CTIDH: faster constant-time CSIDH

Gustavo Banegas[1], Daniel J. Bernstein[2,3], Fabio Campos[4], Tung Chou[5], Tanja Lange[6], Michael Meyer[7], Benjamin Smith[1] and Jana Sotáková[8,9]

[1] Inria and Laboratoire d'Informatique de l'Ecole polytechnique, Institut Polytechnique de Paris, Palaiseau, France
gustavo@cryptme.in
smith@lix.polytechnique.fr
[2] Department of Computer Science, University of Illinois at Chicago, USA
[3] Horst Görtz Institute for IT Security, Ruhr University Bochum, Germany
djb@cr.yp.to
[4] Max Planck Institute for Security and Privacy, Bochum, Germany
campos@sopmac.de
[5] Academia Sinica, Taipei, Taiwan
blueprint@crypto.tw
[6] Eindhoven University of Technology, Eindhoven, The Netherlands
tanja@hyperelliptic.org
[7] Technical University of Darmstadt, Darmstadt, Germany
michael@random-oracles.org
[8] Institute for Logic, Language and Computation, University of Amsterdam, The Netherlands
[9] QuSoft
j.s.sotakova@uva.nl

Abstract. This paper introduces a new key space for CSIDH and a new algorithm for constant-time evaluation of the CSIDH group action. The key space is not useful with previous algorithms, and the algorithm is not useful with previous key spaces, but combining the new key space with the new algorithm produces speed records for constant-time CSIDH. For example, for CSIDH-512 with a 256-bit key space, the best previous constant-time results used 789000 multiplications and more than 200 million Skylake cycles; this paper uses 438006 multiplications and 125.53 million cycles.

Keywords: post-quantum cryptography · non-interactive key exchange · small keys · isogeny-based cryptography · CSIDH · constant-time algorithms

Author list in alphabetical order; see https://www.ams.org/profession/leaders/culture/CultureStatement04.pdf. This work began at the online Lorentz Center workshop "Post-Quantum Cryptography for Embedded Systems". This work was carried out while the second and fifth authors were visiting Academia Sinica. This work was funded in part by the European Commission through H2020 SPARTA, https://www.sparta.eu/, by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy—EXC 2092 CASA—390781972 "Cyber Security in the Age of Large-Scale Adversaries", by the Cisco University Research Program, by the U.S. National Science Foundation under grant 2037867, by the Taiwan's Executive Yuan Data Safety and Talent Cultivation Project (AS-KPQ-109-DSTCP), by Continental AG and Elektrobit Automotive GmbH, by Taiwan Ministry of Science and Technology (MoST) grant MOST105-2221-E-001-014-MY3, 108-2221-E-001-008 and 109-2222-E-001-001-MY3, by Academia Sinica Investigator Award AS-IA-104-M0, by the Netherlands Organisation for Scientific Research (NWO) under grant 628.001.028 (FASOR), by the German Federal Ministry of Education and Research and the Hessian Ministry of Higher Education, Research, Science and the Arts within their joint support of the National Research Center for Applied Cybersecurity ATHENE, by the French Agence Nationale de la Recherche through ANR CIAO (ANR-19-CE48-0008), and by the Dutch Research Council (NWO) through Gravitation-grant Quantum Software Consortium - 024.003.037. "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: 3f5cc78da1af948120b3e69222369fb08aedadf6. Date: 2021.05.26.

1 Introduction

Isogeny-based cryptography, a relatively new area of post-quantum cryptography, has gained substantial attention in the past few years. Schemes like SIDH (Supersingular Isogeny Diffie–Hellman) [14] and CSIDH (Commutative Supersingular Isogeny Diffie–Hellman) [10] offer key-exchange protocols with the smallest key sizes among post-quantum systems. CSIDH is even a non-interactive key exchange, matching the data flow of traditional Diffie–Hellman key exchange, and it has received a considerable amount of attention related to constant-time algorithms [16, 20, 11, 15, 13, 1, 12].

Briefly, one can explain CSIDH as follows. Pick small odd primes $\ell_1 < \ell_2 < \cdots < \ell_n$ such that $p = 4 \cdot \ell_1 \cdots \ell_n - 1$ is also prime. A public key is a supersingular elliptic curve $E_A/\mathbb{F}_p : y^2 = x^3 + Ax^2 + x$, specified by a single element $A \in \mathbb{F}_p$. Given this curve one can efficiently compute two curves ℓ_i-isogenous to E_A, denoted $\mathfrak{l}_i \star E_A$ and $\mathfrak{l}_i^{-1} \star E_A$, for any of the ℓ_i in the definition of p. Alice's private key is a list of exponents $(e_1, \ldots, e_n) \in \mathbb{Z}^n$ where e_i shows how often each $\mathfrak{l}_i$ is used: Alice's public key is $\mathfrak{l}_1^{e_1} \cdots \mathfrak{l}_n^{e_n} \star E_0 = E_A$, and if Bob's public key is E_B then the secret shared with Bob is $\mathfrak{l}_1^{e_1} \cdots \mathfrak{l}_n^{e_n} \star E_B$. The key-exchange protocol works because $\star$ is a commutative group action: the ordering of the isogenies is not important.

The first constant-time CSIDH paper [16] specified each exponent e_i as being between 0 and a public constant m_i, and always computed m_i iterations of $\mathfrak{l}_i$, secretly discarding the dummy operations beyond e_i iterations. The original CSIDH paper [10] had allowed $e_i \in [-m_i, m_i]$; in the constant-time context this might seem to require m_i iterations of $\mathfrak{l}_i$ plus m_i iterations of $\mathfrak{l}_i^{-1}$, but [20] introduced a "2-point" algorithm with just m_i iterations, each iteration being only about 1/3 more expensive than before. All m_i were taken equal in [10], for example taking $e_i \in [-5, 5]$ for CSIDH-512; subsequent papers did better by allowing m_i to depend on i (as suggested in [10, Remark 14]) and accounting for the costs of $\mathfrak{l}_i$. Further speedups in the literature come from various techniques to speed up each $\mathfrak{l}_i$ computation and to merge work across sequences of $\mathfrak{l}_i$ computations.

1.1 Contributions of this paper

This paper introduces a new key space for CSIDH, and a new constant-time algorithm to evaluate the CSIDH group action. The new key space is not useful by itself—it slows down previous constant-time algorithms—and similarly the new constant-time algorithm is not useful for previous key spaces; but there is a synergy between the key space and the algorithm, and using both of them together produces a large improvement in the performance of constant-time CSIDH.

As a very small example of the new key space, assume that one is using just 6 primes and allows at most 6 isogeny computations, with each $\mathfrak{l}_i$ exponent being nonnegative. The standard key space chooses $(e_1, e_2, \ldots, e_6) \in \{0, 1\}^6$, giving $2^6 = 64$ keys. The new key space, with 2 *batches* of 3 primes each, chooses $(e_1, e_2, \ldots, e_6) \in [0, 3]^6$ with the condition that $e_1 + e_2 + e_3 \le 3$ and $e_4 + e_5 + e_6 \le 3$, giving $20^2 = 400$ keys. Similar comments apply when negative exponents are allowed.

The extreme case of putting each prime into a size-1 batch is not new: it is the standard key space. The opposite extreme, putting all primes into 1 giant batch, is also not new: putting a bound on the 1-norm of the key vector was highlighted in [18] as allowing the best tradeoffs between the number of isogenies and the size of the key space. In the above example, 1 giant batch of 6 primes gives 924 keys for 6 isogenies, i.e., 0.61 isogenies per key bit, compared to 1 isogeny per key bit for the standard key space.

However, plugging 1 giant batch into constant-time algorithms takes 36 isogenies for 924 keys, since each of the 6 primes uses 6 computations. Our intermediate example, 2 batches of 3 primes each, uses 18 isogenies for 400 keys, which is still many more isogenies per key bit than 6 isogenies for 64 keys.

