goal is to minimize the number of bit operations. This cost metric pays no attention to the fact that the resulting quantum algorithm for, e.g., CSIDH-512 uses a quantum computer with 2^{40} qubits.

Most of the quantum-algorithms literature pays much more attention to the number of qubits. This is why [17], for example, uses a Regev-type algorithm instead of Kuperberg's algorithm. Similarly, [15] takes Regev's algorithm "as a baseline" given "the larger memory requirement" for Kuperberg's algorithm.

An obvious reason to keep the number of qubits under control is the difficulty of scaling quantum computers up to a huge number of qubits. Post-quantum cryptography starts from the assumption that there will be enough scalability to build a quantum computer using thousands of logical qubits to run Shor's algorithm, but this does not imply that a quantum computer with millions of logical qubits will be only 1000 times as expensive, given limits on physical chip size and costs of splitting quantum computation across multiple chips.

On the other hand, [11] chooses Kuperberg's algorithm, and claims that the number of qubits used in Kuperberg's algorithm is not a problem:

> The algorithm we consider has a subexponential memory cost. More precisely, it needs exactly one qubit per query, plus the fixed overhead of the oracle, which can be neglected.

Concretely, for CSIDH-512, [11, online versions 1, 2, 3] claim $2^{29.5}$ qubits, and [11, online versions 4, 5, 6] claim 2^{31} qubits. However, no justification is provided for the claim that the number of qubits for the oracle "can be neglected". There is no analysis in [11] of the number of qubits used for the oracle.

We are not saying that our techniques *need* 2^{40} qubits. On the contrary: later we mention various ways that the number of qubits can be reduced with only moderate costs in the number of operations. However, one cannot trivially extrapolate from the memory consumption of CSIDH software (a few kilobytes) to the number of qubits used in a quantum computation. The requirement of reversibility makes it more challenging and more expensive to reduce space, since intermediate results cannot simply be erased. See Appendix A.3.

Furthermore, even if enough qubits are available, simply counting qubit operations ignores critical bottlenecks in quantum computation. Fault-tolerant quantum computation corrects errors in every qubit at every time step, even if the qubit is merely being stored; see Appendix A.5. Communicating across many qubits imposes further costs; see Appendix A.6. It is thus safe to predict that the actual cost of a quantum CSIDH query will be much larger than indicated by our operation counts. Presumably the gap will be larger than the gap for, e.g., the AES attack in [28], which has far fewer idle qubits and much less communication overhead.

1.5. Acknowledgments. Thanks to Bo-Yin Yang for suggesting factoring the average over vectors of the generating function in Section 7.3. Thanks to Joost Renes for his comments.

2 Overview of the computation

We recall the definition of the CSIDH group action, focusing on the computational aspects of the concrete construction rather than discussing the general case of the underlying algebraic theory.

Parameters. The only parameter in CSIDH is a prime number p of the form $p = 4 \cdot \ell_1 \cdots \ell_n - 1$, where $\ell_1 < \cdots < \ell_n$ are (small) odd primes and $n \geq 1$. Note that $p \equiv 3 \pmod 8$ and $p > 3$.

Notation. For each $A \in \mathbb{F}_p$ with $A^2 \neq 4$, define E_A as the Montgomery curve $y^2 = x^3 + Ax^2 + x$ over $\mathbb{F}_p$. This curve E_A is supersingular, meaning that $\#E_A(\mathbb{F}_p) \equiv 1 \pmod p$, if and only if it has trace zero, meaning that $\#E_A(\mathbb{F}_p) = p + 1$. Here $E_A(\mathbb{F}_p)$ means the group of points of E_A with coordinates in $\mathbb{F}_p$, including the neutral element at ∞; and $\#E_A(\mathbb{F}_p)$ means the number of points.

Define S_p as the set of A such that E_A is supersingular. For each $A \in S_p$ and each $i \in \{1, \ldots, n\}$, there is a unique $B \in S_p$ such that there is an ℓ_i-isogeny from E_A to E_B whose kernel is $E_A(\mathbb{F}_p)[\ell_i]$, the set of points $Q \in E_A(\mathbb{F}_p)$ with $\ell_i Q = 0$. Define $\mathcal{L}_i(A) = B$. One can show that $\mathcal{L}_i$ is invertible: specifically, $\mathcal{L}_i^{-1}(A) = -\mathcal{L}_i(-A)$. Hence $\mathcal{L}_i^e$ is defined for each integer e.

Inputs and output. Given an element $A \in S_p$ and a list $(e_1, \ldots, e_n)$ of integers, the CSIDH group action computes $\mathcal{L}_1^{e_1}(\mathcal{L}_2^{e_2}(\cdots(\mathcal{L}_n^{e_n}(A))\cdots)) \in S_p$.

2.1. Distribution of exponents. The performance of our algorithms depends on the distribution of the exponent vectors $(e_1, \ldots, e_n)$, which in turn depends on the context.

Constructively, [15] proposes to sample each e_i independently and uniformly from a small range $\{-C, \ldots, C\}$. For example, CSIDH-512 in [15] has $n = 74$ and uses the range $\{-5, \ldots, 5\}$, so there are $11^{74} \approx 2^{256}$ equally likely exponent vectors. We emphasize, however, that all known attacks actually use considerably larger exponent vectors. This means that the distribution of exponents $(e_1, \ldots, e_n)$ our quantum oracle has to process is *not* the same as the distribution used constructively.

The first step in the algorithms of Kuperberg and Regev, applied to a finite abelian group G, is to generate a uniform superposition over all elements of G. CRS and CSIDH define a map from vectors $(e_1, \ldots, e_n)$ to elements $\mathfrak{l}_1^{e_1} \cdots \mathfrak{l}_n^{e_n}$ of the ideal-class group G. This map has a high chance of being surjective but it is far from injective: its kernel is a lattice of rank n. Presumably taking, e.g., 17^{74} length-74 vectors with entries in the range $\{-8, \ldots, 8\}$ produces a close-to-uniform distribution of elements of the CSIDH-512 class group, but the literature does not indicate how Kuperberg's algorithm behaves when each group element is represented as many different strings.

In his original paper on CRS, Couveignes [20] suggested instead generating a unique vector representing each group element as follows. Compute a basis for the lattice mentioned above; on a quantum computer this can be done using Shor's algorithm [67] which runs in polynomial time, and on a conventional computer this can be done using Hafner and McCurley's algorithm [29] which runs in subexponential time. This basis reveals the group size $\#G$ and an easy-to-sample set R of representatives for G, such as $\{(e_1, 0, \ldots, 0) : 0 \le e_1 < \#G\}$ in the special case that $\mathfrak{l}_1$ generates G; for the general case see, e.g., [50, Section 4.1]. Reduce each representative to a short representative, using an algorithm that finds a close lattice vector. If this algorithm is deterministic (for example, if all randomness used in the algorithm is replaced by pseudorandomness generated from the input) then applying it to a uniform superposition over R produces a uniform superposition over a set of short vectors uniquely representing G.

The same idea was mentioned in the Childs–Jao–Soukharev paper [17] on quantum attacks against CRS, and in the description of quantum attacks in the CSIDH paper. However, close-vector problems are not easy, even in dimensions as small as 74. Bonnetain and Schrottenloher [11] estimate that CSIDH-512 exponent vectors can be found whose 1-norm is 4 times larger than vectors used constructively. They rely on a very large precomputation, and they do not justify their assumption that the 1-norm, rather than the ∞-norm, measures the cost of a class-group action in superposition. Jao, LeGrow, Leonardi, and Ruiz-Lopez [38] present an algorithm that guarantees $(\log p)^{O(1)}$ bits in each exponent, i.e., in the ∞-norm, but this also requires a subexponential-time precomputation, and the exponents appear to be rather large.

Perhaps future research will improve the picture of how much precomputation time and per-vector computation time is required for algorithms that find vectors of a specified size; or, alternatively, will show that Kuperberg-type algorithms can handle non-unique representatives of group elements. The best conceivable case for the attacker is the distribution used in CSIDH itself, and we choose this distribution as an illustration in analyzing the concrete cost of our algorithms.

2.2. Verification of costs. To ensure that we are correctly computing the number of bit operations in our group-action algorithms, we have built a bit-operation simulator, and implemented our algorithms inside the simulator. The simulator is available for free from https://quantum.isogeny.org/software.html.

The simulator has a very small core that implements—and counts the number of—NOT, XOR, AND, and OR operations. Higher-level algorithms, from basic integer arithmetic up through isogeny computation, are built on top of this core.

The core also encapsulates the values of bits so that higher-level algorithms do not accidentally inspect those values. There is an explicit mechanism to break the encapsulation so that output values can be checked against separate computations in the Sage computer-algebra system.

2.3. Verification of failure probabilities. Internally, each of our group-action algorithms moves the exponent vector $(e_1, \ldots, e_n)$ step by step towards 0. The algorithm fails if the vector does not reach 0 within the specified number of iterations. Analyzing the failure probability requires analyzing how the distribution of exponent vectors interacts with the distribution of curve points produced inside the algorithm; each e_i step relies on finding a point of order ℓ_i.

We mathematically calculate the failure probability in a model where each generated curve point has probability $1 - 1/\ell_i$ of having order divisible by ℓ_i, and where these probabilities are all independent. The model would be exactly correct if each point were generated independently and uniformly at random. We actually generate points differently, so there is a risk of our failure-probability calculations being corrupted by inaccuracies in the model. To address this risk, we have carried out various point-generation experiments, suggesting that the model is reasonably accurate. Even if the model is inaccurate, one can compensate with a minor increase in costs. See Sections 4.3 and 5.2.

There is a more serious risk of errors in the failure-probability calculations that we carry out within the model. To reduce this risk, we have carried out 10^7 simple trials of the following type for each algorithm: generate a random exponent vector, move it step by step towards 0 the same way the algorithm does (in the model), and see how many iterations are required. The observed distribution of the number of iterations is consistent with the distribution that we calculate mathematically. Of course, if there is a calculation error that somehow affects only very small probabilities, then this error will not be caught by only 10^7 experiments.

2.4. Structure of the computation. We present our algorithms from bottom up, starting with scalar multiplication in Section 3, generation of curve points in Section 4, computation of $\mathcal{L}_i$ in Section 5, and computation of the entire CSIDH group action in Sections 6, 7, and 8. Lower-level subroutines for basic integer arithmetic and modular arithmetic appear in Appendix B and Appendix C respectively.

Various sections and subsections mention ideas for saving time beyond what we have implemented in our bit-operation simulator. These ideas include low-level speedups such as avoiding exponentiations in inversions and Legendre-symbol computations (see Appendix C.4), and higher-level speedups such as using division polynomials (Section 9) and/or modular polynomials (Section 10) to eliminate failures for small primes. All of the specific bit-operation counts that we state, such as the 1118827416420 nonlinear bit operations mentioned above, are fully implemented.

3 Scalar multiplication on an elliptic curve

This section analyzes the costs of scalar multiplication on the curves used in CSIDH, supersingular Montgomery curves $E_A : y^2 = x^3 + Ax^2 + x$ over $\mathbb{F}_p$.

For CSIDH-512, our simulator shows (after our detailed optimizations; see Appendices B and C) that a squaring $\mathbf{S}$ in $\mathbb{F}_p$ can be computed in 349596 nonlinear bit operations, and that a general multiplication $\mathbf{M}$ in $\mathbb{F}_p$ can be computed in 447902 nonlinear bit operations, while addition in $\mathbb{F}_p$ takes only 2044 nonlinear bit operations. We thus emphasize the number of $\mathbf{S}$ and $\mathbf{M}$ in scalar multiplication (and in higher-level operations), although in our simulator we have also taken various opportunities to eliminate unnecessary additions and subtractions.

3.1. How curves are represented. We consider two options for representing E_A. The **affine** option uses $A \in \mathbb{F}_p$ to represent E_A. The **projective** option uses $A_0, A_1 \in \mathbb{F}_p$, with $A_0 \neq 0$, to represent E_A where $A = A_1/A_0$.

The formulas to produce a curve in Section 5 naturally produce (A_0, A_1) in projective form. Dividing A_1 by A_0 to produce A in affine form costs an inversion and a multiplication. Staying in projective form is an example of what Appendix C.5 calls "eliminating inversions", but this requires some extra computation when A is used, as we explain below.

The definition of the class-group action requires producing the output A in affine form at the end of the computation. It could also be beneficial to convert each intermediate A to affine form, depending on the relative costs of the inversion and the extra computation.

3.2. How points are represented. As in [51, page 425, last paragraph] and [53, page 261], we avoid computing the y-coordinate of a point (x, y) on E_A. This creates some ambiguity, since the points (x, y) and $(x, -y)$ are both represented as $x \in \mathbb{F}_p$, but the ambiguity does not interfere with scalar multiplication.

We again distinguish between affine and projective representations. As in [5], we represent both $(0, 0)$ and the neutral element on E_A as $x = 0$, and (except where otherwise noted) we allow $X/0$, including $0/0$, as a projective representation of $x = 0$. The projective representation thus uses $X, Z \in \mathbb{F}_p$ to represent $x = X/Z$ if $Z \neq 0$, or $x = 0$ if $Z = 0$. These definitions eliminate branches from the scalar-multiplication techniques that we use.

3.3. Computing nP. We use the Montgomery ladder to compute nP, given a b-bit exponent n and a curve point P. The Montgomery ladder consists of b "ladder steps" operating on variables (X_2, Z_2, X_3, Z_3) initialized to $(1, 0, x_1, 1)$, where x_1 is the x-coordinate of P. Each ladder step works as follows:

- Conditionally swap (X_2, Z_2) with (X_3, Z_3), where the condition bit in iteration i is bit n_{b-1-i} of n. This means computing $X_2 \oplus X_3$, ANDing each bit with the condition bit, and XORing the result into both X_2 and X_3; and similarly for Z_2 and Z_3.
- Compute $Y = X_2 - Z_2$, Y^2, $T = X_2 + Z_2$, T^2, $X_4 = T^2Y^2$, $E = T^2 - Y^2$, and $Z_4 = E(Y^2 + ((A+2)/4)E)$. This is a **point doubling**: it uses $2\mathbf{S} + 3\mathbf{M}$ and a few additions (counting subtractions as additions). We divide $A+2$ by 4 modulo p before the scalar multiplication, using two conditional additions of p and two shifts.
- Compute $C = X_3 + Z_3$, $D = X_3 - Z_3$, DT, CY, $X_5 = (DT + CY)^2$, and $Z_5 = x_1(DT - CY)^2$. This is a **differential addition**: it also uses $2\mathbf{S} + 3\mathbf{M}$ and a few additions.
- Set $(X_2, Z_2, X_3, Z_3) \leftarrow (X_4, Z_4, X_5, Z_5)$.
- Conditionally swap (X_2, Z_2) with (X_3, Z_3), where the condition bit is again n_{b-1-i}. We merge this conditional swap with the conditional swap at the beginning of the next iteration by using $n_{b-i-i} \oplus n_{b-i-2}$ as condition bit.

Then nP has projective representation (X_2, Z_2) by [9, Theorem 4.5]. The overall cost is $4b\mathbf{S} + 6b\mathbf{M}$ plus a small overhead for additions and conditional swaps.

Representing the input point projectively as X_1/Z_1 means computing $X_5 = Z_1(DT + CY)^2$ and $Z_5 = X_1(DT - CY)^2$, and starting from $(1, 0, X_1, Z_1)$. This costs $b\mathbf{M}$ extra. Beware that [9, Theorem 4.5] requires $Z_1 \neq 0$.

Similarly, representing A projectively as A_1/A_0 means computing $X_4 = T^2(4A_0Y^2)$ and $Z_4 = E(4A_0Y^2 + (A_1 + 2A_0)E)$, after multiplying Y^2 by $4A_0$. This also costs $b\mathbf{M}$ extra.

Other techniques. The initial $Z_2 = 0$ and $Z_3 = 1$ (for an affine input point) are small, and remain small after the first conditional swap, saving time in the next additions and subtractions. Our framework for tracking sizes of integers recognizes this automatically. The framework does not, however, recognize that half of the output of the last conditional swap is unused. We could save some bit operations by incorporating dead-value elimination and other standard peephole optimizations.

Montgomery [53, page 260] considered carrying out many scalar multiplications at once, using affine coordinates for intermediate points inside each scalar multiplication (e.g., $x_2 = X_2/Z_2$), and batching inversions across the scalar multiplications. This could be slightly less expensive than the Montgomery ladder for large b, depending on the $\mathbf{S}/\mathbf{M}$ ratio. Our computation of a CSIDH group action involves many scalar multiplications, but not in large enough batches to justify considering affine coordinates for intermediate points. Computing the group action for a batch of inputs might change the picture, but for simplicity we focus on the problem of computing the group action for one input.

A more recent possibility is scalar multiplication on a birationally equivalent Edwards curve. Sliding-window Edwards scalar multiplication is somewhat less expensive than the Montgomery ladder for large b; see generally [8] and [34]. On the other hand, for constant-time computations it is important to use fixed windows rather than sliding windows. Despite this difficulty, we estimate that small speedups are possible for $b = 512$.

3.4. Computing $\boldsymbol{P, 2P, 3P, \dots, kP}$. An important subroutine in isogeny computation (see Section 5) is to compute the sequence $P, 2P, 3P, \dots, kP$ for a constant $k \geq 1$.

We compute $2P$ by a doubling, $3P$ by a differential addition, $4P$ by a doubling, $5P$ by a differential addition, $6P$ by a doubling, etc. In other words, each multiple of P is computed by the Montgomery ladder as above, but these computations are merged across the multiples (and conditional swaps are eliminated). This takes $2(k-1)\mathbf{S} + 3(k-1)\mathbf{M}$ for affine P and affine A. Projective P adds $\lfloor (k-1)/2 \rfloor \mathbf{M}$, and projective A adds $\lfloor k/2 \rfloor \mathbf{M}$.

We could instead compute $2P$ by a doubling, $3P$ by a differential addition, $4P$ by a differential addition, $5P$ by a differential addition, $6P$ by a differential addition, etc. This again takes $2(k-1)\mathbf{S} + 3(k-1)\mathbf{M}$ for affine P and affine A, but projective P and projective A now have different effects: projective P adds $(k-2)\mathbf{M}$ if $k \geq 2$, and projective A adds $\mathbf{M}$ if $k \geq 2$. The choice here also has an impact on metrics beyond bit operations: doublings increase space requirements but allow more parallelism.

4 Generating points on an elliptic curve

This section analyzes the cost of several methods to generate a random point on a supersingular Montgomery curve $E_A : y^2 = x^3 + Ax^2 + x$, given $A \in \mathbb{F}_p$. As in Section 2, p is a standard prime congruent to 3 modulo 8.

Sometimes one instead wants to generate a point on the twist of the curve. The **twist** is the curve $-y^2 = x^3 + Ax^2 + x$ over $\mathbb{F}_p$; note that -1 is a non-square in $\mathbb{F}_p$. This curve is isomorphic to E_{-A} by the map $(x, y) \mapsto (-x, y)$. Beware that there are several slightly different concepts of "twist" in the literature; the definition here is the most useful definition for CSIDH, as explained in [15].

4.1. Random point on curve or twist. The conventional approach is as follows: generate a uniform random $x \in \mathbb{F}_p$; compute $x^3 + Ax^2 + x$; compute $y = (x^3 + Ax^2 + x)^{(p+1)/4}$; and check that $y^2 = x^3 + Ax^2 + x$.

One always has $y^4 = (x^3 + Ax^2 + x)^{p+1} = (x^3 + Ax^2 + x)^2$ so $\pm y^2 = x^3 + Ax^2 + x$. About half the time, y^2 will match $x^3 + Ax^2 + x$; i.e., (x, y) will be a point on the curve. Otherwise (x, y) will be a point on the twist.

Since we work purely with x-coordinates (see Section 3.2), we skip the computation of y. However, we still need to know whether we have a curve point or a twist point, so we compute the Legendre symbol of $x^3 + Ax^2 + x$ as explained in Appendix C.4.

The easiest distribution of outputs to mathematically analyze is the uniform distribution over the following $p + 1$ pairs:

- $(x, +1)$ where x represents a curve point;
- $(x, -1)$ where x represents a twist point.

One can generate outputs from this distribution as follows: generate a uniform random $u \in \mathbb{F}_p \cup \{\infty\}$; set x to u if $u \in \mathbb{F}_p$ or to 0 if $u = \infty$; compute the Legendre symbol of $x^3 + Ax^2 + x$; and replace symbol 0 with $+1$ if $u = 0$ or -1 if $u = \infty$.

For computation it is slightly simpler to eliminate the two pairs with $x = 0$: generate a uniform random $x \in \mathbb{F}_p^*$ and compute the Legendre symbol of $x^3 + Ax^2 + x$. This generates a uniform distribution over the remaining $p - 1$ pairs.

4.2. Random point on curve.

What if twist points are useless and the goal is to produce a point specifically on the curve (or vice versa)? One approach is to generate, e.g., 100 random curve-or-twist points as in Section 4.1, and select the first point on the curve. This fails with probability $1/2^{100}$. If a computation involves generating 2^{10} points in this way then the overall failure probability is $1 - (1 - 1/2^{100})^{2^{10}} \approx 1/2^{90}$. One can tune the number of generated points according to the required failure probability.