We do better by evaluating isogenies differently. The central challenge tackled in this paper is to develop an efficient constant-time algorithm for the new key space, computing *any* isogeny in a batch using the *same* sequence of operations. This raises several questions:

1. How does one optimally compute a sequence of isogenies, and handle probabilistic failures in standard algorithms to compute $\mathfrak{l}_i$, while at the same time hiding which isogeny is computed? See Section 4 for the introduction of *atomic blocks* for these computations, and Section 5 for how to compute them in constant time.
2. How does one optimally set batches, compute private keys, and determine the number of isogenies per batch to match a required size of the key space? See Section 3 for analysis of the key space, and Section 6 for how to minimize the multiplication count.
3. How does one minimize the cycle count for constant-time software? Section 7 describes our software and low-level ideas; Section 8 presents the speeds.

Our constant-time algorithm combines several old and new techniques. For example, as observed in [5], Vélu's formulas have a Matryoshka-doll structure; we constrain the more recent √élu formulas [3] in a way that creates a Matryoshka-doll structure. The cost of an isogeny computation exploiting this structure depends on the largest prime in the batch for the traditional formulas, but also on the smallest prime in the batch for √élu. Standard algorithms to compute $\mathfrak{l}_i$ fail with probability $1/\ell_i$; to hide which ℓ_i in a batch is used we arrange for failures to occur with probability matching the smallest prime in the batch. Our batches consist of primes of similar sizes, to obtain the optimal tradeoffs between the cost per batch and the size of the key space. Further constant-time optimizations are described throughout the paper.

For comparability we report CSIDH-512 speeds, setting records in multiplications and in cycles for complete constant-time software. Partial analyses [5, 21, 7, 12] suggest that the post-quantum security level of CSIDH-512 is around 2^{60} qubit operations; for applications that want higher security levels, our software also supports larger sizes.

2 Background

Sections 2.1, 2.2, 2.3, and 2.4 review the CSIDH group action, computations of individual ℓ-isogenies, strategies for computing sequences of isogenies, and previous constant-time algorithms.

2.1 The CSIDH group action

CSIDH [10] is a Diffie–Hellman-like key-exchange protocol based on isogenies of supersingular elliptic curves over a finite field $\mathbb{F}_p$. For a prime $p > 3$ an elliptic curve $E/\mathbb{F}_p$ is supersingular if and only if $\#E(\mathbb{F}_p) = p + 1$, where $E(\mathbb{F}_p)$ is its group of points over $\mathbb{F}_p$. CSIDH uses $p \equiv 3 \pmod 8$, and uses supersingular elliptic curves over $\mathbb{F}_p$ that can be written in Montgomery form $E_A : y^2 = x^3 + Ax^2 + x$ for $A \in \mathbb{F}_p \backslash \{-2, 2\}$. We call A the *Montgomery coefficient* of E_A. We write $\mathcal{E} = \{E_A : \#E_A(\mathbb{F}_p) = p + 1\}$ for the set of CSIDH curves and $\mathcal{M} = \{A : E_A \in \mathcal{E}\}$ for the set of corresponding Montgomery coefficients. Curves E_A for distinct $A \in \mathcal{M}$ are non-isomorphic by [10, Proposition 8], and each $E_A(\mathbb{F}_p)$ is cyclic.

An isogeny is a nonzero map $\varphi : E \to E'$ which is given by rational functions and is compatible with elliptic-curve addition. An ℓ-isogeny is an isogeny of degree ℓ (as a rational map). Isogenies are typically defined by their kernels, i.e., by the points they map to ∞. Computing an ℓ-isogeny with Vélu's formulas requires a point P of order ℓ; the isogeny has kernel $\langle P \rangle$, and any point in $\langle P \rangle$ of order ℓ leads to the same isogeny.

The CSIDH prime p is chosen as $4 \cdot \ell_1 \cdots \ell_n - 1$ for small odd primes $\ell_1 < \ell_2 < \cdots < \ell_n$. If E_A is a curve in $\mathcal{E}$ then there are $\ell_j - 1$ points of order ℓ_j in $E_A(\mathbb{F}_p)$. Each of these

points of order ℓ_j generates the kernel of an ℓ_j-isogeny $E_A \to E_{A'}$, which is the same isogeny for all of these points. The codomain $E_{A'}$ of this ℓ_j-isogeny is written $\mathfrak{l}_j \star E_A$.

Fix $i \in \mathbb{F}_{p^2}$ with $i^2 = -1$. Define $\tilde{E}_A(\mathbb{F}_p)$ as the set of points $(x, iy) \in E_A(\mathbb{F}_{p^2})$ where $x, y \in \mathbb{F}_p$, along with the neutral element; equivalently, $\tilde{E}_A(\mathbb{F}_p)$ is the image of $E_{-A}(\mathbb{F}_p)$ under the isomorphism $(x, y) \mapsto (-x, iy)$. For each ℓ_j, there are $\ell_j - 1$ points of order ℓ_j in $\tilde{E}_A(\mathbb{F}_p)$. Each of these points generates the kernel of an ℓ_j-isogeny $E_A \to E_{A''}$, the same isogeny for all of these points. The codomain $E_{A''}$ of this ℓ_j-isogeny is written $\mathfrak{l}_j^{-1} \star E_A$.

The isogeny from E_A to $\mathfrak{l}_j^{-1} \star E_A$ maps the points of order ℓ_j in $E_A(\mathbb{F}_p)$ to points of order ℓ_j, while mapping the points of order ℓ_j in $\tilde{E}_A(\mathbb{F}_p)$ to ∞ on $\mathfrak{l}_j^{-1} \star E_A$. The isogeny from E_A to $\mathfrak{l}_j \star E_A$ maps the points of order ℓ_j in $\tilde{E}_A(\mathbb{F}_p)$ to points of order ℓ_j, while mapping the points of order ℓ_j in $E_A(\mathbb{F}_p)$ to ∞ on $\mathfrak{l}_j \star E_A$.

Applying ℓ_i-isogenies induces a group action [5] of the commutative group $\mathbb{Z}^n$ on $\mathcal{E}$. An exponent vector $(e_1, \ldots, e_n) \in \mathbb{Z}^n$ acts on the curve E_A to produce the curve $(\mathfrak{l}_1^{e_1} \ldots \mathfrak{l}_n^{e_n}) \star E_A$, computed as a sequence having $|e_j|$ many ℓ_j-isogenies for each j, each isogeny using $\mathfrak{l}_j$ if $e_j > 0$ or $\mathfrak{l}_j^{-1}$ if $e_j < 0$.

The private key of each party is a (secret) vector $(e_1, \ldots, e_n)$ sampled from a finite key space $\mathcal{K} \subset \mathbb{Z}^n$. To protect against meet-in-the-middle attacks, it is conventional to take $\#\mathcal{K} \geq 2^{2\lambda}$ for security 2^λ, but see [12] for arguments that smaller key spaces suffice. Beyond the size of $\mathcal{K}$, the specific choice of $\mathcal{K}$ has an important impact on efficiency.

Most previous CSIDH implementations have used one of two types of key space. Given an exponent bound vector $m = (m_1, \ldots, m_n) \in \mathbb{Z}^n_{\geq 0}$, let $\mathcal{K}_m := \prod_{i=1}^n \{-m_i, \ldots, m_i\}$ and $\mathcal{K}_m^+ := \prod_{i=1}^n \{0, \ldots, m_i\}$. Clearly, $\#\mathcal{K}_m = \prod_{i=1}^n (2m_i + 1)$ and $\#\mathcal{K}_m^+ = \prod_{i=1}^n (m_i + 1)$. The original CSIDH paper [10] and [20] use $\mathcal{K}_m$ with $m = (5, \ldots, 5)$ for CSIDH-512. It was suggested in [10, Remark 14] and shown in [13] that allowing the m_i to vary improves speed. The space $\mathcal{K}_m^+$ with $m = (10, \ldots, 10)$ was used in [16] for CSIDH-512.

2.2 Computing isogenies

Let $P \in E_A$ be a point of order ℓ with x-coordinate in $\mathbb{F}_p$ and $\varphi : E_A \to E_{A'} = E_A/\langle P \rangle$ the ℓ-isogeny induced by P. The main computational task, called xISOG, is to compute (1) the Montgomery coefficient A' of the target curve $E_{A'}$ and (2) the images under φ of some specified points (normally zero, one, or two points) $Q \in E_A$ with x-coordinate in $\mathbb{F}_p$.

Vélu and √élu. The main algorithms for xISOG are Vélu's formulas [22] and √élu[1] [3]. The main computational task in both of these algorithms is to evaluate a polynomial

$$h_S(X) = \prod_{s \in S} (X - x([s]P)) \tag{1}$$

for some index set S. All the arithmetic is done using only x-coordinates.

Vélu's formulas evaluate $h_S(X)$ with $S = \{1, 2, 3, \ldots, (\ell - 1)/2\}$ by first computing $x(P), x([2]P), \ldots x([(\ell - 1)/2]P)$ and then evaluating the product (1). This costs $O(\ell)$ field multiplications. Specifically, computing A' costs about 4ℓ field multiplications and computing the image of a point costs about 2ℓ extra field multiplications.

For √élu, in contrast to the linear algorithm of Vélu, the main part of the product is evaluated using a baby-step-giant-step strategy. It is simplest to evaluate $h_S(X)$ with $S = \{1, 3, 5, \ldots, \ell - 2\}$; this set S is split into a "box" $U \times V$ and leftover set W such that $S \leftrightarrow (U \times V) \cup W$. Then $h_S(X)$ is computed as the product of $h_W(X)$ with the resultant of $h_U(X)$ and a polynomial related to $h_V(X)$; see [3] for details. For each ℓ, one chooses U and V to minimize cost. Asymptotically, √élu uses $\tilde{O}(\sqrt{\ell})$ field multiplications.

[1] Pronounced "square-root Vélu".

One can view Vélu's formulas as a special case of √élu in which U and V are empty. This special case is optimal for small primes. The exact cutoff depends on lower-level algorithmic details but is around $\ell = 89$.

Sampling points of order ℓ. No efficient way is known to deterministically generate points of order ℓ in $E(\mathbb{F}_p)$. However, $E(\mathbb{F}_p)$ is cyclic of order $p+1$, so if T is a uniform random point then $P = [(p+1)/\ell]T$ will have order ℓ with probability $1 - 1/\ell$. This can—and often will—fail, and needs to be repeated until it succeeds. Once P has order ℓ, one can use P with Vélu or √élu.

Typically, one uses the Elligator 2 map [4] to sample points in $E(\mathbb{F}_p)$ or $\tilde{E}(\mathbb{F}_p)$. We discuss this approach in Appendix B.

2.3 Strategies

Computing the multiple $[(p+1)/\ell]T$ is very costly. If p has 512 bits then $(p+1)/\ell$ has almost 512 bits and this scalar multiplication costs thousands of field multiplications. The cost of scalar multiplication is typically amortized by pushing points through isogenies. This approach aims to compute a series of isogenies after only sampling one point on the initial curve.

Following [14], we call a method that computes a given series of isogenies a *strategy*. Informally, a strategy determines the order of isogeny evaluations, and how to obtain suitable kernel generators through either scalar multiplications or point evaluations. In the context of SIDH, *optimal strategies* (with the minimum computational cost) can be found [14]. When adapting this to CSIDH, there are three main complications: the choice of isogenies to be combined into a sequence, the possibility of point rejections due to wrong orders, and the pairwise different degrees of the involved isogenies. Indeed, [13] showed that a direct adaption of the method of [14] to CSIDH becomes infeasible when considering all possible permutations. Instead, several avenues for optimizing CSIDH strategies have been proposed, though none claims actual optimality.

Multiplicative strategy. A simple *multiplicative* strategy was used in the algorithm of [10]. Let D, a divisor of $p+1$, be the product of the degrees of the isogenies to be combined in one sequence. Sample a point T on the initial curve, and set $T \leftarrow [(p+1)/D]T$; now the order of T divides D. Compute $P \leftarrow [D/\ell]T$, where ℓ is the degree of the first isogeny. If $P = \infty$, skip this isogeny and continue with the next isogeny. If $P \neq \infty$, then P has order ℓ, so we can compute the required ℓ-isogeny φ, and push T through to get $T \leftarrow \varphi(T)$. Either way, the order of T now divides D/ℓ. For the next isogeny, say of degree ℓ', compute $P \leftarrow [D/(\ell\ell')]T$ as a potential kernel generator. Continue in the same fashion, pushing one point through each isogeny, until $T = \infty$. Note that the scalar multiplications reduce in length at each step: as observed in [17], processing the isogenies in decreasing degree order reduces the total cost.

SIMBA. In [10], the product D was taken as large as possible at each step. The SIMBA (Splitting Isogenies into Multiple Batches) strategy proposed in [16] limits D for better performance. SIMBA-M partitions the set of isogeny degrees $\{\ell_1, \ldots, \ell_n\}$ into M prides,[2] where the i-th pride contains all isogeny degrees ℓ_j for which $j \equiv i \mod M$. Each step, SIMBA picks as many isogenies as possible for a single pride, processing them as described above. This typically results in a smaller total degree D, making the initial scalar multiplication $T \leftarrow [(p+1)/D]T$ more expensive, while the later scalar multiplications

[2] This refers to the term *pride of lions*. The term *batch* was used in [16]; we use *pride* here in order to distinguish between SIMBA batches and the batches of primes considered in CTIDH.

of the form $P \leftarrow [D/\prod \ell_j]T$ become significantly cheaper. Overall, [16] reports that a significant speedup can be achieved for well-chosen values of M.

Point-pushing strategies. One can also push additional points through isogenies. For instance, when computing the first kernel generator as described above via $P \leftarrow [D/\ell]T$, one can save an intermediate point T' of small order divisible by ℓ', push both T and T' through the first isogeny, and then use T' to compute the next kernel generator via a smaller scalar multiplication. This may be more efficient than the multiplicative strategy, depending on the cost of evaluating φ at additional points compared to the savings due to cheaper scalar multiplications. However, except for very small isogeny degrees, the cost of evaluating additional points can be significantly higher than computing scalar multiplications. Thus, an optimal strategy is expected to be closer to the multiplicative strategy, only rarely pushing additional points through isogenies. In [13] optimal strategies are computed under the assumption of always choosing as many isogeny degrees as possible per sequence, and an increasing ordering of the involved degrees. It remains open whether other choices of primes per sequence, as in SIMBA, or different orderings of the degrees could yield a more optimized point-pushing strategy.

Remark 1. The SIMBA approach is generalized in [15] with point-pushing strategies within prides, more efficient partitions of SIMBA prides, and their permutations. However, all optimization attempts have required imposing certain assumptions in order for the optimization problem to be solvable, and thus only produce conditionally optimal strategies. The comparison in [13] shows that all of these approaches give roughly the same performance results for constant-time CSIDH-512 algorithms: that is, within a margin of 4%.

2.4 Previous constant-time algorithms

A private key $(e_1, \ldots, e_n)$ requires us to compute $|e_i|$ isogenies of degree ℓ_i (regardless of the strategy), so the running time of a naïve CSIDH algorithm depends directly on the key. Various constant-time approaches have been proposed to avoid this dependency.

What constant time means. A deterministic algorithm computes a function from inputs to outputs. A randomized algorithm is more complicated: it computes a function from inputs to distributions over outputs, since each run will, in general, depend on random bits generated inside the algorithm. Similarly, the time taken by an algorithm is a function from inputs to distributions of times. "Constant time" means that this function is constant: the distribution of algorithm time for input i matches the distribution of algorithm time for input i'. In other words, the algorithm time provides no information about the input.

In particular, if the input is a CSIDH curve and a private key, and the output is the result of the CSIDH action, then the algorithm time provides no information about the private key, and provides no information about the output.

Avoiding data flow from inputs to branches and array indices is sufficient to ensure the constant-time property for many definitions of algorithm time, and is the main focus of work on constant-time algorithms for CSIDH, including the work in this paper. Beware, however, that this is not sufficient if the definition changes to allow, e.g., a variable-time division instruction, like the division instructions on most computers.