We save time by applying "Elligator" [7], specifically the Elligator 2 map. Elligator 2 is defined for all the curves E_A that we use, *except* the curve E_0, which we discuss below. For each of these curves E_A, Elligator 2 is a fast injective map from $\{2, 3, \ldots, (p - 1)/2\}$ to the set $E_A(\mathbb{F}_p)$ of curve points. This produces only about half of the curve points; see Section 5.2 for analysis of the impact of this nonuniformity upon our higher-level algorithms.

Here are the details of Elligator 2, specialized to these curves, further simplified to avoid computing y, and adapted to allow twists as an option:

- Input $A \in \mathbb{F}_p$ with $A^2 \neq 4$ and $A \neq 0$.
- Input $s \in \{1, -1\}$. This procedure generates a point on E_A if $s = 1$, or on the twist of E_A if $s = -1$.
- Input $u \in \{2, 3, \ldots, (p - 1)/2\}$.
- Compute $v = A/(u^2 - 1)$.
- Compute e, the Legendre symbol of $v^3 + Av^2 + v$.
- Compute x as v if $e = s$, otherwise $-v - A$.

To see that this works, note first that v is defined since $u^2 \neq 1$, and is nonzero since $A \neq 0$. One can also show that $A^2 - 4$ is nonsquare for all of the CSIDH curves, so $v^3 + Av^2 + v \neq 0$, so e is 1 or -1. If $e = s$ then $x = v$ so $x^3 + Ax^2 + x$ is a square for $s = 1$ and a nonsquare for $s = -1$. Otherwise $e = -s$ and $x = -v - A$ so $x^3 + Ax^2 + x = -u^2(v^3 + Av^2 + v)$, which is a square for $s = 1$ and a nonsquare for $s = -1$. This uses that v and $-v - A$ satisfy $(-v - A)^2 + A(-v - A) + 1 = v^2 + Av + 1$ and $-v - A = -u^2 v$.

The $(p - 3)/2$ different choices of u produce $(p - 3)/2$ different curve points, but we could produce any particular x output twice since we suppress y.

The case $A = 0$. One way to extend Elligator 2 to E_0 is to set $v = u$ when $A = 0$ instead of $v = -A/(1 - u^2)$. The point of the construction of v is that

x^3+Ax^2+x for $x = -v-A$ is a non-square times v^3+Av^2+v, i.e., that $(-v-A)/v$ is a non-square; this is automatic for $A = 0$, since -1 is a non-square.

We actually handle E_0 in a different way: we precompute a particular base point on E_0 whose order is divisible by $(p + 1)/4$, and we always return this point if $A = 0$. This makes our higher-level algorithms slightly more effective (but we disregard this improvement in analyzing the success probability of our algorithms), since this point guarantees a successful isogeny computation starting from E_0; see Section 5. The same guarantee removes any need to generate other points on E_0, and is also useful to start walks in Section 10.

4.3. Derandomization. Rather than generating random points, we generate a deterministic sequence of points by taking $u = 2$ for the first point, $u = 3$ for the next point, etc. We precompute the inverses of $1 - 2^2$, $1 - 3^2$, etc., saving bit operations.

An alternative, saving the same number of bit operations, is to precompute inverses of $1 - u^2$ for various random choices of u, embedding the inverses into the algorithm. This guarantees that the failure probability of the outer algorithm for any particular input A, as the choices of u vary, is the same as the failure probability of an algorithm that randomly chooses u upon demand for each A.

We are heuristically assuming that failures are not noticeably correlated across choices of A. To replace this heuristic with a proof, one can generate the u sequence randomly for each input. This randomness, in turn, is indistinguishable from the output of a cipher, under the assumption that the cipher is secure. In this setting one cannot precompute the reciprocals of $1 - u^2$, but one can still batch the inversions.

5 Computing an ℓ-isogenous curve

This section analyzes the cost of computing a single isogeny in CSIDH. There are two inputs: A, specifying a supersingular Montgomery curve E_A over $\mathbb{F}_p$; and i, specifying one of the odd prime factors ℓ_i of $(p + 1)/4 = \ell_1 \cdots \ell_n$. The output is $B = \mathcal{L}_i(A)$. We abbreviate ℓ_i as ℓ and $\mathcal{L}_i$ as $\mathcal{L}$.

Recall that B is characterized by the following property: there is an ℓ-isogeny from E_A to E_B whose kernel is $E_A(\mathbb{F}_p)[\ell]$. Beyond analyzing the costs of computing $B = \mathcal{L}(A)$, we analyze the costs of applying the ℓ-isogeny to a point on E_A, obtaining a point on E_B. See Section 5.4.

The basic reason that CSIDH is much faster than CRS is that the CSIDH construction allows (variants of) Vélu's formulas [72, 18, 62] to use points in $E_A(\mathbb{F}_p)$, rather than points defined over larger extension fields. This section focuses on computing B via these formulas. The cost of these formulas is approximately linear in ℓ, assuming that a point of order ℓ is known. There are two important caveats here:

- Finding a point of order ℓ is not easy to do in constant time. See Section 5.1. We follow the obvious approach, namely taking an appropriate multiple of a random point; but this is expensive—recall from Section 3 that a 500-bit

Montgomery ladder costs $2000\mathbf{S} + 3000\mathbf{M}$ when A and the input point are affine—and has failure probability approximately $1/\ell$.
- In some of our higher-level algorithms, i is a *variable*. Then $\ell = \ell_i$ is also a variable, and Vélu's formulas are variable-time formulas, while we need constant-time computations. Generic branch elimination produces a constant-time computation taking time approximately linear in $\ell_1 + \ell_2 + \cdots + \ell_n$, which is quite slow. However, we show how to do much better, reducing $\ell_1 + \ell_2 + \cdots + \ell_n$ to $\max\{\ell_1, \ell_2, \ldots, \ell_n\}$, by exploiting the internal structure of Vélu's formulas. See Section 5.3.

There are other ways to compute isogenies, as explored in [42, 23]:

- The "Kohel" strategy: Compute a univariate polynomial whose roots are the x-coordinates of the points in $E_A(\mathbb{F}_p)[\ell]$. Use Kohel's algorithm [45, Section 2.4], which computes an isogeny given this polynomial. This strategy is (for CSIDH) asymptotically slower than Vélu's formulas, but could nevertheless be faster when ℓ is very small. Furthermore, this strategy is deterministic and always works.
- The "modular" strategy: Compute the possible j-invariants of E_B by factoring modular polynomials. Determine the correct choice of B by computing the corresponding isogeny kernels or, on subsequent steps, simply by not walking back.

We analyze the Kohel strategy in Section 9, and the modular strategy in Section 10.

5.1. Finding a point of order ℓ. We now focus on the problem of finding a point of order ℓ in $E_A(\mathbb{F}_p)$. By assumption $(p + 1)/4$ is a product of distinct odd primes $\ell_1, \ldots, \ell_n$; $\ell = \ell_i$ is one of those primes; and $\#E_A(\mathbb{F}_p) = p + 1$. One can show that $E_A(\mathbb{F}_p)$ has a point of order 4 and is thus cyclic:

$$E_A(\mathbb{F}_p) \cong \mathbb{Z}/(p + 1) \cong \mathbb{Z}/4 \times \mathbb{Z}/\ell_1 \times \cdots \times \mathbb{Z}/\ell_n\,.$$

We *try* to find a point Q of order ℓ in $E_A(\mathbb{F}_p)$ as follows:

- Pick a random point $P \in E_A(\mathbb{F}_p)$, as explained in Section 4.
- Compute a "cofactor" $(p + 1)/\ell$. To handle the case $\ell = \ell_i$ for variable i, we first use bit operations to compute the list $\ell'_1, \ldots, \ell'_n$, where $\ell'_j = \ell_j$ for $j \neq i$ and $\ell'_i = 1$; we then use a product tree to compute $\ell'_1 \cdots \ell'_n$. (Computing $(p + 1)/\ell$ by a general division algorithm could be faster, but the product tree is simpler and has negligible cost in context.)
- Compute $Q = ((p + 1)/\ell)P$ as explained in Section 3.

If P is a uniform random element of $E_A(\mathbb{F}_p)$ then Q is a uniform random element of $E_A(\mathbb{F}_p)[\ell] \cong \mathbb{Z}/\ell$. The order of Q is thus the desired ℓ with probability $1 - 1/\ell$. Otherwise Q is ∞, the neutral element on the curve, which is represented by $x = 0$. Checking for $x = 0$ is a reliable way to detect this case: the only other point represented by $x = 0$ is $(0, 0)$, which is outside $E_A(\mathbb{F}_p)[\ell]$.

Different concepts of constant time. Beware that there are two different notions of "constant time" for cryptographic algorithms. One notion is that the time for each operation is independent of *secrets*. This notion allows the CSIDH user to generate a uniform random element of $E_A(\mathbb{F}_p)[\ell]$ and try again if the point is ∞, guaranteeing success with an average of $\ell/(\ell-1)$ tries. The time varies, but the variation is independent of the secret A.

A stricter notion is that the time for each operation is independent of *all* inputs. The time depends on parameters, such as p in CSIDH, but does not depend on random choices. We emphasize that a quantum circuit operating on many inputs in superposition is, by definition, using this stricter notion. We thus choose the sequence of operations carried out by the circuit, and analyze the probability that this sequence fails.

Amplifying the success probability. Having each 3-isogeny fail with probability $1/3$, each 5-isogeny fail with probability $1/5$, etc. creates a correctness challenge for higher-level algorithms that compute many isogenies.

A simple workaround is to generate many points $Q_1, Q_2, \ldots, Q_N$, and use bit operations on the points to select the first point with $x \neq 0$. This fails if all of the points have $x = 0$. Independent uniform random points have overall failure probability $1/\ell^N$. One can make $1/\ell^N$ arbitrarily small by choosing N large enough: for example, $1/3^N$ is below $1/2^{32}$ for $N \geq 21$, and is below $1/2^{256}$ for $N \geq 162$.

We return to the costs of generating so many points, and the costs of more sophisticated alternatives, when we analyze algorithms to compute the CSIDH group action.

5.2. Nonuniform distribution of points. We actually generate random points using Elligator (see Section 4.2), which generates only $(p-3)/2$ different curve points P. At most $(p+1)/\ell$ of these points produce $Q = \infty$, so the failure chance is at most $(2/\ell)(p+1)/(p-3) \approx 2/\ell$.

This bound cannot be simultaneously tight for $\ell = 3$, $\ell = 5$, and $\ell = 7$ (assuming $3 \cdot 5 \cdot 7$ divides $p+1$): if it were then the Elligator outputs would include all points having orders dividing $(p+1)/3$ or $(p+1)/5$ or $(p+1)/7$, but this accounts for more than 54% of all curve points, contradiction.

Points generated by Elligator actually appear to be much better distributed modulo each ℓ, with failure chance almost exactly $1/\ell$. Experiments support this conjecture. Readers concerned with the gap between the provable $2/\ell$ and the heuristic $1/\ell$ may prefer to add or subtract a few Elligator 2 outputs, obtaining a distribution provably close to uniform (see [70]) at a moderate cost in performance. A more efficient approach is to accept a doubling of failure probability and use a small number of extra iterations to compensate.

We shall later see other methods of obtaining rational ℓ-torsion points, e.g., by pushing points through ℓ'-isogenies. This does not make a difference in the analysis of failure probabilities.

For comparison, generating a random point on the curve or twist (see Section 4.1) has failure probability above $1/2$ at finding a curve point of order ℓ. See Section 6.2 for the impact of this difference upon higher-level algorithms.

5.3. Computing an ℓ-isogenous curve from a point of order ℓ. Once we have the x-coordinate of a point Q of order ℓ in $E_A(\mathbb{F}_p)$, we compute the x-coordinates of the points $Q, 2Q, 3Q, \ldots, ((\ell-1)/2)Q$. We use this information to compute $B = \mathcal{L}(A)$, the coefficient determining the ℓ-isogenous curve E_B.

Recall from Section 3.4 that computing $Q, 2Q, 3Q, \ldots, ((\ell-1)/2)Q$ costs $(\ell-3)\mathbf{S} + 1.5(\ell-3)\mathbf{M}$ for affine Q and affine A, and just $1\mathbf{M}$ extra for affine Q and projective A. The original CSIDH paper [15] took more time here, namely $(\ell-3)\mathbf{S} + 2(\ell-3)\mathbf{M}$, to handle projective Q and projective A. We decide, based on comparing ℓ to the cost of an inversion, whether to spend an inversion converting Q to affine coordinates.

Given the x-coordinates of $Q, 2Q, 3Q, \ldots, ((\ell-1)/2)Q$, the original CSIDH paper [15] took approximately $3\ell\mathbf{M}$ to compute B. Meyer and Reith [49] pointed out that CSIDH benefits from Edwards-coordinate isogeny formulas from Moody and Shumow [54]; we reuse this speedup. These formulas work as follows:

- Compute $a = A + 2$ and $d = A - 2$.
- Compute the Edwards y-coordinates of $Q, 2Q, 3Q, \ldots, ((\ell-1)/2)Q$. The Edwards y-coordinate is related to the Montgomery x-coordinate by $y = (x-1)/(x+1)$. We are given each x projectively as X/Z, and compute y projectively as Y/T where $Y = X - Z$ and $T = X + Z$. Note that Y and T naturally occur as intermediate values in the Montgomery ladder.
- Compute the product of these y-coordinates: i.e., compute $\prod Y$ and $\prod T$. This uses a total of $(\ell-3)\mathbf{M}$.
- Compute $a' = a^\ell (\prod T)^8$ and $d' = d^\ell (\prod Y)^8$. Each ℓth power takes a logarithmic number of squarings and multiplications; see Appendix C.4.
- Compute, projectively, $B = 2(a'+d')/(a'-d')$. Subsequent computations decide whether to convert B to affine form.

These formulas are almost three times faster than the formulas used in [15]. The total cost of computing B from Q is almost two times faster than in [15].

Handling variable ℓ. We point out that the isogeny computations for $\ell = 3$, $\ell = 5$, $\ell = 7$, etc. have a Matryoshka-doll structure, allowing a constant-time computation to handle many different values of ℓ with essentially the same cost as a single computation for the largest value of ℓ.

Concretely, the following procedure takes approximately $\ell_n\mathbf{S} + 2.5\ell_n\mathbf{M}$, and allows any $\ell \le \ell_n$. If the context places a smaller upper bound upon ℓ then one can replace ℓ_n with that upper bound, saving time; we return to this idea later.

Compute the Montgomery x-coordinates and the Edwards y-coordinates of $Q, 2Q, 3Q, \ldots, ((\ell_n-1)/2)Q$. Use bit operations to replace each Edwards y-coordinate with 1 after the first $(\ell-1)/2$ points. Compute the product of these modified y-coordinates; this is the desired product of the Edwards y-coordinates of the first $(\ell-1)/2$ points. Finish computing B as above. Note that the exponentiation algorithm in Appendix C.4 allows variable ℓ.

5.4. Applying an ℓ-isogeny to a point. The following formulas define an ℓ-isogeny from E_A to E_B with kernel $E_A(\mathbb{F}_p)[\ell]$. The x-coordinate of the image of a point $P_1 \in E_A(\mathbb{F}_p)$ under this isogeny is

$$x(P_1) \prod_{j \in \{1,2,\ldots,(\ell-1)/2\}} \left(\frac{x(P_1)x(jQ) - 1}{x(P_1) - x(jQ)} \right)^2.$$

Each $x(jQ)$ appearing here was computed above in projective form X/Z. The ratio $(x(P_1)x(jQ) - 1)/(x(P_1) - x(jQ))$ is $(x(P_1)X - Z)/(x(P_1)Z - X)$. This takes $2\mathbf{M}$ to compute projectively if $x(P_1)$ is affine, and thus $(\ell-1)\mathbf{M}$ across all j. Multiplying the numerators takes $((\ell-3)/2)\mathbf{M}$, multiplying the denominators takes $((\ell-3)/2)\mathbf{M}$, squaring both takes $2\mathbf{S}$, and multiplying by $x(P_1)$ takes $1\mathbf{M}$, for a total of $(2\ell-3)\mathbf{M} + 2\mathbf{S}$.

If $x(P_1)$ is instead given in projective form as X_1/Z_1 then computing $X_1X - Z_1Z$ and $X_1Z - Z_1X$ might seem to take $4\mathbf{M}$, but one can instead compute the sum and difference of $(X_1 - Z_1)(X + Z)$ and $(X_1 + Z_1)(X - Z)$, using just $2\mathbf{M}$. The only extra cost compared to the affine case is four extra additions. This speedup was pointed out by Montgomery [53] in the context of the Montgomery ladder. The initial CSIDH software accompanying [15] did not use this speedup but [49] mentioned the applicability to CSIDH.

In the opposite direction, if inversion is cheap enough to justify making $x(P_1)$ *and* every $x(jQ)$ affine, then $2\mathbf{M}$ drops to $1\mathbf{M}$, and the total cost drops to approximately $1.5\ell\mathbf{M}$.

As in Section 5.3, we allow ℓ to be a variable. The cost of variable ℓ is the cost of a single computation for the maximum allowed ℓ, plus a minor cost for bit operations to select relevant inputs to the product.

6 Computing the action: basic algorithms

Jao, LeGrow, Leonardi, and Ruiz-Lopez [38] suggested a three-level quantum algorithm to compute $\mathcal{L}_1^{e_1} \cdots \mathcal{L}_n^{e_n}$. This section shows how to make the algorithm an order of magnitude faster for any particular failure probability.

6.1. Baseline: reliably computing each $\mathcal{L}_i$. The lowest level in [38] *reliably* computes $\mathcal{L}_i$ as follows. Generate r uniform random points on the curve or twist, as in Section 4.1. Multiply each point by $(p+1)/\ell_i$, as in Section 5.1, hoping to obtain a point of order ℓ_i on the curve. Use Vélu's formulas to finish the computation, as in Section 5.3.

Each point has success probability $(1/2)(1 - 1/\ell_i)$, where $1/2$ is the probability of obtaining a curve point (rather than a twist point) and $1 - 1/\ell_i$ is the probability of obtaining a point of order ℓ_i (rather than order 1). The chance of all r points failing is thus $(\ell_i + 1)^r/(2\ell_i)^r$, decreasing from $(2/3)^r$ for $\ell_i = 3$ down towards $(1/2)^r$ as ℓ_i grows . One chooses r to obtain a failure probability as small as desired for the isogeny computation, and for the higher levels of the algorithm.

The lowest level optionally computes $\mathcal{L}_i^{-1}$ instead of $\mathcal{L}_i$. The approach in [38], following [15], is to use points on the twist instead of points on the curve; an alternative is to compute $\mathcal{L}_i^{-1}(A)$ as $-\mathcal{L}_i(-A)$.

The middle level of the algorithm computes $\mathcal{L}_i^e$, where e is a variable whose absolute value is bounded by a constant C. This level calls the lowest level exactly C times, performing a series of C steps of $\mathcal{L}_i^{\pm 1}$, using bit operations on e to decide whether to retain the results of each step. The ± 1 is chosen as the sign of e, or as an irrelevant 1 if $e = 0$.

The highest level of the algorithm computes $\mathcal{L}_1^{e_1} \cdots \mathcal{L}_n^{e_n}$, where each e_i is between $-C$ and C, by calling the middle level n times, starting with $\mathcal{L}_1^{e_1}$ and ending with $\mathcal{L}_n^{e_n}$. Our definition of the action applied $\mathcal{L}_n^{e_n}$ first, but the $\mathcal{L}_i$ operators commute with each other, so the order does not matter.

Importance of bounding each exponent. We emphasize that this algorithm requires each exponent e_i to be between $-C$ and C, i.e., requires the vector $(e_1, \ldots, e_n)$ to have ∞-norm at most C.

We use $C = 5$ for CSIDH-512 as an illustrative example, but all known attacks use larger vectors (see Section 2.1). C is chosen in [38] so that every input, every vector in superposition, has ∞-norm at most C; smaller values of C create a failure probability that needs to be analyzed.

We are not saying that the ∞-norm is the only important feature of the input vectors. On the contrary: our constant-time subroutine to handle variable-ℓ isogenies creates opportunities to share work between separate exponents. See Section 5.3 and Section 7.

Concrete example. For concreteness we consider uniform random input vectors $e \in \{-5, \ldots, 5\}^{74}$. The highest level calls the middle level $n = 74$ times, and the middle level calls the lowest level $C = 5$ times. Taking $r = 70$ guarantees failure probability at most $(2/3)^{70}$ at the lowest level, and thus failure probability at most $1 - (1 - (2/3)^{70})^{74 \cdot 5} \approx 0.750 \cdot 2^{-32}$ for the entire algorithm.

This type of analysis is used in [38] to select r. We point out that the failure probability of the algorithm is actually lower, and a more accurate analysis allows a smaller value of r. One can, for example, replace $(1 - (2/3)^r)^{74}$ with $\prod_i (1 - (\ell_i + 1)^r/(2\ell_i)^r)$, showing that $r = 59$ suffices for failure probability below 2^{-32}. With more work one can account for the distribution of input vectors e, rather than taking the worst-case e as in [38]. However, one cannot hope to do better than $r = 55$ here: there is a 10/11 chance that at least one 3-isogeny is required, and taking $r \leq 54$ means that this 3-isogeny fails with probability at least $(2/3)^{54}$, for an overall failure chance at least $(10/11)(2/3)^{54} > 2^{-32}$.