The constant-time property also does not mean that time is deterministic. The paper [5] aims for time to be constant *and* deterministic, so as to be able to run in superposition on a quantum computer, but this costs extra and is not necessary for the objective of stopping timing attacks.

Structurally, every claim of constant-time *software* in the literature relies on various CPU instructions taking constant time, and could be undermined by CPU manufacturers adding timing variations to those instructions. The literature on constant-time software

generally assumes, for example, that multiplication instructions take constant time, and declares that CPUs with variable-time multiplication instructions are out of scope. Formally, the constant-time claims are in a model of "time" where various instructions, including multiplications, take constant time.

Dummy isogenies. [16] used *dummy isogenies* to obtain a fixed number of isogenies per group action evaluation. Essentially, if e_i is sampled such that $|e_i| \leq m_i$, this amounts to computing m_i isogenies of degree ℓ_i, where $m_i - |e_i|$ of these are dummy computations whose results are simply discarded.[3] As noted in Section 2.2, this might require more than m_i attempts to sample a point of order ℓ_i, due to the point rejection probability of $1/\ell_i$. However, the number of attempts only depends on randomness and m_i, and is thus independent of the choice of e_i.

1-point and 2-point approaches. It is observed in [16] that if we compute multiple isogenies from a single sampled point, then the running time of the algorithm depends on the sign distribution of the private keys. Indeed, when a single point is sampled, only ℓ_i-isogenies with equal signs of the corresponding e_i can be combined in a strategy. Since this approach of combining isogenies is desirable for efficiency (see Section 2.3), [16] proposed eliminating this dependency by sampling e_i from $[0, 2m_i]$ instead of $[-m_i, m_i]$, although this requires computing twice as many isogenies per degree.

In order to mitigate this slowdown, [20] proposed sampling two points, $T_0 \in E_A(\mathbb{F}_p)$ and $T_1 \in \tilde{E}_A(\mathbb{F}_p)$. For each isogeny in the sequence, one picks the kernel generator according to the sign of the corresponding e_i. This approach combines isogeny computations independent of key signs, and thus goes back to sampling e_i from $[-m_i, m_i]$ at the cost of pushing two points through each isogeny instead of one.

A dummy-free variant of the 2-point approach was proposed in [11]. This requires roughly twice as many isogenies, but may be useful in situations where fault-injection attacks play an important role. We return to this approach in Appendix A.

3 Batching and key spaces

The main conceptual novelty in CTIDH is the organization of primes and isogenies in batches. For this we define a new batch-oriented key space, which is slightly more complicated than the key spaces $\mathcal{K}_m$ and $\mathcal{K}_m^+$ mentioned in Section 2.

Batching primes. In CTIDH, the sequence of primes $(\ell_1, \ldots, \ell_n)$ is partitioned into a series of *batches*: subsequences of consecutive primes. Let $0 < B \leq n$ be the number of batches; we represent the sequence of the batch sizes by a vector $N = (N_1, \ldots, N_B) \in \mathbb{Z}_{>0}^B$ with $\sum_{i=1}^B N_i = n$. We relabel the primes in batches as: $(\ell_{1,1}, \ldots, \ell_{1,N_1}) := (\ell_1, \ldots, \ell_{N_1})$, $(\ell_{2,1}, \ldots, \ell_{2,N_2}) := (\ell_{N_1+1}, \ldots, \ell_{N_1+N_2}), \ldots, (\ell_{B,1}, \ldots, \ell_{B,N_B}) := (\ell_{n-N_B+1}, \ldots, \ell_n)$. If $\ell_{i,j}$ corresponds to ℓ_k, then we also write $\mathfrak{l}_{i,j}$ for $\mathfrak{l}_k$ and $e_{i,j}$ for e_k.

Example 1. Say we have $n = 6$ primes, $(\ell_1, \ldots, \ell_6)$. If we take $B = 3$ and $N = (3, 2, 1)$, then $(\ell_{1,1}, \ell_{1,2}, \ell_{1,3}) = (\ell_1, \ell_2, \ell_3)$, $(\ell_{2,1}, \ell_{2,2}) = (\ell_4, \ell_5)$, and $(\ell_{3,1}) = (\ell_6)$.

Batching isogenies. Consider the i-th batch of primes $(\ell_{i,1}, \ldots, \ell_{i,N_i})$. Rather than setting a bound $m_{i,j} \geq |e_{i,j}|$ for the number of $\ell_{i,j}$-isogenies for each $1 \leq j \leq N_i$, we set a bound $m_i \geq \sum_{j=1}^{N_i} |e_{i,j}|$ and compute m_i isogenies from the batch $(\ell_{i,1}, \ldots, \ell_{i,N_i})$. This looks analogous to the use of dummy operations in the previous constant-time algorithms,

[3] In [16] some other computations are performed inside dummy isogenies, which facilitate later steps in the algorithm. We omit the details here, since we only use simple dummy isogenies as described above.

but it gives a larger keyspace per isogeny computed because of the ambiguity between the degrees in a batch. Moreover, we will show in Section 5.2.2 that it is possible to evaluate any isogeny within one batch in the same constant time.

Extreme batching choices correspond to well-known approaches to the group action evaluation: one prime per batch ($B = n$ and $N = (1, \ldots, 1)$) was considered in [10]; one n-prime batch ($B = 1$ and $N = (n)$) is considered in [5] for the quantum oracle evaluation and in [18] as a speedup for CSIDH. The intermediate cases are new, and, as we will show, faster.

The new key space. For $N \in \mathbb{Z}_{>0}^B$ and $m \in \mathbb{Z}_{\geq 0}^B$, we define

$$\mathcal{K}_{N,m} := \left\{(e_1, \ldots, e_n) \in \mathbb{Z}^n \mid \textstyle\sum_{j=1}^{N_i} |e_{i,j}| \leq m_i \text{ for } 1 \leq i \leq B\right\}.$$

We may see $\mathcal{K}_{N,m}$ as a generalization of $\mathcal{K}_m$.

Lemma 1. *We have*

$$\#\mathcal{K}_{N,m} = \prod_{i=1}^{B} \Phi(N_i, m_i), \quad \textit{where} \quad \Phi(x, y) = \sum_{k=0}^{\min\{x,y\}} \binom{x}{k} 2^k \binom{y}{k}$$

counts the vectors in $\mathbb{Z}^x$ with 1-norm at most y.

Proof. The size of the key space is the product of the sizes for each batch. In $\Phi(x, y)$ the number of nonzero entries in the x positions is k and there are $\binom{x}{k}$ ways to determine which entries are nonzero. For each of the nonzero entries there are 2 ways to choose the sign. The vector of partial sums over these k nonzero entries has k different integers in $[1, y]$ and each vector uniquely matches one assignment of partial sums. There are $\binom{y}{k}$ ways to pick k different integers in $[1, y]$. □

4 Isogeny atomic blocks

In this section we formalize the concept of *isogeny atomic blocks* (ABs), subroutines that have been widely used in constant-time CSIDH algorithms but never formalized before. The first step of an algorithm chooses a series of degrees for which isogenies still need to be computed, and then uses, for example, the multiplicative strategy (Section 2.3) to compute a sequence of isogenies of those degrees. The next step chooses a possibly different series of degrees, and computes another sequence of isogenies. Each step of the computation is the evaluation of not one isogeny, but a sequence of isogenies. Atomic blocks formalize these steps.

Square-free ABs generalize the approach we take when evaluating the CSIDH group action with the traditional key spaces $\mathcal{K}_m$ and $\mathcal{K}_m^+$ as in Algorithm 1. *Restricted square-free ABs* are used to evaluate the group action using the batching idea with keys in $\mathcal{K}_{N,m}$; with details in Algorithm 2. We postpone the explicit construction of ABs to Section 5.