With the choice $r = 70$ as in [38], there are $74 \cdot 5 \cdot 70 = 25900$ iterations, in total using more than 100 million multiplications in $\mathbb{F}_p$. In the rest of this section we will reduce the number of iterations by a factor 30, and in Section 7 we will reduce the number of iterations by another factor 3, with only moderate increases in the cost of each iteration.

6.2. Fewer failures, and sharing failures.

We now introduce Algorithm 6.1, which improves upon the algorithm from [38] in three important ways. First, we use Elligator to target the curve (or the twist if desired); see Section 4.2. This reduces the failure probability of r points from $(2/3)^r$ to, heuristically, $(1/3)^r$ for $\ell_i = 3$; from $(3/5)^r$ to $(1/5)^r$ for $\ell_i = 5$; from $(4/7)^r$ to $(1/7)^r$ for $\ell_i = 7$; etc.

Algorithm 6.1: Basic class-group action evaluation.

```
Parameters: Odd primes $\ell_1 < \cdots < \ell_n$, a prime $p = 4\ell_1 \cdots \ell_n - 1$, and positive
            integers $(r_1, \ldots, r_n)$.
Input: $A \in S_p$, integers $(e_1, \ldots, e_n)$.
Output: $\mathcal{L}_1^{e_1} \cdots \mathcal{L}_n^{e_n}(A)$ or "fail".
for $i \leftarrow 1$ to $n$ do
    for $j \leftarrow 1$ to $r_i$ do
        Let $s = \mathrm{sign}(e_i) \in \{-1, 0, +1\}$.
        Find a random point $P$ on $E_{sA}$ using Elligator.
        Compute $Q \leftarrow ((p+1)/\ell_i)P$.
        Compute $B$ with $E_B \cong E_{sA}/\langle Q \rangle$ if $Q \neq \infty$.
        Set $A \leftarrow sB$ if $Q \neq \infty$ and $s \neq 0$.
        Set $e_i \leftarrow e_i - s$ if $Q \neq \infty$.
Set $A \leftarrow$ "fail" if $(e_1, \ldots, e_n) \neq (0, \ldots, 0)$.
Return $A$.
```

Second, we allow a separate r_i for each ℓ_i. This lets us exploit the differences in failure probabilities as ℓ_i varies.

Third, we handle failures at the middle level instead of the lowest level. The strategy in [38] to compute $\mathcal{L}_i^e$ with $-C \leq e \leq C$ is to perform C iterations, where each iteration builds up many points on one curve and *reliably* moves to the next curve. We instead perform r_i iterations, where each iteration *tries* to move from one curve to the next by generating just one point. For $C = 1$ this is the same, but for larger C we obtain better tradeoffs between the number of points and the failure probability.

As a concrete example, generating 20 points on one curve with Elligator has failure probability $(1/3)^{20}$ for $\ell_i = 3$. A series of 5 such computations, overall generating 100 points, has failure probability $1 - (1 - (1/3)^{20})^5 \approx 2^{-29.37}$. If we instead perform just 50 iterations, where each iteration generates one point to move 1 step with probability 2/3, then the probability that we move fewer than 5 steps is just $3846601/3^{50} \approx 2^{-57.37}$; see Section 6.3. Our iterations are more expensive than in [38]—next to each Elligator computation, we always perform the steps for computing an ℓ_i-isogeny, even if $Q = \infty$—but (for CSIDH-512 etc.) this is not a large effect: the cost of each iteration is dominated by scalar multiplication.

We emphasize that all of our algorithms take constant time. When we write "Compute $X \leftarrow Y$ if c" we mean that we always compute Y and the bit c, and we then replace the jth bit X_j of X with the jth bit Y_j of Y for each j if c is set, by replacing X_j with $X_j \oplus c(X_j \oplus Y_j)$. This is why Algorithm 6.1 always carries out the bit operations for computing an ℓ_i-isogenous curve, as noted above, even if $Q = \infty$.

Table 6.1. Examples of choices of r_i for Algorithm 6.1 for three levels of failure probability for uniform random CSIDH-512 vectors with entries in $\{-5, \ldots, 5\}$. Failure probabilities ϵ are rounded to three digits after the decimal point. The "total" column is $\sum r_i$, the total number of iterations. The "[38]" column is $74 \cdot 5 \cdot r$, the number of iterations in the algorithm of [38], with r chosen as in [38] to have $1 - (1 - (2/3)^r)^{74 \cdot 5}$ at most 2^{-1} or 2^{-32} or 2^{-256}. Compare Table 6.2 for $\{-10, \ldots, 10\}$.

ϵ \ ℓ_i	3	5	7	11	13	17	…	359	367	373	587	total	[38]
$0.499 \cdot 2^{-1}$	11	9	8	7	7	6	…	5	5	5	5	406	5920
$0.178 \cdot 2^{-32}$	36	25	21	18	17	16	…	10	10	10	9	869	25900
$0.249 \cdot 2^{-256}$	183	126	105	85	80	73	…	37	37	37	34	3640	167610

6.3. Analysis. We consider the inner loop body of Algorithm 6.1 for a fixed i, hence write $\ell = \ell_i$, $e = e_i$, and $r = r_i$ for brevity.

Heuristically (see Section 5.2), we model each point Q as independent and uniform random in a cyclic group of order ℓ, so Q has order 1 with probability $1/\ell$ and order ℓ with probability $1 - 1/\ell$. The number of points of order ℓ through r iterations of the inner loop is binomially distributed with parameters r and $1 - 1/\ell$. The probability that this number is $|e|$ or larger is $\mathrm{prob}_{\ell,e,r} = \sum_{t=|e|}^{r} \binom{r}{t} (1 - 1/\ell)^t / \ell^{r-t}$. This is exactly the probability that Algorithm 6.1 successfully performs the $|e|$ desired iterations of $\mathcal{L}^{\mathrm{sign}(e)}$.

Let C be a nonnegative integer. The overall success probability of the algorithm for a particular input vector $(e_1, \ldots, e_n) \in \{-C, \ldots, C\}^n$ is

$$\prod_{i=1}^{n} \mathrm{prob}_{\ell_i, e_i, r_i} \geq \prod_{i=1}^{n} \mathrm{prob}_{\ell_i, C, r_i}.$$

Average over vectors to see that the success probability of the algorithm for a uniform random vector in $\{-C, \ldots, C\}^n$ is $\prod_{i=1}^{n} \bigl(\sum_{-C \leq e \leq C} \mathrm{prob}_{\ell_i, e, r_i} / (2C+1)\bigr)$.

6.4. Examples of target failure probabilities. The acceptable level of failure probability for our algorithm depends on the attack using the algorithm. For concreteness we consider three possibilities for CSIDH-512 failure probabilities, namely having the algorithm fail for a uniform random vector with probabilities at most 2^{-1}, 2^{-32}, and 2^{-256}.

Our rationale for considering these probabilities is as follows. Probabilities around 2^{-1} are easy to test, and may be of interest beyond this paper for constructive scenarios where failing computations can simply be retried. If each computation needs to work correctly, and there are many computations, then failure probabilities need to be much smaller, say 2^{-32}. Asking for every input in superposition to work correctly in one computation (for example, [38] asks for this) requires a much smaller failure probability, say 2^{-256}. Performance results

Table 6.2. Examples of choices of r_i for Algorithm 6.1 for three levels of failure probability for uniform random CSIDH-512 vectors with entries in $\{-10,\dots,10\}$. Failure probabilities ϵ are rounded to three digits after the decimal point. The "total" column is $\sum r_i$, the total number of iterations. The "[38]" column is $74 \cdot 10 \cdot r$, the number of iterations in the algorithm of [38], with r chosen as in [38] to have $1-(1-(2/3)^r)^{74\cdot 10}$ at most 2^{-1} or 2^{-32} or 2^{-256}. Compare Table 6.1 for $\{-5,\dots,5\}$.

ϵ \ ℓ_i	3	5	7	11	13	17	…	359	367	373	587	total	[38]
$0.521 \cdot 2^{-1}$	20	15	14	13	12	12	…	10	10	10	10	786	13320
$0.257 \cdot 2^{-32}$	48	34	30	25	24	22	…	15	15	15	14	1296	52540
$0.215 \cdot 2^{-256}$	201	139	116	96	90	82	…	43	43	43	41	4185	335960

for these three cases also provide an adequate basis for estimating performance in other cases.

Table 6.1 presents three reasonable choices of $(r_1,\dots,r_n)$, one for each of the failure probabilities listed above, for the case of CSIDH-512 with uniform random vectors with entries in $\{-5,\dots,5\}$. For each target failure probability δ and each i, the table chooses the minimum r_i such that $\sum_{-C\le e\le C} \operatorname{prob}_{\ell_i,e,r_i}/(2C+1)$ is at least $(1-\delta)^{1/n}$. The overall success probability is then at least $1-\delta$ as desired. The discontinuity of choices of $(r_1,\dots,r_n)$ means that the actual failure probability ϵ is actually somewhat below δ, as shown by the coefficients 0.499, 0.178, 0.249 in Table 6.1. We could move closer to the target failure probability by choosing successively $r_n, r_{n-1}, \dots$, adjusting the probability $(1-\delta)^{1/n}$ at each step in light of the overshoot from previous steps. The values r_i for $\epsilon \approx 0.499 \cdot 2^{-1}$ have been experimentally verified using a modified version of the CSIDH software. To illustrate the impact of larger vector entries, we also present similar data in Table 6.2 for uniform random vectors with entries in $\{-10,\dots,10\}$.

The "total" column in Table 6.1 shows that this algorithm uses, e.g., 869 iterations for failure probability $0.178 \cdot 2^{-32}$ with vector entries in $\{-5,\dots,5\}$. Each iteration consists mostly of a scalar multiplication, plus some extra cost for Elligator, Vélu's formulas, etc. Overall there are roughly 5 million field multiplications, accounting for roughly 2^{41} nonlinear bit operations, implying a quantum computation using roughly 2^{45} T-gates.

As noted in Section 1, using the algorithm of [38] on top of the modular-multiplication algorithm from [63] would use approximately 2^{51} nonlinear bit operations for the same distribution of input vectors. We save a factor 30 in the number of iterations compared to [38], and we save a similar factor in the number of bit operations for each modular multiplication compared to [63].

We do not analyze this algorithm in more detail: the algorithms we present below are faster.

7 Reducing the top nonzero exponent

Most of the iterations in Algorithm 6.1 are spent on exponents that are already 0. For example, consider the 869 iterations mentioned above for failure probability $0.178 \cdot 2^{-32}$ for uniform random CSIDH-512 vectors with entries in $\{-5, \ldots, 5\}$. Entry e_i has absolute value 30/11 on average, and needs $(30/11)\ell_i/(\ell_i - 1)$ iterations on average, for a total of $\sum_i (30/11)\ell_i/(\ell_i - 1) \approx 206.79$ useful iterations on average. This means that there are 662.21 useless iterations on average, many more than one would expect to be needed to guarantee this failure probability.

This section introduces a constant-time algorithm that achieves the same failure probability with far fewer iterations. For example, in the above scenario, just 294 iterations suffice to reduce the failure probability below 2^{-32}. Each iteration becomes (for CSIDH-512) about 25% more expensive, but overall the algorithm uses far fewer bit operations.

7.1. Iterations targeting variable ℓ. It is obvious how to avoid useless iterations for variable-time algorithms: when an exponent reaches 0, move on to the next exponent. In other words, always focus on reducing a nonzero exponent, if one exists.

What is new is doing this in constant time. This is where we exploit the Matryoshka-doll structure from Section 5.3, computing an isogeny for variable ℓ in constant time. We now pay for an ℓ_n-isogeny in each iteration rather than an ℓ-isogeny, but the iteration cost is still dominated by scalar multiplication. Concretely, for CSIDH-512, an average ℓ-isogeny costs about 600 multiplications, and an ℓ_n-isogeny costs about 2000 multiplications, but a scalar multiplication costs about 5000 multiplications.

We choose to reduce the top exponent that is not 0. "Top" here refers to position, not value: we reduce the nonzero e_i where i is maximized. See Algorithm 7.1.

7.2. Upper bounds on the failure probability. One can crudely estimate the failure probability of Algorithm 7.1 in terms of the 1-norm $E = |e_1| + \cdots + |e_n|$ as follows. Model each iteration as having failure probability 1/3 instead of $1/\ell_i$; this produces a loose upper bound for the overall failure probability of the algorithm.

In this model, the chance of needing exactly r iterations to find a point of order ℓ_i is the coefficient of x^r in the power series

$$(2/3)x + (2/9)x^2 + (2/27)x^3 + \cdots = 2x/(3 - x).$$

The chance of needing exactly r iterations to find all E points is the coefficient of x^r in the Eth power of that power series, namely $c_r = \binom{r-1}{E-1} 2^E/3^r$ for $r \geq E$. See generally [74] for an introduction to the power-series view of combinatorics; there are many other ways to derive the formula $\binom{r-1}{E-1} 2^E/3^r$, but we make critical use of power series for fast computations in Sections 7.3 and 8.3.

The failure probability of r iterations of Algorithm 7.1 is at most the failure probability of r iterations in this model, namely $f(r, E) = 1 - c_E - c_{E+1} - \cdots - c_r$.

Algorithm 7.1: Evaluating the class-group action by reducing the top nonzero exponent.

```
Parameters: Odd primes $\ell_1 < \cdots < \ell_n$ with $n \geq 1$, a prime $p = 4\ell_1 \cdots \ell_n - 1$,
            and a positive integer $r$.
Input: $A \in S_p$, integers $(e_1, \ldots, e_n)$.
Output: $\mathcal{L}_1^{e_1} \cdots \mathcal{L}_n^{e_n}(A)$ or "fail".
for $j \leftarrow 1$ to $r$ do
    Let $i = \max\{k : e_k \neq 0\}$, or $i = 1$ if each $e_k = 0$.
    Let $s = \operatorname{sign}(e_i) \in \{-1, 0, +1\}$.
    Find a random point $P$ on $E_{sA}$ using Elligator.
    Compute $Q \leftarrow ((p+1)/\ell_i)P$.
    Compute $B$ with $E_B \cong E_{sA}/\langle Q \rangle$ if $Q \neq \infty$, using the $\ell_i$-isogeny formulas from
      Section 5.3 with maximum degree $\ell_n$.
    Set $A \leftarrow sB$ if $Q \neq \infty$ and $s \neq 0$.
    Set $e_i \leftarrow e_i - s$ if $Q \neq \infty$.
Set $A \leftarrow$ "fail" if $(e_1, \ldots, e_n) \neq (0, \ldots, 0)$.
Return $A$.
```

The failure probability of r iterations for a uniform random vector with entries in $\{-C, \ldots, C\}$ is at most $\sum_{0 \leq E \leq nC} f(r, E) g[E]$. Here $g[E]$ is the probability that a vector has 1-norm E, which we compute as the coefficient of x^E in the nth power of the polynomial $(1 + 2x + 2x^2 + \cdots + 2x^C)/(2C+1)$. For example, with $n = 74$ and $C = 5$, the failure probability in this model (rounded to 3 digits after the decimal point) is $0.999 \cdot 2^{-1}$ for $r = 302$; $0.965 \cdot 2^{-2}$ for $r = 319$; $0.844 \cdot 2^{-32}$ for $r = 461$; and $0.570 \cdot 2^{-256}$ for $r = 823$. As a double-check, we observe that a simple simulation of the model for $r = 319$ produces 241071 failures in 1000000 experiments, close to the predicted $0.965 \cdot 2^{-2} \cdot 1000000 \approx 241250$.

7.3. Exact values of the failure probability. The upper bounds from the model above are too pessimistic, except for $\ell_i = 3$. We instead compute the exact failure probabilities as follows.

The chance that $\mathcal{L}_1^{e_1} \cdots \mathcal{L}_n^{e_n}$ requires exactly r iterations is the coefficient of x^r in the power series

$$\left(\frac{(\ell_1 - 1)x}{\ell_1 - x}\right)^{|e_1|} \cdots \left(\frac{(\ell_n - 1)x}{\ell_n - x}\right)^{|e_n|}.$$

What we want is the average of this coefficient over all vectors $(e_1, \ldots, e_n) \in \{-C, \ldots, C\}^n$. This is the same as the coefficient of the average, and the average factors nicely as

$$\left(\sum_{-C \leq e_1 \leq C} \frac{1}{2C+1} \left(\frac{(\ell_1 - 1)x}{\ell_1 - x}\right)^{|e_1|}\right) \cdots \left(\sum_{-C \leq e_n \leq C} \frac{1}{2C+1} \left(\frac{(\ell_n - 1)x}{\ell_n - x}\right)^{|e_n|}\right).$$

We compute this product as a power series with rational coefficients: for example, we compute the coefficients of $x^0, \ldots, x^{499}$ if we are not interested in 500 or more iterations. We then add together the coefficients of $x^0, \ldots, x^r$ to find the exact success probability of r iterations of Algorithm 7.1.

As an example we again take CSIDH-512 with $C = 5$. The failure probability (again rounded to 3 digits after the decimal point) is $0.960 \cdot 2^{-1}$ for $r = 207$; $0.998 \cdot 2^{-2}$ for $r = 216$; $0.984 \cdot 2^{-32}$ for $r = 294$; $0.521 \cdot 2^{-51}$ for $r = 319$; and $0.773 \cdot 2^{-256}$ for $r = 468$. We double-checked these averages against the results of Monte Carlo calculations for these values of r. Each Monte Carlo iteration sampled a uniform random 1-norm (weighted appropriately for the initial probability of each 1-norm), sampled a uniform random vector within that 1-norm, and computed the failure probability for that vector using the single-vector generating function.

7.4. Analysis of the cost. We have fully implemented Algorithm 7.1 in our bit-operation simulator. One iteration for CSIDH-512 uses 9208697761 bit operations, including 3805535430 nonlinear bit operations. More than 95% of the cost is explained as follows:

- Each iteration uses a Montgomery ladder with a 511-bit scalar. (We could save a bit here: the largest useful scalar is $(p+1)/3$, which is below 2^{510}.) We use an affine input point and an affine A, so this costs $2044\mathbf{S} + 3066\mathbf{M}$.
- Each iteration uses the formulas from Section 5.3 with $\ell = 587$. This takes $602\mathbf{S} + 1472\mathbf{M}$: specifically, $584\mathbf{S} + 876\mathbf{M}$ for multiples of the point of order ℓ (again affine); $584\mathbf{M}$ for the product of Edwards y-coordinates; $18\mathbf{S} + 10\mathbf{M}$ for two ℓth powers; and $2\mathbf{M}$ to multiply by two 8th powers. (We merge the $6\mathbf{S}$ for the 8th powers into the squarings used for the ℓth powers.)
- Each iteration uses two inversions to obtain affine Q and A, each $507\mathbf{S}+97\mathbf{M}$, and one Legendre-symbol computation, $506\mathbf{S} + 96\mathbf{M}$.

This accounts for $4166\mathbf{S}+4828\mathbf{M}$ per iteration, i.e., $4166 \cdot 349596 + 4828 \cdot 447902 = 3618887792$ nonlinear bit operations.

The cost of 294 iterations is simply $294 \cdot 3805535430 = 1118827416420$ nonlinear bit operations. This justifies the first (B, ϵ) claim in Section 1.

7.5. Decreasing the maximum degrees. Always performing isogeny computations capable of handling degrees up to ℓ_n is wasteful: With overwhelming probability, almost all of the 294 iterations required for a failure probability of less than 2^{-32} with the approach discussed so far actually compute isogenies of degree (much) less than ℓ_n. For example, with e uniformly random in $\{-5, \ldots, 5\}$, the probability that 10 iterations are not sufficient to eliminate all 587-isogenies is approximately 2^{-50}. Therefore, using smaller upper bounds on the isogeny degrees for later iterations of the algorithm will not do much harm to the success probability while significantly improving the performance. We modify Algorithm 7.1 as follows:

- Instead of a single parameter r, we use a list $(r_1, \ldots, r_n)$ of non-negative integers, each r_i denoting the number of times an isogeny computation capable of handling degrees up to ℓ_i is performed.

- The loop iterating from 1 through r is replaced by an outer loop on u from n down to 1, and inside that an inner loop on j from 1 up to r_u. The loop body is unchanged, except that the maximum degree for the isogeny formulas is now ℓ_u instead of ℓ_n.

For a given sequence $(r_1, \ldots, r_n)$, the probability of success can be computed as follows:

- For each $i \in \{1, \ldots, n\}$, compute the generating function

$$\phi_i(x) = \sum_{-C \le e_i \le C} \frac{1}{2C+1} \left(\frac{(\ell_i - 1)x}{\ell_i - x} \right)^{|e_i|}$$

  of the number of ℓ_i-isogeny steps that have to be performed.
- Since we are no longer only interested in the total number of isogeny steps to be computed, but also in their degrees, we cannot simply take the product of all ϕ_i as before. Instead, to account for the fact that failing to compute a ℓ_i-isogeny before the maximal degree drops below ℓ_i implies a total failure, we iteratively compute the product of the ϕ_i from $k = n$ down to 1, but truncate the product after each step. Truncation after some power x^t means eliminating all branches of the probabilistic process in which more than t isogeny steps are needed for the computations so far. In our case we use $t = \sum_{j=i}^{n} r_j$ after multiplying by ϕ_i, which removes all outcomes in which more isogeny steps of degree $\ge \ell_i$ would have needed to be computed.
- After all ϕ_i have been processed (including the final truncation), the probability of success is the sum of all coefficients of the remaining power series.