4.1 Square-free atomic blocks

Definition 1 (Square-free ABs)**.** Let $R \subseteq \{-1, 0, 1\}$ and $I = (I_1, \ldots, I_k) \in \mathbb{Z}^k$ such that $1 \leq I_1 < I_2 < \cdots < I_k \leq n$. A *square-free atomic block* of length k is a probabilistic algorithm $\alpha_{R,I}$ taking inputs $A \in \mathcal{M}$ and $\epsilon \in R^k$ and returning $A' \in \mathcal{M}$ and $f \in \{0, 1\}^k$ such that $E_{A'} = (\prod_i \mathfrak{l}_{I_i}^{f_i \cdot \epsilon_i}) \star E_A$, satisfying the following two properties:

1. there is a function σ such that, for each (A, ϵ), the distribution of f, given that (A', f) is returned by $\alpha_{R,I}$ on input (A, ϵ), is $\sigma(R, I)$, and

Algorithm 1: Generalization of [20, Algorithm 3], replacing the inner loop with any square-free AB with $R = \{-1, 0, 1\}$. Keys are in $\mathcal{K}_m$.

Parameters: $m = (m_1, \ldots, m_n)$
Input: $A \in \mathcal{M}$, $e = (e_1, \ldots, e_n) \in \mathcal{K}_m$
Output: A' with $E_{A'} = (\prod_i \mathfrak{l}_i^{e_i}) \star E_A$

1 $(\mu_1, \ldots, \mu_n) \leftarrow (m_1, \ldots, m_n)$
2 **while** $(\mu_1, \ldots, \mu_n) \neq (0, \ldots, 0)$ **do**
3 Let $I = (I_1, \ldots, I_k)$ s.t. $I_1 < \cdots < I_k$ and $\{I_1, \ldots, I_k\} = \{1 \leq i \leq n \mid \mu_i > 0\}$
4 **for** $1 \leq i \leq k$ **do**
5 $\epsilon_i \leftarrow \texttt{Sign}(e_{I_i})$ `// 1 if` $e_{I_i} > 0$`; 0 if` $e_{I_i} = 0$`; -1 if` $e_{I_i} < 0$
6 $(A, f) \leftarrow \alpha_{R,I}(A, (\epsilon_1, \ldots, \epsilon_k))$ `// Square-free AB`
7 **for** $1 \leq i \leq k$ **do**
8 $(\mu_{I_i}, e_{I_i}) \leftarrow (\mu_{I_i} - f_i, e_{I_i} - \epsilon_i \cdot f_i)$
9 **return** A

2. there is a function τ such that, for each (A, ϵ) and each f, the distribution of the time taken by $\alpha_{R,I}$, given that (A', f) is returned by $\alpha_{R,I}$ on input (A, ϵ), is $\tau(R, I, f)$.

Suppose an algorithm evaluates the group action on input $e \in \mathcal{K}$ and $A \in \mathcal{M}$ using a sequence of square-free AB calls $(A', f) \leftarrow \alpha_{R,I}(A, \epsilon)$. If in each step the choice of R and I are independent of e, the algorithm does not leak information about e through timing.

This is illustrated by Algorithm 1, which expresses the constant-time group action from [20] using a sequence of square-free ABs with $R = \{-1, 0, 1\}$ to evaluate the action for keys in $\mathcal{K}_m$. The choices of R and I are independent of e for each AB $\alpha_{R,I}$, and all other steps can be easily made constant-time. The choice of I in Step 3 may vary between different executions, due to the varying failure vectors f of previously evaluated ABs. However this only depends on the initial choice of m_i, and is independent of e.

Remark 2. The constant-time group action from [16] can also be expressed simply in terms of ABs. The algorithm is extremely similar to Algorithm 1, using $\mathcal{K}_m^+$ in place of $\mathcal{K}_m$ (the algorithm of [16] uses $m = (10, \ldots, 10)$) and $R = \{0, 1\}$ in place of $\{-1, 0, 1\}$. Line 5 can be simplified to $\epsilon_i \leftarrow 1$ if $e_{I_i} \neq 0$, or 0 if $e_{I_i} = 0$.

Remark 3. The distribution of f depends on how the ABs are constructed. In [16] and [20], $\Pr(f_i = 0) = 1/\ell_{I_i}$ for all i. In [12], f is always $(1, 1, \ldots, 1)$.

4.2 Restricted square-free atomic blocks

In the language of Section 3, restricted square-free ABs are generalizations of square-free ABs that further do not leak information on which of the primes we have chosen from a batch.

Definition 2 (Restricted square-free ABs)**.** Let $R \subseteq \{-1, 0, 1\}$, $B \geq 1$, and $I = (I_1, \ldots, I_k) \in \mathbb{Z}^k$ such that $1 \leq I_1 < I_2 < \cdots < I_k \leq B$. A *restricted square-free atomic block* of length k is a probabilistic algorithm $\beta_{R,I}$ taking inputs $A \in \mathcal{M}$, $\epsilon \in R^k$, and $J \in \mathbb{Z}^k$ with $1 \leq J_i \leq N_{I_i}$ for all $1 \leq i \leq k$, and returning $A' \in \mathcal{M}$ and $f \in \{0, 1\}^k$ such that $E_{A'} = (\prod_i \mathfrak{l}_{I_i, J_i}^{f_i \cdot \epsilon_i}) \star E_A$, satisfying the following two properties:

1. there is a function σ such that, for each (A, ϵ, J), the distribution of f, given that (A', f) is returned by $\beta_{R,I}$ on input (A, ϵ, J), is $\sigma(R, I)$; and
2. there is a function τ such that, for each (A, ϵ, J) and each f, the distribution of the time taken by $\beta_{R,I}$, given that (A', f) is returned by $\beta_{R,I}$ on input (A, ϵ, J), is $\tau(R, I, f)$.

Algorithm 2 uses restricted square-free ABs with $R = \{-1, 0, 1\}$ to compute group actions for keys in $\mathcal{K}_{N,m}$; it may be considered a generalization of Algorithm 1.

Algorithm 2: A constant-time group action for keys in $\mathcal{K}_{N,m}$ based on restricted square-free ABs with $R = \{-1, 0, 1\}$.

Parameters: N, m, B
Input: $A \in \mathcal{M}$, $e = (e_1, \ldots, e_n) \in \mathcal{K}_{N,m}$
Output: A' with $E_{A'} = (\prod_i \mathfrak{l}_i^{e_i}) \star E_A$

$(\mu_1, \ldots, \mu_B) \leftarrow (m_1, \ldots, m_B)$
while $(\mu_1, \ldots, \mu_B) \neq (0, \ldots, 0)$ **do**
  Let $I = (I_1, \ldots, I_k)$ s.t. $I_1 < \cdots < I_k$ and $\{I_1, \ldots, I_k\} = \{1 \leq i \leq B \mid \mu_i > 0\}$
  for $1 \leq i \leq k$ **do**
    if *there exists* j *such that* $e_{I_i,j} \neq 0$ **then**
      $J_i \leftarrow$ some such j
    else
      $J_i \leftarrow$ any element of $\{1, \ldots, N_{I_i}\}$
    $\epsilon_i \leftarrow \texttt{Sign}(e_{I_i,J_i})$ `// 1 if` $e_{I_i,J_i} > 0$`; 0 if` $e_{I_i,J_i} = 0$`; -1 if` $e_{I_i,J_i} < 0$
  $(A, f) \leftarrow \beta_{R,I}(A, (\epsilon_1, \ldots, \epsilon_k), J)$ `// Restricted square-free AB`
  for $1 \leq i \leq k$ **do**
    $(\mu_{I_i}, e_{I_i,J_i}) \leftarrow (\mu_{I_i} - f_i, e_{I_i,J_i} - \epsilon_i \cdot f_i)$
return A

5 Evaluating atomic blocks in constant time

This section introduces the algorithm used in CTIDH to realize the restricted square-free atomic block $\beta_{R,I}$ introduced in Section 4. Throughout this section, R is $\{-1, 0, 1\}$.

As a warmup, Section 5.1 recasts the inner loop of [20, Algorithm 3] as a realization of the square-free atomic block $\alpha_{R,I}$. We first present the algorithm in a simpler variable-time form (Algorithm 3) and then explain the small changes needed to eliminate timing leaks, obtaining $\alpha_{R,I}$.

Section 5.2 presents our new algorithm to realize $\beta_{R,I}$. The extra challenge here is to hide which prime is being used within each batch. Again we begin by presenting a simpler variable-time algorithm (Algorithm 4) and then explain how to eliminate timing leaks.

5.1 Square-free atomic blocks for isogeny evaluation

Algorithm 3 translates the inner loop of [20, Algorithm 3] to the AB framework. The inputs are $A \in \mathcal{M}$ and $\epsilon \in \{-1, 0, 1\}^k$. The goal is to compute k isogenies of degrees $\ell_{I_1}, \ldots, \ell_{I_k}$, but some of these computations may fail. The outputs are a vector $f \in \{0, 1\}^k$ recording which of the computations succeeded, and A' such that $(\prod_i \mathfrak{l}_{I_i}^{f_i \cdot \epsilon_i}) \star E_A = E_{A'}$.

The algorithm uses the 2-point approach with dummy isogenies. It uses two subroutines:

- `UniformRandomPoints` takes $A \in \mathcal{M}$, and returns a uniform random pair of points (T_0, T_1), with $T_0 \in E_A(\mathbb{F}_p)$ and $T_1 \in \tilde{E}_A(\mathbb{F}_p)$; i.e., T_0 is a uniform random element of $E_A(\mathbb{F}_p)$, and T_1, independent of T_0, is a uniform random element of $\tilde{E}_A(\mathbb{F}_p)$.
- `Isogeny` takes $A \in \mathcal{M}$, a point P in $E_A(\mathbb{F}_{p^2})$ with x-coordinate in $\mathbb{F}_p$ generating the kernel of an ℓ_{I_j}-isogeny $\varphi : E_A \to E_{A'} = E_A/\langle P \rangle$, and a tuple of points $(Q_1, \ldots, Q_t)$, and returns A' and $(\varphi(Q_1), \ldots, \varphi(Q_t))$.

See Appendix B for analysis of the Elligator alternative to `UniformRandomPoints`.

Algorithm 3: Inner loop of [20, Algorithm 3].

Parameters: $k \in \mathbb{Z}, R = \{-1, 0, 1\}, I \in \mathbb{Z}^k_{\geq 0}$
Input: $A \in \mathcal{M}, \epsilon \in \{-1, 0, 1\}^k$
Output: $A' \in \mathcal{M}, f \in \{0, 1\}^k$

1 $(T_0, T_1) \leftarrow$ `UniformRandomPoints(`A`)`
2 $(T_0, T_1) \leftarrow ([r]T_0, [r]T_1)$ **where** $r = 4\prod_{i \notin I} \ell_i$
3 $r' \leftarrow \prod_{i \in I} \ell_i$
4 **for** $j = k$ **down to** 1 **do**
5 $\quad r' \leftarrow r'/\ell_{I_j}$
6 $\quad s \leftarrow$ `SignBit(`ϵ_j`)` `// 1 if` $\epsilon_j < 0$`, otherwise 0`
7 $\quad P \leftarrow [r']T_s$
8 $\quad$ **if** $P \neq \infty$ **then** `// branch without secret information`
9 $\quad\quad f_j \leftarrow 1$
10 $\quad\quad (A', (T_0', T_1')) \leftarrow$ `Isogeny(`A, P, (T_0, T_1), I_j`)`
11 $\quad\quad$ **if** $\epsilon_j \neq 0$ **then** `// branch with secret information`
12 $\quad\quad\quad (A, T_0, T_1) \leftarrow (A', T_0', T_1')$
13 $\quad\quad$ **else**
14 $\quad\quad\quad T_s \leftarrow [\ell_{I_j}]T_s$
15 $\quad$ **else**
16 $\quad\quad f_j \leftarrow 0$
17 $\quad T_{1-s} \leftarrow [\ell_{I_j}]T_{1-s}$
18 **return** A, f

Remark 4. Algorithm 3 uses a multiplicative strategy, but it can easily be modified to use a SIMBA or point-pushing strategy, which is much more efficient in general [20, 13]. The isogeny algorithm can be Vélu or $\sqrt{}$élu, whichever is more efficient for the given degree.

5.1.1 Modifying Algorithm 3 to eliminate timing leaks

The following standard modifications to Algorithm 3 produce an algorithm meeting Definition 1, the definition of a square-free atomic block.

Observe first that $f_j = 1$ if and only if the prime ℓ_{I_j} divides the order of the current T_s. This is equivalent to ℓ_{I_j} dividing the order of the initially sampled point T_s (since T_s has been modified only by multiplication by scalars that are not divisible by ℓ_{I_j}, and by isogenies of degrees not divisible by ℓ_{I_j}). This has probability $1 - 1/\ell_{I_j}$, since the initial T_s is a uniform random point in a cyclic group of size $p + 1$. These probabilities are independent across j, since (T_0, T_1) is a uniform random pair of points.

To summarize, the distribution of the f vector has position j set with probability $1 - 1/\ell_{I_j}$, independently across j. This distribution is a function purely of I, independent of (A, ϵ), as required. What follows are algorithm modifications to ensure that the time distribution is a function purely of (I, f); these modifications do not affect f.

Step 7, taking T_0 if $s = 0$ or T_1 if $s = 1$, is replaced with a constant-time point selection: e.g., taking the bitwise XOR $T_0 \oplus T_1$, then ANDing each bit with s, and then XORing the result with T_0. Similar comments apply to the subsequent uses of T_s and T_{1-s}. It is simplest to merge all of these selections into a constant-time swap of T_0, T_1 when $s = 1$, followed by a constant-time swap back at the bottom of the loop. The adjacent swaps at the bottom of one loop and the top of the next loop can be merged, analogous merging is standard in constant-time versions of the Montgomery ladder for scalar multiplication.

Step 11 determines whether an actual isogeny or a dummy isogeny has to be computed. The conditional assignment to (A, T_0, T_1) in the first case is replaced with unconditional

constant-time point selection. The conditional operation in the second case is replaced with an unconditional operation, multiplying T_s by ℓ_{I_j} in both cases. This changes the point T_s in the first case, but does not change the order of T_s (since the isogeny has already removed ℓ_{I_j} from the order of T_s in the first case), and all that matters for the algorithm is the order. See [16, 20] for a slightly more efficient approach, merging the multiplication by ℓ_{I_j} into a dummy isogeny computation.

The branch in Step 8 is determined by public information f_j and does not need to be modified. The isogeny computation inside `Isogeny` takes constant time with standard algorithms; at a lower level, arithmetic in $\mathbb{F}_p$ is handled by constant-time subroutines, not by subroutines that try to save time by suppressing leading zero bits. The computation of `UniformRandomPoints` takes variable time with standard algorithms, but the time distribution is independent of the curve provided as input.

The total time is the sum for initialization (`UniformRandomPoints`, computation of r and r', initial scalar multiplication), f_j computation (division, scalar multiplications, selection), and computations when $f_j = 1$ (`Isogeny`, scalar multiplication, selection). This sum is a function purely of (I, f), independent of (A, ϵ), as required.

5.2 Restricted square-free atomic blocks

We now consider the more difficult goal of hiding which isogeny is being computed within each batch. We present first the high-level algorithm (Algorithm 4), then the `PointAccept` (Section 5.2.1) and `MatryoshkaIsogeny` (Section 5.2.2) subroutines, and finally the algorithm modifications (Section 5.2.3) to meet Definition 2.

The inputs to Algorithm 4 are $A \in \mathcal{M}$, $\epsilon \in \{-1, 0, 1\}^k$, and $J \in \mathbb{Z}^k$. The goal is to compute k isogenies of degrees $\ell_{I_1,J_1}, \ldots, \ell_{I_k,J_k}$. The outputs are $A' \in \mathcal{M}$ and $f \in \{0, 1\}^k$ such that $(\prod_i \mathfrak{l}_{I_i,J_i}^{f_i \cdot \epsilon_i}) \star E_A = E_{A'}$.