Note that we have only described a procedure to compute the success probabilty once $r_1, \ldots, r_n$ are known. It is unclear how to find the optimal values r_i which minimize the cost of the resulting algorithm, while at the same time respecting a certain failure probability. We tried various reasonable-looking choices of strategies to choose the r_i according to certain prescribed failure probabilities after each individual step. Experimentally, a good rule seems to be that the failure probability after processing ϕ_i should be bounded by $\epsilon \cdot 2^{2/i-2}$, where ϵ is the overall target failure probability. The results are shown in Table 7.1.

The average degree of the isogenies used constructively in CSIDH-512 is about 174.6, which is not much smaller than the average degree we achieve. Since we still need to control the error probability, it does not appear that one can expect to get much closer to the constructive case.

Also note that the total number of isogeny steps for $\epsilon \approx 2^{-32}$ and $\epsilon \approx 2^{-256}$ is each only one more than the previous number r of isogeny computations, hence one can expect significant savings using this strategy. Assuming that about 1/4 of the total time is spent on Vélu's formulas (which is close to the real proportion), we get a speedup of about 16% for $\epsilon \approx 2^{-32}$ and about 17% for $\epsilon \approx 2^{-256}$.

Table 7.1. Examples of choices of $r_i, \dots, r_i$ for Algorithm 7.1 with reducing the maximal degree in Vélu's formulas for uniform random CSIDH-512 vectors with entries in $\{-5, \dots, 5\}$. Failure probabilities ϵ are rounded to three digits after the decimal point.

ϵ	$r_n \dots r_1$	$\sum r_i$	avg. ℓ
$0.594 \cdot 2^{-1}$	5 3 4 5 3 5 5 4 3 5 4 3 4 4 3 4 3 4 3 3 3 4 3 3 3 4 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 2 4 2 3 3 3 3 2 3 3 3 2 3 3 2 3 2 3 2 3 2 2 3 2 2 2 1 1 1 0 0	218	205.0
$0.970 \cdot 2^{-32}$	9 5 5 5 5 5 4 5 5 5 4 5 4 5 5 4 5 4 4 5 5 4 4 4 5 4 4 4 4 4 3 5 3 4 4 4 3 4 4 4 3 4 4 3 4 3 4 3 4 4 3 4 3 3 4 4 3 3 4 3 3 4 3 4 3 3 4 3 3 3 4 3 3 4	295	196.0
$0.705 \cdot 2^{-256}$	34 8 6 6 5 6 6 5 5 6 5 6 5 5 5 5 6 5 5 5 5 6 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 4 6 5 5 5 5 5 4 6 5 5 5 5 5 5 5 6 5 5 6 6 6 6 7 7 11 16 38	469	182.7

8 Pushing points through isogenies

Algorithms 6.1 and 7.1 spend most of their time on scalar multiplication. This section pushes points through isogenies to reduce the time spent on scalar multiplication, saving time overall.

The general idea of balancing isogeny computation with scalar multiplication was introduced in [22] in the SIDH context, and was reused in the variable-time CSIDH algorithms in [15]. This section adapts the idea to the context of constant-time CSIDH computation.

8.1. Why pushing points through isogenies saves time. To illustrate the main idea, we begin by considering a sequence of just two isogenies with the same sign. Specifically, assume that, given distinct ℓ_1 and ℓ_2 dividing $p+1$, we want to compute $\mathcal{L}_1\mathcal{L}_2(A) = B$. Here are two different methods:

- **Method 1.** The method of Algorithm 6.1 uses Elligator to find $P_1 \in E_A(\mathbb{F}_p)$, computes $Q_1 \leftarrow [(p+1)/\ell_1]P_1$, computes $E_{A'} = E_A/\langle Q_1\rangle$, uses Elligator to find $P_2 \in E_{A'}(\mathbb{F}_p)$, computes $Q_2 \leftarrow [(p+1)/\ell_2]P_2$, and computes $E_B = E_{A'}/\langle Q_2\rangle$. Failure cases: if $Q_1 = \infty$ then this method computes $A' = A$, failing to compute $\mathcal{L}_1$; similarly, if $Q_2 = \infty$ then this method computes $B = A'$, failing to compute $\mathcal{L}_2$.
- **Method 2.** The method described in this section instead uses Elligator to find $P \in E_A(\mathbb{F}_p)$, computes $R \leftarrow [(p+1)/\ell_1\ell_2]P$, computes $Q \leftarrow [\ell_2]R$, computes $\varphi : E_A \to E_{A'} = E_A/\langle Q\rangle$ and $Q' = \varphi(R)$, and computes $E_B = E_{A'}/\langle Q'\rangle$. Failure cases: if $Q = \infty$ then this method computes $Q' = R$ (which has order dividing ℓ_2) and $A' = A$, failing to compute $\mathcal{L}_1$; if $Q' = \infty$ then this method computes $B = A'$, failing to compute $\mathcal{L}_2$.

For concreteness, we compare the costs of these methods for CSIDH-512. The rest of this subsection uses approximations to the costs of lower-level operations to simplify the analysis. The main costs are as follows:

- For p a 512-bit prime, Elligator costs approximately $600\mathbf{M}$.
- Given $P \in E(\mathbb{F}_p)$ and a positive integer k, the computation of $[k]P$ via the Montgomery ladder, as described in Section 3.3, costs approximately $10(\log_2 k)\mathbf{M}$, i.e., approximately $(5120 - 10\log_2 \ell)\mathbf{M}$ if $k = (p+1)/\ell$.
- The computation of a degree-ℓ isogeny via the method described in Section 5.3 costs approximately $(3.5\ell + 2\log_2 \ell)\mathbf{M}$.
- Given an ℓ-isogeny $\varphi_\ell : E \to E'$ and $P \in E(\mathbb{F}_p)$, the computation of $\varphi_\ell(P)$ via the method described in Section 5.4 costs approximately $2\ell\mathbf{M}$.

Method 1 costs approximately

$$(1200 + 10240 + 3.5\ell_1 + 3.5\ell_2 - 8\log_2 \ell_1 - 8\log_2 \ell_2)\mathbf{M},$$

while Method 2 costs approximately

$$(600 + 5120 + 5.5\ell_1 + 3.5\ell_2 - 8\log_2 \ell_1 + 2\log_2 \ell_2)\mathbf{M}.$$

The savings of $(600 + 5120)\mathbf{M}$ clearly outweighs the loss of $(2\ell_1 + 10\log_2 \ell_2)\mathbf{M}$, since the largest value of ℓ_i is 587.

There are limits to the applicability of Method 2: it cannot combine two isogenies of opposite signs, it cannot combine two isogenies using the same prime, and it cannot save time in applying just one isogeny. We will analyze the overall magnitude of these effects in Section 8.3.

8.2. Handling the general case, two isogenies at a time. Algorithm 8.1 computes $\mathcal{L}_1^{e_1} \cdots \mathcal{L}_n^{e_n}(A)$ for any exponent vector $(e_1, \ldots, e_n)$. Each iteration of the algorithm tries to perform two isogenies: one for the top nonzero exponent (if the vector is nonzero), and one for the next exponent having the same sign (if the vector has another exponent of this sign). As in Section 7, “top” refers to position, not value.

The algorithm pushes the first point through the first isogeny, as in Section 8.1, to save the cost of generating a second point. Scalar multiplication, isogeny computation, and isogeny application use the constant-time subroutines described in Sections 3.3, 5.3, and 5.4 respectively. The cost of these algorithms depends on the bound ℓ_n for the prime for the top nonzero exponent and the bound ℓ_{n-1} for the prime for the next exponent. The two prime bounds have asymmetric effects upon costs; we exploit this by applying the isogeny for the top nonzero exponent *after* the isogeny for the next exponent.

Analyzing the correctness of Algorithm 8.1—assuming that there are enough iterations; see Section 8.3—requires considering three cases. The first case is that the exponent vector is 0. Then i, i', s are initialized to $0, 0, 1$ respectively, and i, i' stay 0 throughout the iteration, so A does not change and the exponent vector does not change.

The second case is that the exponent vector is nonzero and the top nonzero exponent e_i is the only exponent having sign s. Then i' is 0 throughout the iteration, so the “first isogeny” portion of Algorithm 8.1 has no effect. The point $Q = R$ in the “second isogeny” portion is cP where $c = (p+1)/\ell_i$, so $\ell_i Q = \infty$. If $Q = \infty$ then i is set to 0 and the entire iteration has no effect, except for setting

Algorithm 8.1: Evaluating the class-group action by reducing the top nonzero exponent and the next exponent with the same sign.

```
Parameters: Odd primes $\ell_1 < \cdots < \ell_n$ with $n \geq 1$, a prime $p = 4\ell_1 \cdots \ell_n - 1$,
            and a positive integer $r$.
Input: $A \in S_p$, integers $(e_1, \ldots, e_n)$.
Output: $\mathcal{L}_1^{e_1} \cdots \mathcal{L}_n^{e_n}(A)$ or "fail".
for $j \leftarrow 1$ to $r$ do
    Set $I \leftarrow \{k : 1 \leq k \leq n$ and $e_k \neq 0\}$.
    Set $i \leftarrow \max I$ and $s \leftarrow \mathrm{sign}(e_i) \in \{-1, 1\}$, or $i \leftarrow 0$ and $s \leftarrow 1$ if $I = \{\}$.
    Set $I' \leftarrow \{k : 1 \leq k < i$ and $\mathrm{sign}(e_k) = s\}$.
    Set $i' \leftarrow \max I'$, or $i' \leftarrow 0$ if $I' = \{\}$.
    Twist. Set $A \leftarrow sA$.
    Isogeny preparation. Find a random point $P$ on $E_A$ using Elligator.
    Compute $R \leftarrow cP$ where $c = 4\prod_{1 \leq j \leq n, j \neq i, j \neq i'} \ell_j$.
    First isogeny. Compute $Q \leftarrow \ell_i R$, where $\ell_0$ means 1.
    [Now $\ell_{i'} Q = \infty$ if $i' \neq 0$.] Set $i' \leftarrow 0$ if $Q = \infty$.
    Compute $B$ with $E_B \cong E_A/\langle Q \rangle$ if $i' \neq 0$, using the $\ell_{i'}$-isogeny formulas from
      Section 5.3 with maximum degree $\ell_{n-1}$.
    Set $R$ to the image of $R$ in $E_B$ if $i' \neq 0$, using the $\ell_{i'}$-isogeny formulas from
      Section 5.4 with maximum degree $\ell_{n-1}$.
    Set $A \leftarrow B$ and $e_{i'} \leftarrow e_{i'} - s$ if $i' \neq 0$.
    Second isogeny. Set $Q \leftarrow R$.
    [Now $\ell_i Q = \infty$ if $i \neq 0$.] Set $i \leftarrow 0$ if $Q = \infty$.
    Compute $B$ with $E_B \cong E_A/\langle Q \rangle$ if $i \neq 0$, using the $\ell_i$-isogeny formulas from
      Section 5.3 with maximum degree $\ell_n$.
    Set $A \leftarrow B$ and $e_i \leftarrow e_i - s$ if $i \neq 0$.
    Untwist. Set $A \leftarrow sA$.
Set $A \leftarrow$ "fail" if $(e_1, \ldots, e_n) \neq (0, \ldots, 0)$.
Return $A$.
```

A to sA and then back to $s(sA) = A$. If $Q \neq \infty$ then i stays nonzero and A is replaced by $\mathcal{L}_i(A)$, so A at the end of the iteration is $\mathcal{L}_i^s$ applied to A at the beginning of the iteration, while s is subtracted from e_i.

The third case is that the exponent vector is nonzero and that $e_{i'}$ is the next exponent having the same sign s as the top nonzero exponent e_i. By construction $i' < i \leq n$ so $\ell_{i'} \leq \ell_{n-1}$. Now $R = cP$ where $c = (p+1)/(\ell_i \ell_{i'})$. The first isogeny uses the point $Q = \ell_i R$, which is either ∞ or a point of order $\ell_{i'}$. If Q is ∞ then i' is set to 0; both A and the vector are unchanged; the point R must have order dividing ℓ_i; and the second isogeny proceeds as above using this point. If Q has order $\ell_{i'}$ then the first isogeny replaces A with $\mathcal{L}_{i'}(A)$, while subtracting s from $e_{i'}$ and replacing R with a point of order dividing ℓ_i on the new curve (note that the $\ell_{i'}$-isogeny removes any $\ell_{i'}$ from orders of points); again the second isogeny proceeds as above.

8.3. Analysis of the failure probability. Consider a modified dual-isogeny algorithm in which the isogeny with a smaller prime is saved to handle later:

- Initialize an iteration counter to 0.
- Initialize an empty bank of positive isogenies.
- Initialize an empty bank of negative isogenies.
- For each ℓ in decreasing order:
  - While an ℓ-isogeny needs to be done and the bank has an isogeny of the correct sign: Withdraw an isogeny from the bank, apply the isogeny, and adjust the exponent.
  - While an ℓ-isogeny still needs to be done: Apply an isogeny, adjust the exponent, deposit an isogeny with the bank, and increase the iteration counter.

This uses more bit operations than Algorithm 8.1 (since the work here is not shared across two isogenies), but it has the same failure probability for the same number of iterations. We now focus on analyzing the distribution of the number of iterations used by this modified algorithm.

We use three variables to characterize the state of the modified algorithm before each ℓ:

- $i \geq 0$ is the iteration counter;
- $j \geq 0$ is the number of positive isogenies in the bank;
- $k \geq 0$ is the number of negative isogenies in the bank.

The number of isogenies actually applied so far is $2i-(j+k) \geq i$. The distribution of states is captured by the three-variable formal power series $\sum_{i,j,k} s_{i,j,k}x^i y^j z^k$ where $s_{i,j,k}$ is the probability of state (i,j,k). Note that there is no need to track which primes are paired with which; this is what makes the modified algorithm relatively easy to analyze.

If there are exactly h positive ℓ-isogenies to perform then the new state after those isogenies is $(i, j-h, k)$ if $h \leq j$, or $(i+h-j, h-j, k)$ if $h > j$. This can be viewed as a composition of two operations on the power series. First, multiply by y^{-h}. Second, replace any positive power of y^{-1} with the same power of xy; i.e., replace $x^i y^j z^k$ for each $j < 0$ with $x^{i-j} y^{-j} z^k$.

We actually have a distribution of the number of ℓ-isogenies to perform. Say there are h isogenies with probability q_h. We multiply the original series by $\sum_{h\geq 0} q_h y^{-h}$, and then eliminate negative powers of y as above. We similarly handle $h < 0$, exchanging the role of (j, y) with the role of (k, z).

As in the analyses earlier in the paper, we model each point Q for an ℓ-isogeny as having order 1 with probability $1/\ell$ and order ℓ with probability $1 - 1/\ell$, and we assume that the number of ℓ-isogenies to perform is a uniform random integer $e \in \{-C, \ldots, C\}$. Then q_h for $h \geq 0$ is the coefficient of x^h in $\sum_{0\leq e\leq C}(((\ell-1)x)/(\ell-x))^e/(2C+1)$; also, $q_{-h} = q_h$.

We reduce the time spent on these computations in three ways. First, we discard all states with $i > r$ if we are not interested in more than r iterations. This leaves a cubic number of states for each ℓ: every i between 0 and r inclusive, every j between 0 and i inclusive, and every k between 0 and $i - j$ inclusive.

Second, we use fixed-precision arithmetic, rounding each probability to an integer multiple of (e.g.) 2^{-512}. We round down to obtain lower bounds on success probabilities; we round up to obtain upper bounds on success probabilities; we choose the scale 2^{-512} so that these bounds are as tight as desired. We could save more time by reducing the precision slightly at each step of the computation, and by using standard interval-arithmetic techniques to merge computations of lower and upper bounds.

Third, to multiply the series $\sum_{i,j,k} s_{i,j,k} x^i y^j z^k$ by $\sum_{h \ge 0} q_h y^{-h}$, we actually multiply $\sum_j s_{i,j,k} y^j$ by $\sum_{h \ge 0} q_h y^{-h}$ for each (i, k) separately. We use Sage for these multiplications of univariate polynomials with integer coefficients. Sage, in turn, uses fast multiplication algorithms whose cost is essentially bd for d b-bit coefficients, so our total cost for n primes is essentially bnr^3.

Concretely, we use under two hours on one core of a 3.5GHz Intel Xeon E3-1275 v3 to compute lower bounds on all the success probabilities for CSIDH-512 with $b = 512$ and $r = 349$, and under three hours[4] to compute upper bounds. Our convention of rounding failure probabilities to 3 digits makes the lower bounds and upper bounds identical, so presumably we could have used less precision.

We find, e.g., failure probability $0.943 \cdot 2^{-1}$ after 106 iterations, failure probability $0.855 \cdot 2^{-32}$ after 154 iterations, and failure probability $0.975 \cdot 2^{-257}$ after 307 iterations. Compared to the $207, 294, 468$ single-isogeny iterations required in Section 7.3, the number of iterations has decreased to 51.2%, 52.3%, 65.6% respectively.

8.4. Analysis of the cost. We have fully implemented Algorithm 8.1 in our bit-operation simulator. An iteration of Algorithm 8.1 uses 4969644344 nonlinear bit operations, about 1.306 times more expensive than an iteration of Algorithm 7.1.

If the number of iterations were multiplied by exactly 0.5 then the total cost would be multiplied by 0.653. Given the actual number of iterations (see Section 8.3), the cost is actually multiplied by 0.669, 0.684, 0.857 respectively. In particular, we reach failure probability $0.855 \cdot 2^{-32}$ with $154 \cdot 4969644344 = 765325228976$ nonlinear bit operations. This justifies the second (B, ϵ) claim in Section 1.

8.5. Variants. The idea of pushing points through isogenies can be combined with the idea of gradually reducing the maximum prime allowed in the Matryoshka-doll isogeny formulas. This is compatible with our techniques for analyzing failure probabilities.

[4] It is unsurprising that lower bounds are faster: many coefficients q_h round down to 0. We could save time in the upper bounds by checking for stretches of coefficients that round up to, e.g., $1/2^{512}$, and using additions to multiply by those stretches.

A dual-isogeny iteration very late in the computation is likely to have a useless second isogeny. It should be slightly better to replace some of the last dual-isogeny iterations with single-isogeny iterations. This is also compatible with our techniques for analyzing failure probabilities.

There are many different possible pairings of primes: one can take any two distinct positions where the exponents have the same sign. Possibilities include reducing exponents from the bottom rather than the top; reducing the top nonzero exponent and the bottom exponent with the same sign; always pairing "high" positions with "low" positions; always reducing the largest exponents in absolute value; always reducing e_i where $|e_i|\ell_i/(\ell_i-1)$ is largest. For some of these ideas it is not clear how to efficiently analyze failure probabilities.

This section has focused on reusing an Elligator computation and large scalar multiplication for (in most cases) two isogeny computations, dividing the scalar-multiplication cost by (nearly) 2, in exchange for some overhead. We could push a point through more isogenies, although each extra isogeny has further overhead with less and less benefit, and computing the failure probability becomes more expensive. For comparison, [15] reuses one point for every ℓ where e_ℓ has the same sign; the number of such ℓ is variable, and decreases as the computation continues. For small primes it might also save time to push multiple points through one isogeny, as in [22].

9 Computing ℓ-isogenies using division polynomials

As the target failure probability decreases, the algorithms earlier in this paper spend more and more iterations handling the possibility of repeated failures for small primes ℓ—especially $\ell = 3$, where each generated point fails with probability 1/3.

This section presents and analyzes an alternative: a deterministic constant-time subroutine that uses division polynomials to *always* compute ℓ-isogenies. Using division polynomials is more expensive than generating random points, and the cost gap grows rapidly as ℓ increases, but division polynomials have the advantage that each iteration is guaranteed to compute an ℓ-isogeny. See also Section 10 for an alternative that uses modular polynomials rather than division polynomials.

Division polynomials can be applied as a first step to any of our class-group evaluation algorithms: compute the group action for some number of powers of $\mathcal{L}_1^{\pm1},\ldots,\mathcal{L}_s^{\pm1}$ (not necessarily C powers of each), and then handle the remaining isogenies as before. Our rough estimates in this section suggest that the optimal choice of s is small: division polynomials are not of interest for large primes ℓ.

9.1. Algorithm. The idea behind the following algorithm is to take the ℓ-division polynomial ψ_ℓ of E_A, whose roots are the x-coordinates of nonzero ℓ-torsion points; identify a divisor χ_ℓ of ψ_ℓ that defines the $\mathbb{F}_p$-rational subgroup of $E_A[\ell]$; and finally use a variant of Kohel's formulas [45, Section 2.4] to compute the codomain of the isogeny defined by χ_ℓ and thus $B = \mathcal{L}(A)$.