Like Algorithm 3, Algorithm 4 uses a 2-point approach and dummy isogenies. It uses the following subroutines:

- `UniformRandomPoints` is as before.
- `PointAccept` replaces the check $P \neq \infty$ to prevent timing leakage. It takes a point P and $I_j, J_j \in \mathbb{Z}$ such that P either has order ℓ_{I_j,J_j} or 1, and outputs either 0 or 1, under the condition that the output is 0 whenever $P = \infty$.
- `MatryoshkaIsogeny` replaces `Isogeny` from Algorithm 3. There is an extra input J_j indicating the secret position within a batch.

Note that the output of `PointAccept` can be 0 when $P \neq \infty$, so we add a multiplication by ℓ_{I_j,J_j} in Step 14 to make sure we continue the loop with points of expected order.

5.2.1 PointAccept

Step 8 of Algorithm 4 computes a potential kernel generator, P. The probability that $P = \infty$ is $1/\ell_{I_j,J_j}$, which depends on J_j. For the batch $(\ell_{I_j,1}, \ldots, \ell_{I_j,N_{I_j}})$, `PointAccept` artificially increases this probability to $1/\ell_{I_j,1}$, independent of ℓ_{I_j,J_j}, by tossing a coin with success probability

$$\gamma = \frac{\ell_{I_j,J_j} \cdot (\ell_{I_j,1} - 1)}{\ell_{I_j,1} \cdot (\ell_{I_j,J_j} - 1)}$$

and only returning $f_j = 1$ if $P \neq \infty$ *and* the coin toss is successful. The probability that the output is 1 is then $\gamma \cdot (1 - 1/\ell_{I_j,J_j}) = 1 - 1/\ell_{I_j,1}$, which is independent of J_j. Thus the batch can fail publicly.

Algorithm 4: The CTIDH inner loop.

Parameters: $k \in \mathbb{Z}, R = \{-1, 0, 1\}, I \in \mathbb{Z}^k_{\geq 0}$
Input: $A \in \mathcal{M}, \epsilon \in \{-1, 0, 1\}^k, J \in \mathbb{Z}^k_{>0}$
Output: $A' \in \mathcal{M}, f \in \{0, 1\}^k$

$(T_0, T_1) \leftarrow$ `UniformRandomPoints(`A`)`
$(T_0, T_1) \leftarrow ([r]T_0, [r]T_1)$ **where** $r = 4\prod_{i \notin I} \prod_{1 \leq j \leq N_i} \ell_{i,j}$
$(T_0, T_1) \leftarrow ([\tilde{r}]T_0, [\tilde{r}]T_1)$ **where** $\tilde{r} = \prod_{i \in I} \prod_{1 \leq j \leq N_i, j \neq J_i} \ell_{i,j}$ `// hide selection`
$r' \leftarrow \prod_{i \in I} \ell_{i,J_i}$ `// hide selection`
for $j = k$ **down to** 1 **do**
  $r' \leftarrow r'/\ell_{I_j,J_j}$ `// hide` ℓ_{I_j,J_j}`, batch is public`
  $s \leftarrow$ `SignBit(`ϵ_j`)` `// 1 if` $\epsilon_j < 0$`, otherwise 0`
  $P \leftarrow [r']T_s$ `// hide` ℓ_{I_j,J_j}`, batch is public`
  $f_j \leftarrow$ `PointAccept(`P`,` I_j`,` J_j`)`
  if $f_j = 1$ **then** `// this branch is on public information`
    $(A', (T_0', T_1')) \leftarrow$ `MatryoshkaIsogeny(`A`,` P`,` $(T_0, T_1), I_j, J_j$`)`
    if $\epsilon_j \neq 0$ **then** `// branch with secret information`
      $(A, T_0, T_1) \leftarrow (A', T_0', T_1')$
  $(T_0, T_1) \leftarrow ([\ell_{I_j,J_j}]T_0, [\ell_{I_j,J_j}]T_1)$ `// hide selection`
return A, f

5.2.2 MatryoshkaIsogeny

`MatryoshkaIsogeny` replaces the `Isogeny` computation. It takes the Montgomery coefficient of a curve E_A, a batch $(\ell_{i,1}, \ldots, \ell_{i,N_i})$, an isogeny index j within the batch, a point P of order $\ell_{i,j}$ generating the kernel of an isogeny $\varphi : E_A \to E_A/\langle P \rangle = E_{A'}$, and a tuple of points $(Q_1, \ldots, Q_t)$, and returns A' and $(\varphi(Q_1), \ldots, \varphi(Q_t))$. `MatryoshkaIsogeny` is computed with cost independent of j.

For Vélu's formulas, [5] showed how to compute any ℓ_i-isogeny for $\ell_i \leq \ell$ using the computation of an ℓ-isogeny and masking. The first step of computing (1) is to compute $x(P), x([2]P), \ldots, x([(\ell-1)/2]P)$. This includes the computation for smaller ℓ_i; [5] described this as a Matryoshka-doll property.

In this paper we specialize the √élu formulas so as to obtain a Matryoshka-doll structure. We define the sets U and V, introduced in Section 2.2, as the optimal choices for the *smallest* degree in the batch: i.e., $\ell_{i,1}$. The leftover set W is chosen to make the formulas work even for the *largest* prime ℓ_{i,N_i} in the batch. Then the baby-step giant-step algorithm stays unchanged; while we iterate through W we save the intermediate results corresponding to all degrees $\ell_{i,j}$ in the batch. In the final step, we select the result corresponding to the index j that we wanted to compute.

The sets U and V have size around $\sqrt{\ell_{i,1}}$. If the primes in the batch are sufficiently close then the rounded values match or are marginally different, meaning that the Matryoshka-like formulas are at worst marginally slower than the optimal formulas for ℓ_{i,N_i}.

5.2.3 Modifying Algorithm 4 to eliminate timing leaks

We now indicate algorithm modifications to meet Definition 2, the definition of a restricted square-free atomic block.

As in Section 5.1.1, we begin with the distribution of f. For each input (A, ϵ, J), the distribution has f_j set with probability $1 - 1/\ell_{I_j,1}$ (not $1 - 1/\ell_{I_j,J_j}$; see Section 5.2.1), independently across j. This distribution is a function purely of I, independent of (A, ϵ, J), as required. What remains is to ensure that the time distribution is a function purely of

(I, f).

There are secret scalars $\tilde{r}$, r', and ℓ_{I_j,J_j} used in various scalar multiplications in Steps 3, 8, and 14. Standard integer-arithmetic algorithms that dynamically suppress leading zero bits are replaced by constant-time algorithms that always use the maximum number of bits, and variable-time scalar-multiplication algorithms are replaced by a constant-time Montgomery ladder, as in [5]. It is straightforward to compute an upper bound on each scalar in Algorithm 4. See Section 7 for faster alternatives.

Section 5.2.2 explains how to compute `MatryoshkaIsogeny` in time that depends only on the batch, not on the selection of a prime within the batch. Everything else is as in Section 5.1.1: the distribution of `UniformRandomPoints` timings is independent of the inputs, Step 8 uses constant-time selection, the branch in Step 12 is replaced by constant-time selection, and the branch in Step 10 does not need to be modified.

6 Strategies and parameters for CTIDH

The optimization process for previous constant-time algorithms for CSIDH has two levels. The bottom level tries to minimize the cost of each AB, for example by optimizing $\sqrt{\text{élu}}$ parameters and searching for a choice of strategy from Section 2.3. The top level searches for a choice of exponent bounds $m = (m_1, \ldots, m_n)$, trying to minimize the total AB cost subject to the key space reaching a specified size. A cost function that models the cost of an AB, taking account of the bottom-level search, is plugged into the top-level search.

Optimizing CTIDH is more complicated. There is a new top level, searching for a choice of batch sizes $N = (N_1, \ldots, N_B)$. These batch sizes influence the success chance and cost of an AB at the bottom level: see Sections 5.2.1 and 5.2.2. They also influence the total cost of any particular choice of 1-norm bounds $m = (m_1, \ldots, m_B)$ at the middle level. The size of the key space depends on both N and m; see Lemma 1.