Algorithm 9.1: ℓ-isogeny using division polynomials.

```
Parameters: Odd primes $\ell_1 < \cdots < \ell_n$, a prime $p = 4\ell_1 \cdots \ell_n - 1$, and
            $\ell \in \{\ell_1, \dots, \ell_n\}$.
Input: $A \in S_p$.
Output: $\mathcal{L}(A) \in S_p$.
Compute the $\ell$-division polynomial $\psi_\ell \in \mathbb{F}_p[X]$ of $E_A$.
Compute $\psi'_\ell = \gcd(X^p - X, \psi_\ell)$.
Let $\rho = X^3 + AX^2 + X$ and compute $\chi_\ell = \gcd(\rho^{(p+1)/2} - \rho, \psi'_\ell)$.
Use Lemma 9.1 on $\chi_\ell$ to compute $B$ such that $E_B \cong E_A/E_A(\mathbb{F}_p)[\ell]$.
Return $B$.
```

Lemma 9.1. *Let $E_A \colon y^2 = x^3 + Ax^2 + x$ be a Montgomery curve defined over a field k with $\operatorname{char}(k) \neq 2$. Consider a finite subgroup $G \leq E$ of odd size $n \geq 3$ and let $\chi \in k[x]$ be a monic squarefree polynomial of degree $d = (n-1)/2$ whose roots are exactly the x-coordinates of all nonzero points in G. Write*

$$\sigma = -\chi[d-1]\,; \quad \tau = (-1)^{d+1} \cdot \chi[1]\,; \quad \pi = (-1)^d \cdot \chi[0]\,,$$

where $\chi[i] \in k$ is the coefficient of x^i in χ. Then there exists an isogeny $E_A \to E_B$ with kernel G, where

$$B = \pi(\pi(A - 6\sigma) + 6\tau)\,.$$

Proof. This is obtained by decomposing the formulas from [62] into elementary symmetric polynomials, which happen to occur as the given coefficients of χ. □

Lemma 9.2. *Algorithm 9.1 is correct.*

Proof. First, $X^p - X = \prod_{a \in \mathbb{F}_p}(X - a)$ implies that ψ'_ℓ is the part of ψ_ℓ that splits into linear factors over $\mathbb{F}_p$. Second, for any $\rho \in \mathbb{F}_p$, choosing $y \in \overline{\mathbb{F}_p}$ such that $y^2 = \rho$ gives

$$\rho^{(p+1)/2} - \rho = y(y^p - y) = y \prod_{\alpha \in \mathbb{F}_p} (y - \alpha).$$

Therefore the roots of χ_ℓ are exactly the x-coordinates of the nonzero $\mathbb{F}_p$-rational ℓ-torsion points on E. Finally, the correctness of the output follows from Lemma 9.1. □

9.2. Cost. To analyze how this approach compares to Vélu's formulas, we focus on rough estimates of how cost scales with ℓ, rather than an exact cost analysis. Finite field squarings $\mathbf{S}$ are counted as $\mathbf{M}$ for simplicity. Let $\mu(d)$ denote the cost of multiplying two d-coefficient polynomials. To establish a rough lower bound, we assume $\mu(d) = (d \log_2 d)\mathbf{M}$, which is a model of the complexity of FFT-based fast multiplication techniques. For a rough upper bound, we use $d^2\mathbf{M}$, which is a model of the cost of schoolbook multiplication.

Computing division polynomials. There are two obvious approaches for obtaining division polynomials: Either evaluate the recursive definition directly on a given $A \in \mathbb{F}_p$, or precompute the division polynomials as elements of $\mathbb{F}_p[A, x]$ in advance and evaluate them at a given $A \in \mathbb{F}_p$ at runtime. We estimate the number of operations required for both approaches.

Recursive definition. Ignoring multiplications by small fixed polynomials, the division polynomials satisfy a recursive equation of the form

$$f_\ell = f_a f_b f_c^2 - f_{a'} f_{b'} f_{c'}^2,$$

where the indices a, b, c, a', b', c' are integers within 2 of $\ell/2$ (so there are at most 5 distinct indices). Continuing this recursion involves indices within $2/1 + 2 = 3$ of $\ell/4$ (at most 7 distinct indices), within 3.5 of $\ell/8$ (at most 8), within 3.75 of $\ell/16$ (at most 8), etc.

Each of f_a, f_b, f_c has approximately $\ell^2/8$ coefficients, so computing $f_a f_b f_c^2$ costs $(2\mu(\ell^2/8) + \mu(\ell^2/4))\mathbf{M}$. The rough lower bound is $(\ell^2 \log_2 \ell)\mathbf{M}$, and the rough upper bound is $(3\ell^4/32)\mathbf{M}$.

Computing f_ℓ involves computing both $f_a f_b f_c^2$ and $f_{a'} f_{b'} f_{c'}^2$. The recursion involves at most 5 computations for $\ell/2$, at most 7 computations for $\ell/4$, and at most 8 computations for each subsequent level. The total is

$$(1 + 5/2 + 7/4 + 8/8 + 8/16 + \cdots)(2\ell^2 \log_2 \ell)\mathbf{M} = (29/2)(\ell^2 \log_2 \ell)\mathbf{M}$$

for the rough lower bound, and

$$(1 + 5/4 + 7/16 + 8/64 + 8/256 + \cdots)(3\ell^4/16)\mathbf{M} = (137/256)\ell^4\mathbf{M}$$

for the rough upper bound.

Evaluating precomputed polynomials. The degree of $\Psi_\ell \in \mathbb{F}_p[A, x]$ is $(\ell^2 - 1)/2$ in x and $\leq \ell^2/8 + 1$ in A, so overall Ψ_ℓ has at most about $\ell^4/16$ coefficients. Evaluating a precomputed $\Psi_\ell \in \mathbb{F}_p[x][A]$ at $A \in \mathbb{F}_p$ using Horner's method takes at most about $(\ell^4/16)\mathbf{M}$. This improves the rough upper bound.

Extracting the split part. As stated in Algorithm 9.1, extracting the part ψ'_ℓ of ψ_ℓ that splits over $\mathbb{F}_p$ amounts to computing $\gcd(X^p - X, \psi_\ell)$. The exponentiation $X^p \bmod \psi_\ell$ is computed using square-and-multiply with windows (similar to Appendix C.5), which uses about $\log_2 p$ squarings and about $(\log_2 p)/(\log_2 \log_2 p)$ multiplications. For simplicity we count this as a total of $1.2 \log_2 p$ multiplications, which is a reasonable estimate for 512-bit p.

For the number of $\mathbb{F}_p$-multiplications to compute $X^p \bmod \psi_\ell$ we obtain approximately $2.4 \log_2 p \cdot \ell^2 \log \ell$ for the lower bound on $\mu(d)$ and $0.6 \log_2 p \cdot \ell^4$ for the upper. Here we assume cost $\mu(d)$ for reducing a degree-$(2d-2)$ polynomial modulo a degree-d polynomial.

The computation of $\gcd\big((X^p \bmod \psi_\ell) - X, \psi_\ell\big)$ can be done using Stevin's algorithm which uses roughly $d^2\mathbf{M}$, where d is the degree of the larger of the input polynomials. In this case, since $\deg \psi_\ell \approx \ell^2/2$, this amounts to about $(\ell^4/4)\mathbf{M}$.

Table 9.1. Rough estimates for the number of $\mathbb{F}_p$-multiplications to compute $\mathcal{L}(A)$ using division polynomials (Algorithm 9.1).

ℓ	3	5	7	11	13	17	19	23	29
rough upper bound	$2^{15.1}$	$2^{17.8}$	$2^{19.6}$	$2^{22.1}$	$2^{23.1}$	$2^{24.6}$	$2^{25.3}$	$2^{26.4}$	$2^{27.7}$
rough lower bound	$2^{14.5}$	$2^{16.4}$	$2^{17.6}$	$2^{19.1}$	$2^{19.7}$	$2^{20.6}$	$2^{20.9}$	$2^{21.6}$	$2^{22.3}$

Fast arithmetic improves gcd computation to $O(d \cdot (\log_2 d)^2)\mathbf{M}$ asymptotically; see, e.g., [69]. We are not aware of literature presenting concrete speeds for fast constant-time gcd computation. For a rough lower bound we assume $2d(\log_2 d)^2\mathbf{M}$, i.e., about $4\ell^2(\log_2 \ell)^2\mathbf{M}$.

Extracting the kernel polynomial. Note that $\deg \psi'_\ell = \ell - 1$: Each root in $\mathbb{F}_p$ of ψ_ℓ gives rise to two points of order ℓ in the $+1$ or -1 Frobenius eigenspace, which contain $\ell - 1$ nonzero points each. Hence, as before, the cost of obtaining χ_ℓ from ψ'_ℓ is roughly $(2.4 \log_2 p \cdot \ell \log_2 \ell)\mathbf{M}$ resp. $2.4 \log_2 p \cdot \ell^2\mathbf{M}$ for the exponentiation, plus $2\ell(\log_2 \ell)^2\mathbf{M}$ resp. $\ell^2\mathbf{M}$ for the gcd computation.

Computing the isogeny. Lemma 9.1 is just a simple formula in terms of a few coefficients of χ_ℓ and can be realized using $2\mathbf{M}$ and some additions, hence has negligible cost.

9.3. Total cost. In summary, the cost of Algorithm 9.1 in $\mathbb{F}_p$-multiplications has a rough lower bound of

$$\min\left\{(29/2)\ell^2 \log_2 \ell, \ell^4/16\right\} + 2.4 \log_2 p \cdot \ell^2 \log_2 \ell + 4\ell^2(\log_2 \ell)^2 \\ + 2.4 \log_2 p \cdot \ell \log_2 \ell + 2\ell(\log_2 \ell)^2$$

and a rough upper bound of

$$\ell^4/16 + 0.6 \log_2 p \cdot \ell^4 + \ell^4/4 + 2.4 \log_2 p \cdot \ell^2 + \ell^2.$$

Table 9.1 lists values of these formulas for $\log_2 p \approx 512$ and small ℓ.

The main bottleneck is the computation of $X^p \bmod \psi_\ell$: for each bit of p there is a squaring modulo ψ_ℓ, a polynomial of degree $(\ell^2 - 1)/2$. For comparison, the scalar multiplication in Section 5 involves about $10\mathbf{M}$ for each bit of p, no matter how large ℓ is, but is not guaranteed to produce a point of order ℓ.

10 Computing ℓ-isogenies using modular polynomials

One technique suggested by De Feo, Kieffer, and Smith [42, 23] to compute the CRS group action is to use the (classical) *modular polynomials* $\Phi_\ell(X, Y)$, which vanish exactly on the pairs of j-invariants that are connected by a cyclic ℓ-isogeny. For prime ℓ, the polynomial $\Phi_\ell(X, Y)$ is symmetric and has degree $\ell + 1$

in the two variables, hence fixing one of the variables to some j-invariant and finding the roots of the resulting univariate polynomial suffices to find neighbours in the ℓ-isogeny graph.

The advantage of modular polynomials over division polynomials is that the degree $\ell + 1$ of $\Phi_\ell(j, Y)$ grows more slowly than the degree $(\ell^2 - 1)/2$ of the ℓ-division polynomial ψ_ℓ used in Section 9: modular polynomials are smaller for all $\ell \geq 5$. However, using modular polynomials requires solving two problems: disambiguating twists and disambiguating directions. We address these problems in the rest of this section.

10.1. Disambiguating twists. It may seem that the idea of computing ℓ-isogenous curves by finding roots of $\Phi_\ell(X, Y)$ is not applicable to the CSIDH setting, since a single j-invariant almost always defines *two* distinct nodes in the supersingular $\mathbb{F}_p$-rational isogeny graph, namely E_B and E_{-B} for some $B \in S_p$. Knowing $j(E_{\mathcal{L}(A)})$ is not enough information to distinguish $\mathcal{L}(A)$ from $-\mathcal{L}(A)$.

This problem does not arise in CRS. Twists always have the same j-invariant but, in the ordinary case, are not isogenous. A random twist point has negligible chance of being annihilated by the expected group order, so one can reliably recognize the twist at the expense of a scalar multiplication.

For CSIDH, one way to distinguish the cases $\mathcal{L}(A) = B$ and $\mathcal{L}(A) = -B$ is to apply a different isogeny-computation method (from, e.g., Section 5 or Section 9) to compute $\mathcal{L}(B)$. If $\mathcal{L}(B) = -A$ then $\mathcal{L}(A) = -B$; otherwise $\mathcal{L}(A) = B$.

This might seem to remove any possible advantage of having used modular polynomials to compute $\pm B$ in the first place, since one could simply have used the different method to compute $\mathcal{L}(A)$. However, below we will generalize the same idea to $\mathcal{L}^e(A)$, amortizing the costs of the different method across the costs of e computations using modular polynomials.

An alternative is as follows. The Bostan–Morain–Salvy–Schost algorithm [13], given a curve C (in short Weierstrass form, but the algorithm is easily adjusted to apply to Montgomery curves) and an ℓ-isogenous curve C', finds a formula for the unique normalized ℓ-isogeny from C to C'. Part of this formula is the kernel polynomial of the isogeny: the monic degree-$(\ell-1)$ polynomial $D \in \mathbb{F}_p[X]$ whose roots are the x-coordinates of the nonzero elements of the kernel of the isogeny. The algorithm uses $\ell^{1+o(1)}$ field operations with fast multiplication techniques. The output of the algorithm can be efficiently verified to be an ℓ-isogeny from C to C', so the algorithm can also be used to test whether two curves are ℓ-isogenous.

Use this algorithm to test whether there is an ℓ-isogeny from E_A to E_B, and, if so, to find the kernel polynomial D. Check whether D divides $X^p - X$, i.e., whether all of the nonzero elements of the kernel have x-coordinates defined over $\mathbb{F}_p$; this takes one exponentiation modulo D. Also check whether D divides $(X^3 + AX^2 + X)^{(p+1)/2} - (X^3 + AX^2 + X)$, i.e., whether all of the nonzero elements of the kernel have y-coordinates defined over $\mathbb{F}_p$. These tests are all passed if and only if $B = \mathcal{L}(A)$.

Both of these approaches also incur the cost of computing $B \in \mathbb{F}_p$ given $j(E_B)$, which we handle as follows. First note that there are at most two such B:

different Montgomery models of the same curve arise from the choice of point of order 2 which is moved to $(0,0)$; in our setting, there is only one rational order-2 point, hence B is unique up to sign. The j-invariant of E_B is an even rational function of degree 6 in B, hence solving for $B \in \mathbb{F}_p$ given $j(E_B)$ amounts to finding the $\mathbb{F}_p$-roots of a degree-6 polynomial $g \in \mathbb{F}_p[Y^2]$. To do so, we first compute $h = \gcd(Y^p - Y, g)$ to extract the split part; by the above h has degree 2. A solution $B \in \mathbb{F}_p$ can then be obtained by computing a square root.

We also mention a further possibility that appears to eliminate all of the costs above: replace the classical modular polynomials for j with modular polynomials for the Montgomery coefficient A. Starting with standard techniques to compute classical modular polynomials, and replacing j with A, appears to produce, at the same speed, polynomials that vanish exactly on the pairs (A, B) where E_A and E_B are connected by a cyclic ℓ-isogeny. The main cost here is in proof complexity: to guarantee that this approach works, one must switch from the well-known theory of classical modular polynomials to a suitable theory of Montgomery (or Edwards) modular polynomials.

10.2. Disambiguating directions. A further problem is that each curve has two neighbors in the ℓ-isogeny graph. The modular polynomial does not contain enough information to distinguish between the two neighbours. Specifically, the roots of $\Phi_\ell(j(E_A), Y)$ in $\mathbb{F}_p$ are $j(E_{\mathcal{L}(A)})$ and $j(E_{\mathcal{L}^{-1}(A)})$, which are almost always different. Switching from j-invariants to other geometric invariants does not solve this problem.

This is already a problem for CRS, and is already solved in [42, 23] using the Bostan–Morain–Salvy–Schost algorithm. The application of this algorithm in the CRS context is slightly simpler than the application explained above, since there is no need for isogeny verification: one knows that E_B is ℓ-isogenous to E_A, and the only question is whether the kernel is in the correct Frobenius eigenspace. For CSIDH, the question is whether the y-coordinates in the kernel are defined over $\mathbb{F}_p$.

10.3. Isogeny walks. We now consider the problem of computing $\mathcal{L}^e(A)$. As before, $\mathcal{L}^{-e}(A)$ can be computed as $-\mathcal{L}^e(-A)$, so we focus on the case $e > 0$.

After the first step $\mathcal{L}(A)$ has been computed (see above), identifying the correct direction in each subsequent step is easy, as pointed out in [23, Algorithm ElkiesWalk]. The point is that (except for degenerate cases) another step in the same direction never leads back to the previously visited curve; hence simply avoiding backward steps is enough. The cost of disambiguating directions is thus amortized across all e steps.

We also amortize the cost of disambiguating twists across all e steps as follows. We ascertain the correct direction at the first step. We then compute the sequence of j-invariants for all e steps. At the last step, we compute the corresponding Montgomery coefficient and ascertain the correct twist.

Algorithm 10.1 combines these ideas. For simplicity, the algorithm avoids the Bostan–Morain–Salvy–Schost algorithm. Instead it uses another isogeny-computation method, such as Algorithm 9.1, to disambiguate the direction at the first step and to disambiguate the twist at the last step.

Algorithm 10.1: Isogeny graph walking using modular polynomials.

```
Parameters: Odd primes $\ell_1 < \cdots < \ell_n$, a prime $p = 4\ell_1 \cdots \ell_n - 1$,
            $\ell \in \{\ell_1, \ldots, \ell_n\}$, and an integer $e \geq 1$.
Input: $A \in S_p$.
Output: $\mathcal{L}^e(A) \in S_p$.
Compute $B = \mathcal{L}(A)$ using another algorithm.
Set $j_{prev} = j(E_A)$ and $j_{cur} = j(E_B)$.
for $i \leftarrow 2$ to $e$ do
    Compute $f \leftarrow \gcd(Y^p - Y, \Phi_\ell(j_{cur}, Y))$.
    Let $c, d \in \mathbb{F}_p$ be the coefficients of $f$, such that $f = Y^2 + cY + d$.
    Set $(j_{prev}, j_{cur}) \leftarrow (j_{cur}, c - j_{prev})$.
Find $B \in \mathbb{F}_p$ such that $j(E_B) = j_{cur}$.
Compute $C = \mathcal{L}(B)$ using another algorithm.
Set $B \leftarrow -B$ if $j(E_C) = j_{prev}$.
Return $B$.
```

The correctness of Algorithm 10.1 is best explained through the graph picture: Recall that the ℓ-isogeny graph (labelled by A-coefficients) is a disjoint union of cycles which have a natural orientation given by the map $\mathcal{L}$.

Since $-\mathcal{L}(-A) = \mathcal{L}^{-1}(A)$, negating all labels in a cycle $\mathcal{C}$ corresponds to inverting the orientation of the cycle. For a cycle $\mathcal{C}$ as above, let $\mathcal{C}/\pm$ denote the quotient graph of $\mathcal{C}$ by negation. This is the same thing as applying j-invariants to all nodes. If $\mathcal{C}$ contains 0, then $\mathcal{C}/\pm$ is a line with inflection points at the ends; else $\mathcal{C}/\pm$ has the same structure as $\mathcal{C}$. In both cases $\mathcal{C}/\pm$ is unoriented.

For brevity, write $j_i = j(E_{\mathcal{L}^i(A)})$. Algorithm 10.1 starts out on a cycle $\mathcal{C}$ as above by computing one step $\mathcal{L}$ with known-good orientation. It then reduces to $\mathcal{C}/\pm$ and continues walking in the same direction simply by avoiding backwards steps when possible; there are only (up to) two neighbours at all times. Therefore, the property $j_{cur} = j_i$ holds at the end of each iteration of the loop; in particular, arbitrarily lifting j_e to a node with the right j-invariant yields $B \in \{\pm\mathcal{L}^e(A)\}$. Finally, computing and comparing $j(E_{\mathcal{L}(B)}) = j(E_{\mathcal{L}(\pm\mathcal{L}^e(A))}) = j_{e\pm1}$ to the value j_{e-1} known from the previous iteration of the loop reveals the correct sign.

10.4. Cost. Algorithm 10.1 requires two calls to a separate subroutine for $\mathcal{L}$ and some extra work (computing a p^{th} power modulo a degree-6 polynomial), so it is never faster than repeated applications of the separate subroutine when $e \leq 2$. On the other hand, replacing this subroutine with the Bostan–Morain–Salvy–Schost algorithm, and/or replacing classical modular polynomials with modular polynomials for A, might make this approach competitive for $e = 2$ and perhaps even $e = 1$.

No matter how large e is, Algorithm 10.1 requires computing the polynomials $\gcd(Y^p - Y, g)$ and $\gcd(Y^p - Y, \Phi_\ell(j_{cur}, Y))$ for each isogeny. The degree of g is smaller than in Algorithm 9.1 for $\ell \geq 5$, but the gcd cost quickly becomes much more expensive than the "Vélu" method from Section 5 as ℓ grows. However, this algorithm may nevertheless be of interest for small values of ℓ. If Algorithm 9.1 (rather than the Vélu method) is used as the separate $\mathcal{L}$ subroutine then Algorithm 10.1 is deterministic and always works.

References

[1] Reza Azarderakhsh, David Jao, Kassem Kalach, Brian Koziel, and Christopher Leonardi. Key compression for isogeny-based cryptosystems. In *AsiaPKC@AsiaCCS*, pages 1–10. ACM, 2016. https://ia.cr/2016/229.

[2] Paul Barrett. Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor. In *CRYPTO*, volume 263 of *Lecture Notes in Computer Science*, pages 311–323. Springer, 1986.

[3] Charles H. Bennett. Logical reversibility of computation. *IBM Journal of Research and Development*, 17:525–532, 1973.

[4] Charles H. Bennett. Time/space trade-offs for reversible computation. *SIAM J. Comput.*, 18(4):766–776, 1989.

[5] Daniel J. Bernstein. Curve25519: New Diffie-Hellman speed records. In *Public Key Cryptography*, volume 3958 of *Lecture Notes in Computer Science*, pages 207–228. Springer, 2006. https://cr.yp.to/papers.html#curve25519.

[6] Daniel J. Bernstein. Batch binary Edwards. In *CRYPTO*, volume 5677 of *Lecture Notes in Computer Science*, pages 317–336. Springer, 2009. https://cr.yp.to/papers.html#bbe.

[7] Daniel J. Bernstein, Mike Hamburg, Anna Krasnova, and Tanja Lange. Elligator: elliptic-curve points indistinguishable from uniform random strings. In *ACM Conference on Computer and Communications Security*, pages 967–980. ACM, 2013. https://ia.cr/2013/325.

[8] Daniel J. Bernstein and Tanja Lange. Analysis and optimization of elliptic-curve single-scalar multiplication. In *Finite fields and applications. Proceedings of the eighth international conference on finite fields and applications, Melbourne, Australia, July 9–13, 2007*, pages 1–19. Providence, RI: American Mathematical Society (AMS), 2008. https://ia.cr/2007/455.

[9] Daniel J. Bernstein and Tanja Lange. Montgomery curves and the Montgomery ladder. In Joppe W. Bos and Arjen K. Lenstra, editors, *Topics in computational number theory inspired by Peter L. Montgomery*, pages 82–115. Cambridge University Press, 2017. https://ia.cr/2017/293.

[10] Xavier Bonnetain and María Naya-Plasencia. Hidden shift quantum cryptanalysis and implications. In *ASIACRYPT*, volume 11274 of *Lecture Notes in Computer Science*. Springer, 2018. https://ia.cr/2018/432.

[11] Xavier Bonnetain and André Schrottenloher. Quantum security analysis of CSIDH and ordinary isogeny-based schemes, 2018. IACR Cryptology ePrint Archive 2018/537. https://ia.cr/2018/537.

[12] Joppe W. Bos. Constant time modular inversion. *J. Cryptographic Engineering*, 4(4):275–281, 2014. http://www.joppebos.com/files/CTInversion.pdf.

[13] Alin Bostan, François Morain, Bruno Salvy, and Éric Schost. Fast algorithms for computing isogenies between elliptic curves. *Math. Comput.*, 77(263):1755–1778, 2008. https://www.ams.org/journals/mcom/2008-77-263/S0025-5718-08-02066-8/S0025-5718-08-02066-8.pdf.

[14] Richard P. Brent and Hsiang-Tsung Kung. The area-time complexity of binary multiplication. *J. ACM*, 28(3):521–534, 1981. https://maths-people.anu.edu.au/~brent/pd/rpb055.pdf.

[15] Wouter Castryck, Tanja Lange, Chloe Martindale, Lorenz Panny, and Joost Renes. CSIDH: An efficient post-quantum commutative group action. In *ASIACRYPT*, volume 11274 of *Lecture Notes in Computer Science*, pages 395–427. Springer, 2018. https://ia.cr/2018/383.

[16] Yiping Cheng. Space-efficient Karatsuba multiplication for multi-precision integers. *CoRR*, abs/1605.06760, 2016. https://arxiv.org/abs/1605.06760.

[17] Andrew M. Childs, David Jao, and Vladimir Soukharev. Constructing elliptic curve isogenies in quantum subexponential time. *J. Mathematical Cryptology*, 8(1):1–29, 2014. https://arxiv.org/abs/1012.4019.

[18] Craig Costello and Hüseyin Hisil. A simple and compact algorithm for SIDH with arbitrary degree isogenies. In *ASIACRYPT (2)*, volume 10625 of *Lecture Notes in Computer Science*, pages 303–329. Springer, 2017. https://ia.cr/2017/504.

[19] Craig Costello, David Jao, Patrick Longa, Michael Naehrig, Joost Renes, and David Urbanik. Efficient compression of SIDH public keys. In *EUROCRYPT (1)*, volume 10210 of *Lecture Notes in Computer Science*, pages 679–706, 2017.

[20] Jean-Marc Couveignes. Hard Homogeneous Spaces, 2006. IACR Cryptology ePrint Archive 2006/291. https://ia.cr/2006/291.

[21] Luigi Dadda. Some schemes for parallel multipliers. *Alta frequenza*, 34(5):349–356, 1965.

[22] Luca De Feo, David Jao, and Jérôme Plût. Towards quantum-resistant cryptosystems from supersingular elliptic curve isogenies. *Journal of Mathematical Cryptology*, 8(3):209–247, 2014. IACR Cryptology ePrint Archive 2011/506. https://ia.cr/2011/506.

[23] Luca De Feo, Jean Kieffer, and Benjamin Smith. Towards practical key exchange from ordinary isogeny graphs. In *ASIACRYPT*, volume 11274 of *Lecture Notes in Computer Science*. Springer, 2018. https://ia.cr/2018/485.

[24] Vassil S. Dimitrov, Laurent Imbert, and Andrew Zakaluzny. Multiplication by a constant is sublinear. In *18th IEEE Symposium on Computer Arithmetic (ARITH-18 2007), 25-27 June 2007, Montpellier, France*, pages 261–268. IEEE Computer Society, 2007. http://www.lirmm.fr/~imbert/pdfs/constmult_arith18.pdf.

[25] Austin G. Fowler, Matteo Mariantoni, John M. Martinis, and Andrew N. Cleland. Surface codes: Towards practical large-scale quantum computation. *Physical Review A*, 86:032324, 2012. https://arxiv.org/abs/1208.0928.

[26] Martin Fürer. Faster integer multiplication. In *STOC*, pages 57–66. ACM, 2007.

[27] Craig Gidney. Halving the cost of quantum addition. *Quantum*, 2:74, 2017. https://quantum-journal.org/papers/q-2018-06-18-74/.

[28] Markus Grassl, Brandon Langenberg, Martin Roetteler, and Rainer Steinwandt. Applying Grover's algorithm to AES: quantum resource estimates. In *PQCrypto*, volume 9606 of *Lecture Notes in Computer Science*, pages 29–43. Springer, 2016. https://arxiv.org/abs/1512.04965.

[29] James L. Hafner and Kevin S. McCurley. A rigorous subexponential algorithm for computation of class groups. *J. Amer. Math. Soc.*, 2(4):837–850, 1989. https://www.ams.org/journals/jams/1989-02-04/S0894-0347-1989-1002631-0/S0894-0347-1989-1002631-0.pdf.

[30] Thomas Häner, Martin Roetteler, and Krysta M. Svore. Factoring using $2n+2$ qubits with Toffoli based modular multiplication. *Quantum Information & Computation*, 17(7&8):673–684, 2017. https://arxiv.org/abs/1611.07995.

[31] David Harvey and Joris van der Hoeven. Faster integer multiplication using short lattice vectors. *CoRR*, abs/1802.07932, 2018. https://arxiv.org/abs/1802.07932.

[32] David Harvey, Joris van der Hoeven, and Grégoire Lecerf. Even faster integer multiplication. *J. Complexity*, 36:1–30, 2016. https://arxiv.org/abs/1407.3360.

[33] David Harvey, Joris van der Hoeven, and Grégoire Lecerf. Faster polynomial multiplication over finite fields. *J. ACM*, 63(6):52:1–52:23, 2017. https://arxiv.org/abs/1407.3361.

[34] Hüseyin Hişil. *Elliptic curves, group law, and efficient computation*. PhD thesis, Queensland University of Technology, 2010. https://eprints.qut.edu.au/33233/.

[35] Michael Hutter and Peter Schwabe. Multiprecision multiplication on AVR revisited. *J. Cryptographic Engineering*, 5(3):201–214, 2015. https://ia.cr/2014/592.

[36] David Jao, Reza Azarderakhsh, Matthew Campagna, Craig Costello, Luca De Feo, Basil Hess, Amir Jalali, Brian Koziel, Brian LaMacchia, Patrick Longa, Michael Naehrig, Joost Renes, Vladimir Soukharev, and David Urbanik. SIKE. Submission to [55]. http://sike.org.

[37] David Jao and Luca De Feo. Towards quantum-resistant cryptosystems from supersingular elliptic curve isogenies. In *PQCrypto*, volume 7071 of *Lecture Notes in Computer Science*, pages 19–34. Springer, 2011. https://ia.cr/2011/506/20110918:024142.

[38] David Jao, Jason LeGrow, Christopher Leonardi, and Luis Ruiz-Lopez. A subexponential-time, polynomial quantum space algorithm for inverting the CM group action. *Journal of Mathematical Cryptology*, 2018. To appear.

[39] Cody Jones. Low-overhead constructions for the fault-tolerant Toffoli gate. *Physical Review A*, 87:022328, 2012.

[40] Anatoly A. Karatsuba and Y. Ofman. Multiplication of multidigit numbers on automata. *Soviet Physics Doklady*, 7:595–596, 1963.

[41] Shane Kepley and Rainer Steinwandt. Quantum circuits for $\mathbb{F}_{2^n}$-multiplication with subquadratic gate count. *Quantum Information Processing*, 14(7):2373–2386, 2015.

[42] Jean Kieffer. *Étude et accélération du protocole d'échange de clés de Couveignes–Rostovtsev–Stolbunov*. Mémoire du Master 2, Université Paris VI, 2017. https://arxiv.org/abs/1804.10128.

[43] Emanuel Knill. An analysis of Bennett's pebble game. *CoRR*, abs/math/9508218, 1995. https://arxiv.org/abs/math/9508218.

[44] Donald E. Knuth. *The Art of Computer Programming, Volume II: Seminumerical Algorithms, 2nd Edition*. Addison-Wesley, 1981.

[45] David Kohel. *Endomorphism rings of elliptic curves over finite fields*. PhD thesis, University of California at Berkeley, 1996. http://iml.univ-mrs.fr/~kohel/pub/thesis.pdf.

[46] Greg Kuperberg. A subexponential-time quantum algorithm for the dihedral hidden subgroup problem. *SIAM J. Comput.*, 35(1):170–188, 2005. https://arxiv.org/abs/quant-ph/0302112.

[47] Greg Kuperberg. Another subexponential-time quantum algorithm for the dihedral hidden subgroup problem. In *TQC*, volume 22 of *LIPIcs*, pages 20–34. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2013. https://arxiv.org/abs/1112.3333.

[48] Vincent Lefèvre. Multiplication by an integer constant: Lower bounds on the code length. In *5th Conference on Real Numbers and Computers 2003 - RNC5*, pages 131–146, Lyon, France, 2003. https://hal.inria.fr/inria-00072095v1.

[49] Michael Meyer and Steffen Reith. A faster way to the CSIDH, 2018. IACR Cryptology ePrint Archive 2018/782. https://ia.cr/2018/782.

[50] Daniele Micciancio. Improving lattice based cryptosystems using the Hermite normal form. In *CaLC*, volume 2146 of *Lecture Notes in Computer Science*, pages 126–145. Springer, 2001. https://cseweb.ucsd.edu/~daniele/papers/HNFcrypt.html.

[51] Victor S. Miller. Use of elliptic curves in cryptography. In *CRYPTO*, volume 218 of *Lecture Notes in Computer Science*, pages 417–426. Springer, 1985.

[52] Peter L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44:519–521, 1985. http://www.ams.org/journals/mcom/1985-44-170/S0025-5718-1985-0777282-X/home.html.

[53] Peter L. Montgomery. Speeding the Pollard and elliptic curve methods of factorization. *Mathematics of Computation*, 48(177):243–264, 1987. https://www.ams.org/journals/mcom/1987-48-177/S0025-5718-1987-0866113-7/.

[54] Dustin Moody and Daniel Shumow. Analogues of Vélu's formulas for isogenies on alternate models of elliptic curves. *Math. Comput.*, 85(300):1929–1951, 2016. https://ia.cr/2011/430.

[55] National Institute of Standards and Technology. Post-quantum cryptography standardization, December 2016. https://csrc.nist.gov/Projects/Post-Quantum-Cryptography/Post-Quantum-Cryptography-Standardization.

[56] Peter J. Nicholson. Algebraic theory of finite Fourier transforms. *J. Comput. Syst. Sci.*, 5(5):524–547, 1971.

[57] Alex Parent, Martin Roetteler, and Michele Mosca. Improved reversible and quantum circuits for Karatsuba-based integer multiplication. In *TQC*, volume 73 of *LIPIcs*, pages 7:1–7:15. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017. https://arxiv.org/abs/1706.03419.

[58] Christophe Petit. Faster algorithms for isogeny problems using torsion point images. In *ASIACRYPT (2)*, volume 10625 of *Lecture Notes in Computer Science*, pages 330–353. Springer, 2017. https://ia.cr/2017/571.

[59] Julia Pieltant and Hugues Randriam. New uniform and asymptotic upper bounds on the tensor rank of multiplication in extensions of finite fields. *Math. Comput.*, 84(294):2023–2045, 2015. https://arxiv.org/abs/1305.5166.

[60] John M. Pollard. The fast Fourier transform in a finite field. *Mathematics of Computation*, 25:365–374, 1971. https://www.ams.org/journals/mcom/1971-25-114/S0025-5718-1971-0301966-0/.

[61] Oded Regev. A subexponential time algorithm for the dihedral hidden subgroup problem with polynomial space, 2004. https://arxiv.org/abs/quant-ph/0406151.

[62] Joost Renes. Computing isogenies between Montgomery curves using the action of $(0,0)$. In *PQCrypto*, volume 10786 of *Lecture Notes in Computer Science*, pages 229–247. Springer, 2018. https://ia.cr/2017/1198.

[63] Martin Roetteler, Michael Naehrig, Krysta M. Svore, and Kristin E. Lauter. Quantum resource estimates for computing elliptic curve discrete logarithms. In *ASIACRYPT (2)*, volume 10625 of *Lecture Notes in Computer Science*, pages 241–270. Springer, 2017. https://ia.cr/2017/598.

[64] Alexander Rostovtsev and Anton Stolbunov. Public-key cryptosystem based on isogenies, 2006. IACR Cryptology ePrint Archive 2006/145. https://ia.cr/2006/145.

[65] Arnold Schönhage. Schnelle Multiplikation von Polynomen über Körpern der Charakteristik 2. *Acta Inf.*, 7:395–398, 1977.

[66] Arnold Schönhage and Volker Strassen. Schnelle Multiplikation großer Zahlen. *Computing*, 7(3-4):281–292, 1971.

[67] Peter W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM J. Comput.*, 26(5):1484–1509, 1997. https://arxiv.org/abs/quant-ph/9508027.

[68] Anton Stolbunov. Constructing public-key cryptographic schemes based on class group action on a set of isogenous elliptic curves. *Adv. in Math. of Comm.*, 4(2):215–235, 2010.

[69] Volker Strassen. The computational complexity of continued fractions. *SIAM Journal on Computing*, 12:1–27, 1983.

[70] Mehdi Tibouchi. Elligator squared: Uniform points on elliptic curves of prime order as uniform random strings. In *Financial Cryptography*, volume 8437 of *Lecture Notes in Computer Science*, pages 139–156. Springer, 2014. https://ia.cr/2014/043.

[71] Andrei L. Toom. The complexity of a scheme of functional elements realizing the multiplication of integers. *Soviet Mathematics Doklady*, 3:714–716, 1963. http://toomandre.com/my-articles/engmat/MULT-E.PDF.

[72] Jacques Vélu. Isogénies entre courbes elliptiques. *Comptes Rendus de l'Académie des Sciences de Paris*, 273:238–241, 1971.

[73] Christopher S. Wallace. A suggestion for a fast multiplier. *IEEE Transactions on electronic Computers*, (1):14–17, 1964.

[74] Herbert S. Wilf. *generatingfunctionology*. Academic Press, 1994. https://www.math.upenn.edu/~wilf/DownldGF.html.

[75] Gustavo Zanon, Marcos A. Simplício Jr., Geovandro C. C. F. Pereira, Javad Doliskani, and Paulo S. L. M. Barreto. Faster isogeny-based compressed key agreement. In *PQCrypto*, volume 10786 of *Lecture Notes in Computer Science*, pages 248–268. Springer, 2018. https://ia.cr/2017/1143.

A Cost metrics for quantum computation

This appendix reviews several cost metrics relevant to this paper.

A.1. Bit operations. Computations on today's non-quantum computers are ultimately nothing more than sequences of bit operations. The hardware carries out a sequence of NOT gates $b \mapsto 1 - b$; AND gates $(a, b) \mapsto ab$; OR gates $(a, b) \mapsto \max\{a, b\}$; and XOR gates $(a, b) \mapsto a \oplus b$. Some of the results are displayed as outputs.

Formally, a computation is a finite directed acyclic graph where each node has 0, 1, or 2 inputs. Each 0-input node in the graph is labeled as constant 0, constant 1, or a specified input bit. Each 1-input node in the graph is labeled NOT. Each 2-input node in the graph is labeled AND, OR, or XOR. There is also a labeling of output bits as particular nodes in the graph.

The graph induces a function from sequences of input bits to sequences of output bits. Specifically, given values of the input bits, the graph assigns a value to each node as specified by the label (e.g., the value at an AND node is the minimum of the values of its two input nodes), and in particular computes values of the output bits.

Our primary cost metric in this paper is the number of **nonlinear bit operations**: i.e., we count the number of ANDs and ORs, disregarding the number of NOTs and XORs (and 0s and 1s). The advantage of choosing this cost metric is comparability to the Toffoli cost metric used in, e.g., [30] and [63], which in turn is motivated by current estimates of the costs of various quantum operations, as we explain below.

A potential disadvantage of choosing this cost metric is that the cost metric can hide arbitrarily large sequences of linear operations. For example, there are known algorithms to multiply n-coefficient polynomials in $\mathbb{F}_2[x]$ using $\Theta(n)$ nonlinear operations (see, e.g., [59]), but this operation count hides $\Theta(n^2)$ linear operations. Other algorithms using $n(\log n)^{1+o(1)}$ total bit operations (see, e.g., [65] and [33]) are much better when n is large, even though they have many more nonlinear bit operations.

This seems to be less of an issue for integer arithmetic than for polynomial arithmetic. Adding nonzero costs for NOT and XOR requires a reevaluation of, e.g., the quantitative cutoff between schoolbook multiplication and Karatsuba multiplication, but does not seem to have broader qualitative impacts on the speedups that we consider in this paper. Similarly, our techniques can easily be adapted to, e.g., a cost metric that allows NAND gates with lower cost than AND gates, reflecting the reality of computer hardware.

A.2. The importance of constant-time computations. One can object to the very simple model of computation explained above as not allowing variable-time computations. The graph uses a constant number of bit operations to produce its outputs, whereas real users often wait input-dependent amounts of time for the results of a computation. If a particular input is processed faster than the worst case, then the time saved can be spent on other useful computations.

However, our primary goal in this paper is to evaluate the cost of carrying out a CSIDH group action on a huge number of inputs in quantum superposition. Operations are carried out on all of the inputs simultaneously, and then a measurement retroactively selects a particular input. The cost depends on the number of operations carried out on all inputs, not on the number of operations that in retrospect could have been carried out for the selected input.

The same structure has an impact at every level of algorithm design. In conventional algorithm design, if a function calls subroutine X for some inputs and subroutine Y for other inputs, then the cost of the function is the *maximum* of the costs of X and Y. However, a computation graph does not allow this branching. One must instead compute a suitable combination such as

$$bX(\text{inputs}) + (1-b)Y(\text{inputs}),$$

taking the *total* time for X and Y, or search for ways to overlap portions of the computations of X and Y.

One can provide branches as a higher-level abstraction by building a computation graph that manipulates an input-dependent pointer into an array of instructions, imitating the way that CPU hardware is built. It is important to realize, however, that the number of bit operations required to read an instruction from a variable location in an array grows with the size of the array, so the total number of bit operations in this approach grows much more rapidly than the number of instructions. A closer look at what actually needs to be computed drastically reduces the number of bit operations.

The speedup techniques considered in this paper can also be used in constant-time non-quantum software and hardware for CSIDH, reducing the cost of protecting CSIDH users against timing attacks. However, our main focus is the quantum case.

A.3. Reversible bit operations. Bits cannot be erased or copied inside a quantum computation. For example, one cannot simply compute a XOR gate, replacing (a, b) with $a \oplus b$, or an AND gate, replacing (a, b) with ab. However, one can compute a "CNOT" gate, replacing (a, b) with $(a, a \oplus b)$; or a "Toffoli" gate, replacing (a, b, c) with $(a, b, c \oplus ab)$.

In general, an ***n*-bit reversible computation** begins with a list of n input bits, and then applies a sequence of NOT gates, CNOT gates, and Toffoli gates to specified positions in the list, eventually producing n output bits. Each of these gates is its own inverse, so one can map output back to input by applying the same gates in the reverse order.

Bennett's conversion (see [3], which handles the more complicated case of Turing machines) is a generic transformation from computations, as defined in Appendix A.1, to reversible computations. Say the original computation maps $x \in \{0,1\}^k$ to $F(x) \in \{0,1\}^\ell$. The reversible computation then maps $(x, y, 0) \in \{0,1\}^{k+\ell+m}$ to $(x, y \oplus F(x), 0) \in \{0,1\}^{k+\ell+m}$, for some choice of m that will be clear in a moment; the m auxiliary zero bits are called **ancillas**. The effect of the reversible computation upon more general inputs $(x, y, z) \in \{0,1\}^{k+\ell+m}$ is more complicated, and usually irrelevant.

For each AND gate $(a, b) \mapsto ab$ in the original computation, the reversible computation allocates an ancilla c and performs $(a, b, c) \mapsto (a, b, c \oplus ab)$ as a Toffoli gate. Note that if the ancilla c begins as 0 then this Toffoli gate produces the desired bit ab. More generally, for each gate in the original computation, the reversible computation allocates an ancilla c and operates reversibly on this ancilla, in such a way that if the ancilla begins with 0 then it ends with the same bit computed by the original gate. For example:

- For each constant-1 gate $() \mapsto 1$, the reversible computation allocates an ancilla c and performs a NOT gate $c \mapsto 1 - c$.
- For each NOT gate $b \mapsto 1-b$, the reversible computation allocates an ancilla c and performs $(b, c) \mapsto (b, c \oplus 1 \oplus b)$ as a NOT gate and a CNOT gate.
- For each XOR gate $(a, b) \mapsto a \oplus b$, the reversible computation allocates an ancilla c and performs $(a, b, c) \mapsto (a, b, c \oplus a \oplus b)$ as two CNOT gates.

The reversible computation thus maps $(x, y, 0)$ to (x, y, z) where z is the entire sequence of bits in the original computation, including all intermediate results. In particular, z includes the bits of $F(x)$, and ℓ additional CNOT gates produce

$(x, y \oplus F(x), z)$. Finally, re-running the computation of z in reverse order has the effect of "uncomputing" z, producing $(x, y \oplus F(x), 0)$ as claimed.

The number of Toffoli gates here is exactly twice the number of nonlinear bit operations in the original computation: once in computing z and once in uncomputing z. There is a larger expansion in the number of NOT and CNOT gates compared to the original number of linear bit operations, but, as mentioned earlier, we focus on nonlinear bit operations.

Sometimes these overheads can be reduced. For example, if the original computation is simply an AND $(a, b) \mapsto ab$, then the reversible computation stated above uses two Toffoli gates and one ancilla—

- $(a, b, y, 0) \mapsto (a, b, y, ab)$ with a Toffoli gate,
- $(a, b, y, ab) \mapsto (a, b, y \oplus ab, ab)$ with a CNOT gate,
- $(a, b, y \oplus ab, ab) \mapsto (a, b, y \oplus ab, 0)$ with another Toffoli gate,

—but it is better to simply compute $(a, b, y) \mapsto (a, b, y \oplus ab)$ with one Toffoli gate and no ancillas. We do not claim that the optimal number of bit operations is a perfect predictor of the optimal number of Toffoli gates; we simply use the fact that the ratio is between 1 and 2.

Note that Bennett's reversible computation operates on an n-bit state where $n = k + \ell + m$ is essentially the number of *bit operations* in the original computation. Perhaps the original computation can fit into a much smaller state (depending on the order of operations, something not expressed by the computation graph), but this often relies on erasing intermediate results, which a reversible computation cannot do. Even in a world where arbitrarily large quantum computers can be built, this number n has an important impact on the cost of the corresponding quantum computation, so it becomes important to consider ways to reduce n, as explained in Appendix A.5.

A.4. *T*-gates. The state of n qubits is, by definition, a nonzero element $(v_0, v_1, \ldots)$ of the vector space $\mathbb{C}^{2^n}$. Measuring these n qubits produces an n-bit index $i \in \{0, 1, \ldots, 2^n - 1\}$, while modifying the vector to have 1 at position i and 0 elsewhere. The chance of obtaining i is proportional to $|v_i|^2$. One can, if desired, normalize the vectors so that $\sum_i |v_i|^2 = 1$.

An ***n*-qubit quantum computation** applies a sequence of NOT (often written "X"), CNOT, Hadamard ("H"), T, and T^{-1} gates to specified positions within n qubits. There is a standard representation of these gates as the matrices

$$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, \begin{pmatrix} 1&0&0&0 \\ 0&1&0&0 \\ 0&0&0&1 \\ 0&0&1&0 \end{pmatrix}, \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}, \begin{pmatrix} 1 & 0 \\ 0 & \exp(i\pi/4) \end{pmatrix}, \begin{pmatrix} 1 & 0 \\ 0 & \exp(-i\pi/4) \end{pmatrix}$$

respectively; if vectors are normalized then the Hadamard matrix is divided by $\sqrt{2}$. There is also a standard way to interpret these matrices as acting upon vectors in $\mathbb{C}^{2^n}$. For example, applying the NOT gate to qubit 0 of $(v_0, v_1, v_2, v_3, \ldots)$

produces $(v_1, v_0, v_3, v_2, \ldots)$; measuring after the NOT has the same effect as measuring before the NOT and then complementing bit 0 of the result. Applying the NOT gate to qubit 1 of $(v_0, v_1, v_2, v_3, \ldots)$ produces $(v_2, v_3, v_0, v_1, \ldots)$.

The consensus of quantum-computer engineers appears to be that "Clifford operations" such as NOT, CNOT, H, T^2, and T^{-2} are at least two orders of magnitude less expensive than T and T^{-1}. It is thus common practice to allow T^2 ("S" or "P") and T^{-2} as further gates, and to count the number of T and T^{-1}, while disregarding the number of NOT, CNOT, H, T^2, and T^{-2}. The total number of T and T^{-1} is, by definition, the number of ***T*-gates**.

There is a standard conversion from an n-bit reversible computation to an n-qubit quantum computation. NOT is converted to NOT; CNOT is converted to CNOT; Toffoli is converted to a sequence of 7 T-gates and some Clifford gates. Multiplying 7 by an upper bound on the number of Toffoli gates thus produces an upper bound on the number of T-gates.

As in Appendix A.3, these overheads can sometimes be reduced. For example:

- All of the quantum gates mentioned here operate on one or two qubits at a time. The intermediate results in a Toffoli computation can often be reused for other computations.
- In a more sophisticated model of quantum computation that allows internal measurements, Jones [39] showed how to implement a Toffoli gate as 4 T-gates, some Clifford gates, and a measurement. We follow [28] in mentioning but disregarding this alternative.
- In the same model, a recent paper by Gidney [27] showed how to implement n-bit integer addition using about $4n$ T-gates (and a similar number of Clifford gates and measurements). For comparison, a standard "ripple carry" adder uses about $2n$ nonlinear bit operations.

As before, we do not claim that the optimal number of Toffoli gates is a perfect predictor of the number of T-gates; we simply use the fact that the ratio is between 1 and 7.

A.5. Error-correction steps; the importance of parallelism. To recap: Our primary focus is producing an upper bound on the number of nonlinear bit operations. Multiplying by 2 gives an upper bound on the number of Toffoli gates for a reversible computation, and multiplying this second upper bound by 7 gives an upper bound on the number of T-gates for a quantum computation. Linear bit operations (and the corresponding reversible and quantum gates) do not seem to be a bottleneck for the types of computations considered in this paper.

There is, however, a much more important bottleneck that is ignored in these cost metrics: namely, fault-tolerance seems to require continual error correction of every stored qubit.

Surface codes [25] are the leading candidates for fault-tolerant quantum computation. A logical qubit is encoded in a particular way as many entangled physical qubits spread over a surface. *Some* of the physical qubits are continually measured, and operations are carried out on the physical qubits to correct any errors revealed by the measurements. The consensus of the literature appears to be that performing a computation on a logical qubit will be only a small constant factor more expensive than storing an idle logical qubit.

One consequence of this structure is that all fault-tolerant quantum computations involve entanglement throughout the entire computation, contrary to the claim in [11] that a particular quantum algorithm "does not need to have a highly entangled memory for a long time".

Another consequence of this structure is that the cost of a quantum computation can grow *quadratically* with the number of bit operations. Consider, for example, an n-bit ripple-carry adder, or the adder from [27]. This computation involves $\Theta(n)$ sequential bit operations and finishes in time $\Theta(n)$. Each of the $\Theta(n)$ qubits needs active error correction at each time step, for a total of $\Theta(n^2)$ error-correction steps.

The product of computer size and time is typically called "area-time product" or "*AT*" in the literature on non-quantum computation; "volume" in the literature on quantum computation; and "price-performance ratio" in the literature on economics. The cost of quantum error correction is not the only argument for viewing this product as the true cost of computation: there is a more fundamental argument stating that the total cost assigned to two separate computations should not depend on whether the computations are carried out in serial (using hardware for twice as much time) or in parallel (using twice as much hardware).

From this perspective, it is much better to use parallel algorithms for integer addition that finish in time $\Theta(\log n)$. This still means $\Theta(n \log n)$ error-correction steps, so the cost is larger by a factor $\Theta(\log n)$ than the number of bit operations.

At a higher level, the CSIDH computation involves various layers for which highly parallel algorithms are not known. For example, modular exponentiation is notoriously difficult to parallelize. A conventional computation of $x \bmod n$, $x^2 \bmod n$, $x^4 \bmod n$, $x^8 \bmod n$, etc. can store each intermediate result on top of the previous result, but Bennett's conversion produces a reversible computation that uses much more storage, and the resulting quantum computation requires continual error correction for all of the stored qubits. Shor's algorithm avoids this issue because it computes a superposition of powers of a *constant* x; this is not helpful in the CSIDH context.

Bennett suggested reducing the number of intermediate results in a reversible computation by checkpointing the computation halfway through:

- Compute the middle as a function of the beginning.
- Uncompute intermediate results, leaving the beginning and the middle.
- Compute the end as a function of the middle.
- Uncompute intermediate results, leaving the beginning, middle, and end.
- Recompute the middle from the beginning.
- Uncompute intermediate results, leaving the beginning and end.

This multiplies the number of qubits by about 0.5 but multiplies the number of gates by about 1.5. See [4] and [43] for analyses of further tradeoffs along these lines.

This paper focuses on bit operations, as noted above. Beyond this, Appendix C.6 makes some remarks on the number of qubits required for our computations. We have not attempted to analyze the time required for a parallel computation using a specified number of qubits.

A.6. Error-correction steps on a two-dimensional mesh. There is a further problem with counting bit operations: in many computations, the main bottleneck is communication.

For example, FFT-based techniques multiply n-bit integers using $n^{1+o(1)}$ bit operations, and can be parallelized to use time just $n^{o(1)}$ with area $n^{1+o(1)}$. However, Brent and Kung showed [14] that integer multiplication on a two-dimensional mesh of area $n^{1+o(1)}$ requires time $n^{0.5+o(1)}$, even in a model where information travels instantaneously through arbitrarily long wires.

Plausible architectures for fault-tolerant quantum computation, such as [25], are built from near-neighbor interactions on a two-dimensional mesh. Presumably, as in [14], $n^{1+o(1)}$ qubits computing an n-bit product require time $n^{0.5+o(1)}$, and thus $n^{1.5+o(1)}$ error-correction steps. One might hope for quantum teleportation to avoid some of the bottlenecks, but spreading an entangled pair of qubits across distance $n^{0.5+o(1)}$ takes time $n^{0.5+o(1)}$ in the same architectures.

We have not attempted to analyze the impact of these effects for concrete sizes of n. We have also not analyzed communication costs at higher levels of the CSIDH computation. For comparison, attacks against AES [28] use fewer qubits, and perform much longer stretches of computation on nearby qubits.

B Basic integer arithmetic

We use b bits $n_0, n_1, n_2, \ldots, n_{b-1}$ to represent the nonnegative integer $n_0 + 2n_1 + 4n_2 + \cdots + 2^{b-1}n_{b-1}$. Each element of $\{0, 1, \ldots, 2^b - 1\}$ has a unique representation as b bits. This appendix analyzes the cost of additions, subtractions, multiplications, and squarings in this representation.

B.1. Addition. We use a standard sequential **ripple-carry adder**. If $b \geq 1$ then the sum of the b-bit integers represented by $n_0, n_1, n_2, \ldots, n_{b-1}$ and $m_0, m_1, m_2, \ldots, m_{b-1}$ is the $(b+1)$-bit integer represented by $s_0, s_1, s_2, \ldots, s_b$ computed as follows:

$$\begin{array}{lll}
x_0 = n_0 \oplus m_0; & s_0 = x_0; & c_0 = n_0 m_0; \\
x_1 = n_1 \oplus m_1; & s_1 = x_1 \oplus c_0; & c_1 = n_1 m_1 \oplus x_1 c_0; \\
x_2 = n_2 \oplus m_2; & s_2 = x_2 \oplus c_1; & c_2 = n_2 m_2 \oplus x_2 c_1; \\
\vdots & & \\
x_{b-1} = n_{b-1} \oplus m_{b-1}; & s_{b-1} = x_{b-1} \oplus c_{b-2}; & c_{b-1} = n_{b-1} m_{b-1} \oplus x_{b-1} c_{b-2}; \\
 & s_b = c_{b-1}. &
\end{array}$$

There are $5b - 3$ bit operations here, including $2b - 1$ nonlinear bit operations. Our primary cost metric is the number of nonlinear bit operations.

More generally, to add a b-bit integer to an a-bit integer with $a \leq b$, we use the formulas above to obtain a $(b+1)$-bit sum, skipping computations that refer to $m_a, m_{a+1}, \ldots, m_{b-1}$.

Minor speedups: If $a = 0$ then we instead produce a b-bit sum. More generally, we could (but currently do not) track ranges of integers more precisely, and decide based on the output range whether a sum needs b bits or $b+1$ bits. This is compatible with constant-time computation: the sequence of bit operations being carried out is independent of the values of the bits being processed.

B.2. Subtraction. We use a standard ripple-borrow subtractor to subtract two b-bit integers modulo 2^b, obtaining a b-bit integer. The formulas are similar to the ripple-carry adder. The total number of operations grows from 5 to 7 for each bit but the number of nonlinear operations is still 2 per bit.

B.3. Multiplication. Write $Q(b)$ for the minimum number of nonlinear bit operations for b-bit integer multiplication. We combine Karatsuba multiplication [40] and schoolbook multiplication, as explained below, to obtain concrete upper bounds on $Q(b)$ for various values of b. See Table B.1.

We are not aware of previous analyses of $Q(b)$. It is easy to find literature stating the number of bit operations for schoolbook multiplication, but we do better starting at 14 bits. For $b = 512$ we obtain $Q(512) \leq 241908$ (using an algorithm with a total of 536184 bit operations), while schoolbook multiplication uses 784896 nonlinear bit operations (and a total of 1568768 bit operations).

It is also easy to find literature on the number of bit operations for polynomial multiplication mod 2, but carries make the integer case much more expensive and qualitatively change the analysis. For example, [41] uses Karatsuba's method for polynomials all the way down to single-bit multiplication, exploiting the fact that polynomial addition costs 0 nonlinear bit operations; for integer multiplication, Karatsuba's method has much more overhead. Concretely, Karatsuba's method uses just $3^9 = 19683$ nonlinear bit operations to multiply 512-bit polynomials; we use 12 times as many nonlinear bit operations to multiply 512-bit integers.

Schoolbook multiplication. Schoolbook multiplication of two b-bit integers has two stages. The first stage is b^2 parallel multiplications of individual bits. This produces 1 product at position 0; 2 products at position 1; 3 products at position 2; $\ldots$; b products at position $b-1$; $b-1$ products at position b; $\ldots$; 1 product at position $2b-2$. The second stage repeatedly

- adds two bits at position i, obtaining one bit at position i and a carry bit at position $i+1$, or, more efficiently,
- adds three bits at position i, obtaining one bit at position i and a carry bit at position $i+1$,

until there is only one bit at each position.

There are several standard ways to organize the second stage for parallel computation: for example, Wallace trees [73] and Dadda trees [21]. Dadda trees use fewer bit operations since they make sure to add three bits whenever possible rather than two bits. Since parallelism is not visible in our primary cost metric,

1	1	65	8313	129	25912	193	50221	257	79732	321	114068	385	153686	449	197391
2	6	66	8497	130	26224	194	50631	258	80237	322	114669	386	154350	450	198131
3	18	67	8813	131	26733	195	51310	259	81067	323	115655	387	155448	451	199341
4	36	68	8940	132	26925	196	51563	260	81387	324	116035	388	155866	452	199802
5	60	69	9201	133	27377	197	52161	261	82097	325	116877	389	156807	453	200845
6	90	70	9397	134	27701	198	52594	262	82614	326	117490	390	157494	454	201601
7	126	71	9664	135	28098	199	53124	263	83267	327	118263	391	158354	455	202540
8	168	72	9736	136	28233	200	53296	264	83467	328	118499	392	158615	456	202834
9	216	73	10070	137	28699	201	53930	265	84252	329	119428	393	159655	457	203956
10	270	74	10272	138	28968	202	54295	266	84712	330	119972	394	160261	458	204608
11	330	75	10618	139	29440	203	54924	267	85442	331	120834	395	161233	459	205675
12	396	76	10757	140	29644	204	55200	268	85774	332	121226	396	161674	460	206152
13	468	77	11042	141	29992	205	55657	269	86315	333	121863	397	162385	461	206932
14	535	78	11256	142	30267	206	56017	270	86720	334	122340	398	162923	462	207529
15	630	79	11547	143	30682	207	56565	271	87330	335	123058	399	163738	463	208429
16	684	80	11625	144	30762	208	56669	272	87473	336	123225	400	163918	464	208619
17	795	81	11989	145	31307	209	57384	273	88217	337	124101	401	164926	465	209751
18	851	82	12209	146	31649	210	57835	274	88691	338	124659	402	165568	466	210468
19	974	83	12585	147	32206	211	58537	275	89441	339	125541	403	166571	467	211559
20	1036	84	12736	148	32416	212	58808	276	89718	340	125866	404	166944	468	211981
21	1171	85	13045	149	32910	213	59434	277	90403	341	126671	405	167858	469	212960
22	1239	86	13277	150	33264	214	59872	278	90883	342	127235	406	168495	470	213636
23	1386	87	13592	151	33697	215	60402	279	91444	343	127892	407	169237	471	214440
24	1460	88	13676	152	33844	216	60597	280	91656	344	128140	408	169521	472	214750
25	1608	89	14070	153	34352	217	61190	281	92288	345	128880	409	170347	473	215625
26	1688	90	14308	154	34645	218	61532	282	92644	346	129296	410	170812	474	216126
27	1859	91	14703	155	35159	219	62153	283	93343	347	130115	411	171729	475	217072
28	1934	92	14866	156	35381	220	62411	284	93626	348	130446	412	172097	476	217453
29	2092	93	15188	157	35759	221	62873	285	94130	349	131034	413	172758	477	218153
30	2195	94	15427	158	36058	222	63243	286	94553	350	131529	414	173314	478	218725
31	2369	95	15755	159	36509	223	63788	287	95187	351	132271	415	174142	479	219559
32	2431	96	15845	160	36595	224	63887	288	95275	352	132371	416	174254	480	219694
33	2607	97	16247	161	37188	225	64619	289	96171	353	133423	417	175429	481	220845
34	2726	98	16492	162	37560	226	65072	290	96724	354	134072	418	176152	482	221551
35	2914	99	16917	163	38165	227	65820	291	97632	355	135125	419	177314	483	222704
36	2978	100	17081	164	38393	228	66106	292	97982	356	135535	420	177773	484	223156
37	3172	101	17438	165	38929	229	66750	293	98758	357	136432	421	178755	485	224126
38	3303	102	17706	166	39313	230	67219	294	99323	358	137082	422	179465	486	224834
39	3509	103	18058	167	39782	231	67808	295	100036	359	137904	423	180371	487	225741
40	3579	104	18154	168	39941	232	67990	296	100254	360	138158	424	180650	488	226010
41	3791	105	18597	169	40491	233	68699	297	101111	361	139137	425	181723	489	227102
42	3934	106	18860	170	40808	234	69113	298	101613	362	139712	426	182357	490	227747
43	4158	107	19290	171	41364	235	69781	299	102409	363	140618	427	183334	491	228727
44	4234	108	19477	172	41604	236	70083	300	102771	364	141029	428	183780	492	229181
45	4464	109	19811	173	42012	237	70576	301	103360	365	141703	429	184514	493	229913
46	4619	110	20061	174	42335	238	70949	302	103801	366	142205	430	185052	494	230446
47	4850	111	20423	175	42822	239	71513	303	104465	367	142955	431	185849	495	231243
48	4932	112	20514	176	42914	240	71640	304	104620	368	143134	432	186052	496	231449
49	5169	113	20959	177	43555	241	72338	305	105430	369	144043	433	186996	497	232382
50	5325	114	21237	178	43957	242	72782	306	105946	370	144621	434	187597	498	232980
51	5585	115	21698	179	44599	243	73482	307	106762	371	145536	435	188569	499	233970
52	5673	116	21872	180	44845	244	73743	308	107063	372	145874	436	188919	500	234312
53	5928	117	22278	181	45412	245	74380	309	107808	373	146706	437	189807	501	235231
54	6107	118	22572	182	45815	246	74826	310	108330	374	147290	438	190436	502	235886
55	6349	119	22937	183	46309	247	75351	311	108939	375	147973	439	191165	503	236628
56	6432	120	23056	184	46480	248	75549	312	109169	376	148228	440	191431	504	236899
57	6702	121	23492	185	47050	249	76139	313	109855	377	149000	441	192272	505	237769
58	6868	122	23745	186	47380	250	76473	314	110241	378	149435	442	192742	506	238247
59	7154	123	24183	187	47956	251	77120	315	111000	379	150281	443	193666	507	239231
60	7265	124	24373	188	48203	252	77383	316	111307	380	150625	444	194044	508	239630
61	7510	125	24699	189	48630	253	77853	317	111853	381	151233	445	194697	509	240309
62	7692	126	24954	190	48966	254	78244	318	112312	382	151742	446	195250	510	240901
63	7939	127	25337	191	49467	255	78828	319	113000	383	152505	447	196090	511	241814
64	8009	128	25415	192	49565	256	78914	320	113094	384	152611	448	196197	512	241908

Table B.1. Upper bounds on $Q(b)$ for $b \le 512$: e.g., $Q(3) \le 18$. $Q(b)$ is the minimum number of nonlinear bit operations for b-bit integer multiplication.

we simply add sequentially from the bottom bit. Overall we use $6b^2 - 8b$ bit operations for b-bit schoolbook multiplication (if $b \geq 2$), including $3b^2 - 3b$ nonlinear bit operations.

Karatsuba multiplication. When b is not very small, we do better using Karatsuba's method: the product of $X_0 + 2^b X_1$ and $Y_0 + 2^b Y_1$ is $Z_0 + 2^b Z_1 + 2^{2b} Z_2$ where $Z_0 = X_0 Y_0$, $Z_2 = X_1 Y_1$, and $Z_1 = (X_0 + X_1)(Y_0 + Y_1) - (Z_0 + Z_2)$.

Karatsuba's method reduces a $2b$-bit multiplication to two b-bit multiplications for Z_0 and Z_2, two b-bit additions for $X_0 + X_1$ and $Y_0 + Y_1$, one $(b+1)$-bit multiplication, one $2b$-bit addition for $Z_0 + Z_2$, one subtraction modulo 2^{2b+1} for Z_1, and a $4b$-bit addition for $(Z_0 + 2^{2b} Z_2) + 2^b Z_1$. Some operations in the $4b$-bit addition can be eliminated, and counting carefully shows that

$$\begin{aligned} Q(2b-1) &\leq Q(b-1) + Q(b) + Q(b+1) + 17b - 12, \\ Q(2b) &\leq 2Q(b) + Q(b+1) + 17b - 4. \end{aligned}$$

These formulas do better than schoolbook multiplication for $Q(14)$ and for $Q(16), Q(17), Q(18), \ldots$.

For comparison, similar formulas apply to $M(b)$, the total number of bit operations (linear and nonlinear) for b-bit polynomial multiplication mod 2. The cost of schoolbook multiplication then scales as $2b^2$ rather than $3b^2$. The overhead of "refined Karatsuba" multiplication scales as $7b$ rather than $17b$, already giving improved bounds on $M(6)$, and giving, e.g., $M(512) \leq 109048$.

Other techniques. We have skipped some small speedups. For example, the top bit of $(X_0 + X_1)(Y_0 + Y_1)$ does not need to be computed. As another example, one can use "refined Karatsuba" multiplication for integers; see [35] for one way to organize the carry chains. Presumably we have missed some other small speedups.

Toom multiplication [71] implies $Q(b) \in b^{1+o(1)}$. FFT-based improvements in the $o(1)$ appear in, e.g., [60], [56, page 532], [66], [26], [32], and [31]. Our Karatsuba-based bounds on $Q(b)$ can thus be improved for sufficiently large values of b, and perhaps for values of b relevant to CSIDH. For comparison, Bernstein [6] obtained $M(512) \leq 98018$ using Toom multiplication, not a large improvement upon the $M(512) \leq 109048$ bound mentioned above from refined Karatsuba multiplication.

B.4. Squaring. Schoolbook squaring saves about half the work of schoolbook multiplication. Specifically, for each pair (i, j) with $i < j$, schoolbook multiplication adds both $n_i m_j$ and $n_j m_i$ to position $i + j$, while schoolbook squaring adds $n_i n_j$ to position $i + j + 1$; also, schoolbook multiplication adds $n_i m_i$ to position $2i$, while schoolbook squaring adds n_i (which is the same as n_i^2) to position $2i$. Overall we use $3b^2 - 6b + 3$ bit operations for b-bit schoolbook squaring, including $1.5b^2 - 2.5b + 1$ nonlinear bit operations.

Karatsuba squaring also has less overhead than Karatsuba multiplication, but the overhead/schoolbook ratio is somewhat larger for squaring than for multiplication, making Karatsuba squaring somewhat less effective. We obtain squaring speedups from Karatsuba squaring—in our primary cost metric, nonlinear bit operations—starting at 22 bits. For 512 bits we use 143587 nonlinear bit operations, about 60% of the nonlinear bit operations that we use for multiplication.

B.5. Multiplication by a constant. We save even more in the multiplications that arise in reduction modulo p (see Appendix C.1), namely multiplications by large constants. The exact savings depend on the constant; for example, for seven different 512-bit constants, we use

$$107338, 110088, 109574, 111760, 107925, 107711, 108234$$

nonlinear bit operations, about 45% of the multiplication cost. Here the schoolbook method is as follows: if m_j is the constant 1 then add n_i to position $i + j$. We use Karatsuba multiplication starting at 30 bits.

Other techniques. There is some literature studying addition chains (and addition-subtraction chains) with free doublings. For example, [24] shows that multiplication by a b-bit constant uses $O(b/\log b)$ additions (and [48] shows that most constants require $\Theta(b/\log b)$ additions), for a total of $O(b^2/\log b)$ bit operations. This is asymptotically beaten by Karatsuba multiplication, but could be useful as an intermediate step between schoolbook multiplication and Karatsuba multiplication.

C Modular arithmetic

CSIDH uses elliptic curves defined over $\mathbb{F}_p$, where p is a standard prime number. For example, in CSIDH-512, p is the prime number $4 \cdot 3 \cdot 5 \cdot 7 \cdot 11 \cdots 373 \cdot 587 - 1$, between 2^{510} and 2^{511}; all primes between 3 and 373 appear in the product.

Almost all of the bit operations in our computation are consumed by a long series of multiplications modulo p, organized into various higher-level operations such as exponentiation and elliptic-curve scalar multiplication. This appendix analyzes the performance of modular multiplication, exponentiation, and inversion.

C.1. Reduction. We completely reduce a nonnegative integer z modulo p as follows. Assume that z has c bits (so $0 \le z < 2^c$), and assume $2^{b-1} < p < 2^b$ with $b \ge 2$.

If $c < b$ then there is nothing to do: $0 \le z < 2^{b-1} < p$. Assume from now on that $c \ge b$.

Compute an approximation q to z/p precise enough to guarantee that $0 \le z - qp < 2p$. Here we use the standard idea of multiplying by a precomputed reciprocal:

- Precompute $R = \lfloor 2^{c+2}/p \rfloor$. Formally, this costs 0 in our primary cost metric, since precomputation is part of *constructing* our algorithm rather than *running* our algorithm. More importantly, our entire algorithm uses only a few small values of c, so this precomputation has negligible cost.

- Compute $q = \lfloor \lfloor z/2^{b-2} \rfloor R/2^{c-b+4} \rfloor$. The cost of computing q is the cost of multiplying the $(c-b+2)$-bit integer $\lfloor z/2^{b-2} \rfloor$ by the constant $(c-b+3)$-bit integer R. Computing $\lfloor z/2^{b-2} \rfloor$ means simply taking the top $c-b+2$ bits of z.

By construction $R \le 2^{c+2}/p$ and $q \le zR/2^{c+2}$ so $q \le z/p$. Checking that $z/p < q+2$ involves more inequalities:

- $2^{c+2}/p < R+1$ so $z/p < z(R+1)/2^{c+2} < zR/2^{c+2} + 1/4$. This uses the fact that $0 \le z < 2^c$.
- $z/2^{b-2} < \lfloor z/2^{b-2} \rfloor + 1$, so $(z/2^{b-2})R/2^{c-b+4} < \lfloor z/2^{b-2} \rfloor R/2^{c-b+4} + 1/2$. This uses the fact that $0 \le R < 2^{c-b+3}$.
- $\lfloor z/2^{b-2} \rfloor R/2^{c-b+4} < q+1$.
- Hence $z/p < q + 1 + 1/2 + 1/4 = q + 7/4$.

Next replace z with $z - qp$. This involves a multiplication of the $(c-b+1)$-bit integer q by the constant b-bit integer p, and a subtraction. We save some time here by computing only the bottom $b+1$ bits of qp and $z - qp$, using the fact that $0 \le z - qp < 2^{b+1}$.

At this point (the new) z is between 0 and $2p-1$, so all that remains is to subtract p from z if $z \ge p$.

Compute $y = z - p \bmod 2^{b+1}$. Use y_b, the bit at position b of y, to select between the bottom b bits of z and the bottom b bits of y: specifically, compute $y_0 \oplus y_b(y_0 \oplus z_0)$, $y_1 \oplus y_b(y_1 \oplus z_1)$, and so on through $y_{b-1} \oplus y_b(y_{b-1} \oplus z_{b-1})$. If $z \ge p$ then $0 \le z - p < p < 2^b$ so $y = z - p$ and $y_p = 0$, so these output bits are $y_0, y_1, \ldots, y_{b-1}$ as desired; if $z < p$ then $-2^b < -p \le z - p < 0$ so $y = z - p + 2^{b+1}$ and $y_p = 1$, so these output bits are $z_0, z_1, \ldots, z_{b-1}$ as desired.

Other techniques. We could save time in the multiplication by R by skipping most of the computations involved in bottom bits of the product. It is important for the total of the bits thrown away to be at most 2^{c-b+2}, so that q is reduced by at most $1/4$, the gap between $q+2$ and the $q+7/4$ mentioned above.

We could vary the number of bits in R, the allowed range of $z - qp$, etc. The literature sometimes recommends repeatedly subtracting p once z is known to be small, but if the range is (e.g.) 0 through $4p-1$ then it is slightly better to first subtract $2p$ and then subtract p.

Historical notes. Multiplying by a precomputed reciprocal, to compute a quotient and then a remainder, is often called “Barrett reduction”, in reference to a 1986 paper [2]. However, Knuth [44, page 264] had already commented in 1981 that Newton's method “for evaluating the reciprocal of a number was extensively used in early computers” and that, for “extremely large numbers”, Newton's method and “subsequent multiplication” using fast multiplication techniques can be “considerably faster” than a simple quadratic-time division method.

C.2. Multiplication. To multiply b-bit integers x, y modulo p, we follow the conventional approach of first multiplying x by y, and then reducing the $2b$-bit product xy modulo p as explained in Appendix C.1.

For example, for CSIDH-512, we use 241814 nonlinear bit operations for 511-bit multiplication, and 206088 nonlinear bit operations for reduction modulo p, for a total of 447902 nonlinear bit operations for multiplication modulo p.

Generic conversion to a quantum algorithm (see Appendix A.4) produces $14 \cdot 447902 = 6270628$ T-gates. This T-gate count is approximately 48 times larger than the cost "2^{17}" claimed in [11, Table 6]. The ratio is actually closer to 100, since [11] claims to count "Clifford+T" gates while we count only T-gates. We do not claim that the generic conversion is optimal, but there is no justification for [11] using an estimate for the costs of multiplication in a binary field as an estimate for the costs of multiplication in $\mathbb{F}_p$.

Squaring. For CSIDH-512, we use 143508 nonlinear bit operations for 511-bit squaring, and again 206088 nonlinear bit operations for reduction modulo p, for a total of 349596 nonlinear bit operations for squaring modulo p. This is about 78% of the cost of a general multiplication, close to the traditional 80% estimate.

Other techniques. Montgomery multiplication [52] computes $xy/2^b$ modulo p, using a multiple of p to clear the bottom bits of xy. This has the same asymptotic performance as clearing the top bits; it sometimes requires extra multiplications and divisions by 2^b modulo p but might be slightly faster overall.

C.3. Addition.

A standard speedup for many software platforms is to avoid reductions after additions. For example, to compute $(x + y)z$ modulo a 511-bit p, one computes the 512-bit sum $x + y$, computes the 1023-bit product $(x + y)z$, and then reduces modulo p.

However, bit operations are not the same as CPU cycles. An intermediate reduction of $x + y$ modulo p (using the last step of the reduction procedure explained in Appendix C.1, a conditional subtraction of p) involves relatively few bit operations, and saves more bit operations because the multiplication and reduction are working with slightly smaller inputs.

C.4. Exponentiation with small variable exponents.

Our isogeny algorithms involve various computations $x^e \bmod p$ where e is a variable having only a few bits, typically under 10 bits.

To compute $x^e \bmod p$ where $e = e_0 + 2e_1 + 4e_2 + \cdots + 2^{b-1}e_{b-1}$, we start with $x^{e_{b-1}}$, square modulo p, multiply by $x^{e_{b-2}}$, square modulo p, and so on through multiplying by x^{e_0}. We compute each x^{e_i} by using the bit e_i to select between 1 and x; this takes a few bit operations per bit of x, as in Appendix C.1.

Starting at $b = 4$, we instead use "width-2 windows". This means that we perform a sequence of square-square-multiply operations, using two bits of e at a time to select from a precomputed table of $1, x, x^2 \bmod p, x^3 \bmod p$. For example, for 10-bit exponents, we use 9 squarings and 5 general multiplications.

None of the CSIDH parameters that we tested involved variable exponents e large enough to justify window width 3 or larger.

C.5. Inversion.

We compute the inverse of x in $\mathbb{F}_p$ as $x^{p-2} \bmod p$. This is different from the situation in Appendix C.4, in part because the exponent here is a constant and in part because the exponent here has many more bits.

We use fractional sliding windows to compute $x^{p-2} \bmod p$. This means that we begin by computing $x^2, x^3, x^5, x^7, x^9, \ldots, x^W$ modulo p, where W is a parameter; "fractional" means that $W+1$ is not required to be a power of 2. We then recursively compute x^e as $(x^{e/2})^2$ if e is even, and as x^r times x^{e-r} if e is odd, where $r \in \{1, 3, 5, 7, 9, \ldots, W\}$ is chosen to maximize the number of 0 bits at the bottom of $e - r$. For small e we use some minor optimizations listed in [8, Section 3]: for example, we compute x^e as $x^{e/2-1}x^{e/2+1}$ if e is a multiple of 4 and $e \leq 2W - 2$.

We choose W as follows. Given a b-bit target exponent e, we automatically evaluate the cost of the computation described above for each odd $W \leq 2b + 3$. For this evaluation we model the cost of a squaring as 0.8 times the cost of a general multiplication, without regard to p. We could instead substitute the exact costs for arithmetic modulo p.

For CSIDH-512, we use 537503414 bit operations for inversion, including 220691666 nonlinear bit operations. Here W is chosen as 33. There are 507 squarings, accounting for $507 \cdot 349596 = 177245172$ nonlinear bit operations, and 97 general multiplications, accounting for the remaining $97 \cdot 447902 = 43446494$ nonlinear bit operations.

Batching inversions. We use Montgomery's trick [53] of computing $1/y$ and $1/z$ by first computing $1/yz$ and then multiplying by z and y respectively. This reduces a batch of two inversions to one inversion and three multiplications; a batch of three inversions to one inversion and six multiplications; etc.

Inversion by exponentiation allows input 0 and produces output 0. This extension of the inversion semantics is often convenient for higher-level computations: for example, some of our computations sometimes generate input 0 in settings where the output will later be thrown away. However, Montgomery's trick does not preserve these semantics: for example, if $y = 0$ and $z \neq 0$ then Montgomery's trick will produce 0 for both outputs.

We therefore tweak Montgomery's trick by replacing each input 0 with input 1 (and replacing the corresponding output with 0; we have not checked whether any of our computations need this). To do this with a constant sequence of bit operations, we compare the input to 0 by ORing all the bits together, and we then XOR the complement of the result into the bottom bit of the input.

Eliminating inversions. Sometimes, instead of dividing x by z, we maintain x/z as a fraction. This skips the inversion of z, but usually costs some extra multiplications. We quantify the effects of this choice in describing various higher-level computations: for example, this is the choice between "affine" and "projective" coordinates for elliptic-curve points in Section 3.2.

The Legendre symbol. The Legendre symbol of x modulo p is, by definition, 1 if x is a nonzero square modulo p; -1 if x is a non-square modulo p; and 0 if x is divisible by p. The Legendre symbol is congruent modulo p to $x^{(p-1)/2}$, and we compute it this way.

The cost of the Legendre symbol is marginally smaller than the cost of inversion. For example, for CSIDH-512, there are 506 squarings and 96 general multiplications, in total using 535577602 bit operations, including 218988158 nonlinear bit operations.

Other techniques. It is well known that inversion in $\mathbb{F}_p$ via an extended version of Euclid's algorithm is asymptotically much faster than inversion via exponentiation. Similar comments apply to Legendre-symbol computation.

However, Euclid's algorithm is a variable-time loop, where each iteration consists of a variable-time division. This becomes very slow when it is converted in a straightforward way into a constant-time sequence of bit operations. Faster constant-time variants of Euclid's algorithm are relatively complicated and still have considerable overhead; see, e.g., [12] and [63, Section 3.4].

We encourage further research into these constant-time algorithms. Sufficiently fast inversion and Jacobi-symbol computation could save more than 10% of our overall computation time.

C.6. Fewer qubits. In this subsection we look beyond our primary cost metric and consider some of the other costs incurred by integer arithmetic.

Consider, e.g., the sequence of bit operations described in Appendix C.5 for inversion in the CSIDH-512 prime field: 537503414 bit operations, including 220691666 nonlinear bit operations. A generic conversion (see Appendix A.3) produces a reversible computation using $2 \cdot 220691666 = 441383332$ Toffoli gates.

It is important to realize that this reversible computation also uses 537503414 bits of intermediate storage, and the corresponding quantum computation (see Appendix A.4) uses 537503414 qubits. The factor 2 mentioned in the previous paragraph accounts for the cost of "uncomputation" to recompute these intermediate results in reverse order; all of the results are stored in the meantime. Presumably many of the linear operations can be carried out in place, reducing the intermediate storage, but this improvement is limited: about 40% of the bit operations that we use are nonlinear. The number of qubits is even larger for higher-level computations, such as our algorithms for the CSIDH group action.

In traditional non-reversible computation, the bits used to store intermediate results in one multiplication can be erased and reused to store intermediate results for the next multiplication. Something similar is possible for reversible computation (and quantum computation), but one does not simply erase the intermediate results; instead one immediately uncomputes each multiplication, doubling the cost of each multiplication. The inversion operation uses many of these double-cost multiplications and accumulates its own sequence of intermediate results, which also need to be uncomputed, again using the double-cost multiplications. To summarize, this reuse of bits doubles the number of Toffoli gates used for inversion from 441383332 to 882766664. Similar comments apply to qubits and T-gates.

The intermediate space used for multiplication outputs in inversion, in scalar multiplication, etc. can similarly be reused, but this produces another doubling of costs. Even after these two doublings, our higher-level computations still require something on the scale of a million qubits.

Quantum algorithms are normally designed to fit into far fewer qubits, even when this means sacrificing many more qubit operations. For example—in the context of applying Shor's attack to an elliptic curve defined over a prime field—Roetteler, Naehrig, Svore, and Lauter [63, Table 1] squeeze b-bit reversible modular multiplication into

- $5b + 4$ bits using approximately $(16 \log_2 b - 26.3)b^2$ Toffoli gates, or
- $3b + 2$ bits using approximately $(32 \log_2 b - 59.4)b^2$ Toffoli gates.

These are about $2^{24.87}$ or $2^{25.83}$ Toffoli gates for $b = 511$, far more than the number of Toffoli gates we use.

We focus on the challenge of minimizing the number of nonlinear bit operations for the CSIDH class-group action. Understanding the entire tradeoff curve between operations and qubits—never mind more advanced issues such as parallelism (Appendix A.5) and communication costs (Appendix A.6)—goes far beyond the scope of this paper. See [57] for some recent work on improving these tradeoffs for reversible Karatsuba multiplication; see also [16], which fits Karatsuba multiplication into fewer bits but does not analyze reversibility.