This section describes a reasonably efficient method to search for CTIDH parameters.

Strategies for CTIDH. We save time at the lowest level of the search by simply using multiplicative strategies. As in previous papers, it would be easy to adapt Algorithm 4 to use SIMBA or optimized point-pushing strategies or both, giving many further parameters that could be explored with more search time, but this is unlikely to produce large benefits.

Seen from a high level, evaluating ABs multiplicatively in CTIDH has a similar effect to SIMBA strategies for previous algorithms. For example, SIMBA-N_1 for traditional batch sizes $(1, \ldots, 1)$ limits each AB to at most n/N_1 isogenies (if n is divisible by N_1), in order to save multiplicative effort. Now consider CTIDH where all B batches have size N_1, i.e., $N = (N_1, \ldots, N_1)$. Each CTIDH AB then computes at most $B = n/N_1$ isogenies, saving multiplicative effort in the same way.

One could split a CTIDH AB into further SIMBA prides, but [16] already shows that most of the benefit of SIMBA comes from putting some cap on the number of isogenies in an AB; the exact choice of cap is relatively unimportant. One could also try to optimize point-pushing strategies as an alternative to multiplicative strategies, as an alternative to SIMBA, or within each SIMBA pride, but the searches in [15] and [13] suggest that optimizing these strategies saves at most a small percentage in the number of multiplications, while incurring overhead for managing additional points.

Cost functions for CTIDH. The search through various CTIDH batching configuration vectors N and 1-norm bound vectors m tries to minimize a function $C(N, m)$, a model of the cost of a group-action evaluation. The numerical search examples later in this section use the following cost function: the average number of multiplications (counting squarings as multiplications) used by the CTIDH algorithms, including the speedups described in Section 7.

One way to compute this function is to statistically approximate it: run the software from Section 7 many times, inspect the multiplication counter built into the software, and take the average over many experiments. A more efficient way to compute the same function with the same accuracy is with a simulator that skips the multiplications but still counts how many there are. Our simulator, despite being written in Python, is about 50 times faster than the software from Section 7.

However, using a statistical approximation raises concerns about the impact of statistical variations. So, instead of using the software or the simulator, we directly compute the average cost of the first AB, the average cost of the second AB, etc., stopping when the probability of needing any further AB is below 10^{-9}.

Batch b, with smallest prime $\ell_{b,1}$, has success probability $1-1/\ell_{b,1}$ from each AB, so the chance q_b of reaching m_b successes within R ABs is the sum of the coefficients of $x^{m_b}, x^{m_b+1}, \ldots$ in the polynomial $(1/\ell_{b,1}+(1-1/\ell_{b,1})x)^R$. Batches are independent, so $q_1q_2\cdots q_B$ is the probability of not needing any further AB. Note that multiplying the polynomial $(1/\ell_{b,1}+(1-1/\ell_{b,1})x)^R$ by $1/\ell_{b,1}+(1-1/\ell_{b,1})x$ for each increase in R is more efficient than computing binomial coefficients.

Computing the cost of an AB (times the probability that the AB occurs) is more complicated. Splitting the analysis into 2^B cases—e.g., one case, occurring with probability $(1-q_1)(1-q_2)\cdots(1-q_B)$, is that all B batches still remain to be done—might be workable, since B is not very large and one can skip cases that occur with very low probability. We instead take the following approach. Fix b. The probability that batch b is in the AB is $1-q_b$; the probability that batch a is in the AB for exactly j values $a<b$ is the coefficient of x^j in the polynomial $\prod_{a<b}(q_a+(1-q_a)x)$; and the probability that batch c is in the AB for exactly k values $c>b$ is the coefficient of x^k in the polynomial $\prod_{c>b}(q_c+(1-q_c)x)$. There are $O(B^2)$ possibilities for (j,k); each possibility determines the total number of batches in the AB and the position of b in the AB, assuming b is in the AB. For the AB algorithms considered here, this is enough information to determine the contribution of batch b to the cost of the AB. Our Python implementation of this approach has similar cost to 100 runs of the simulator, depending on B.

We also explored various simpler possibilities for cost functions. A deterministic model of ABs is easier to compute and simulates real costs reasonably well, leading to parameters whose observed costs were consistently within 10% of the best costs we found via the cost function defined above.

Optimizing the 1-norm bounds. Given a fixed configuration N of B batches, we use a greedy algorithm as in [13] to search for a 1-norm bound vector m as follows:

1. Choose an initial $m=(m_1,\ldots,m_B)$ such that $\mathcal{K}_{N,m}$ is large enough, and set $C_{\min}\leftarrow C(N,m)$.
2. For each i in $\{1,\ldots,B\}$, do the following:
   (a) Set $\tilde{m}\leftarrow(m_1,\ldots,m_{i-1},m_i-1,m_{i+1},\ldots,m_B)$.
   (b) If $\mathcal{K}_{N,\tilde{m}}$ is large enough, set $(m,C_{\min})\leftarrow(\tilde{m},C(N,\tilde{m}))$.
   (c) Else, set $\tilde{m}'\leftarrow\tilde{m}$, and for each $j\neq i$ in $\{1,\ldots,B\}$ do the following:
      i. Set $\tilde{m}\leftarrow(\tilde{m}'_1,\ldots,\tilde{m}'_{j-1},\tilde{m}'_j+1,\tilde{m}'_{j+1},\ldots,\tilde{m}'_B)$.
      ii. If $\mathcal{K}_{N,\tilde{m}}$ is too small, recursively go to Step 2(c).
      iii. Else, if $C(N,\tilde{m})<C_{\min}$, set $(m,C_{\min})\leftarrow(\tilde{m},C(N,\tilde{m}))$.
3. If $C_{\min}$ was updated in Step 2, then repeat Step 2.
4. Return $(m,C_{\min})$.

This algorithm applies small changes to the bound vector m at each step, reducing one entry while possibly increasing others. Obviously, this finds only a *locally* optimal m with respect to these changes and the initial choice of m in Step 1; different choices generally produce different results.

One way to choose an initial m is to try $(1, 1, \ldots, 1)$, then $(2, 2, \ldots, 2)$, etc., stopping when $\mathcal{K}_{N,m}$ is large enough. Another approach, in the context of the N search described below, is to start from the best m found for the parent N, and merely increase the first component of m until $\mathcal{K}_{N,m}$ is large enough; usually at most one increase is needed.

The algorithm involves at least $B(B-1)$ evaluations of the cost function for the final pass through Step 2. It can involve many more evaluations if there are many recursive calls or if there are many improvements to m, but usually these are small effects.

Optimizing the prime batches. We optimize N via a similar greedy algorithm, using the algorithm above as a subroutine. For a *fixed* number of batches B, we proceed as follows:

1. Choose an initial $N = (N_1, \ldots, N_B)$ with $\sum_i N_i = n$, and let $(m, C_{\min})$ be the output of the algorithm above applied to N.
2. For each $i \in \{1, \ldots, B\}$, do the following:
   (a) Set $\tilde{N}^i \leftarrow (N_1, \ldots, N_{i-1}, N_i - 1, N_{i+1}, \ldots, N_B)$.
   (b) For each $j \neq i$ in $\{1, \ldots, B\}$,
      i. Set $\tilde{N}^{i,j} \leftarrow (\tilde{N}^i_1, \ldots, \tilde{N}^i_{j-1}, \tilde{N}^i_j + 1, \tilde{N}^i_{j+1}, \ldots, \tilde{N}^i_B)$.
      ii. Let $(\tilde{m}, \tilde{C})$ be the output of the algorithm above applied to $N^{i,j}$.
      iii. If $\tilde{C} < C_{\min}$, then update $(N, m, C_{\min}) \leftarrow (\tilde{N}^{i,j}, \tilde{m}, \tilde{C})$.
3. If $C_{\min}$ was updated in Step 2, then repeat Step 2.
4. Return N, m, and $C_{\min}$.

This algorithm also finds only a local optimum with respect to these changes, and with respect to the initial choice of N in Step 1; again, different choices may lead to different results. Our experiments took an initial choice for N such that $z \leq N_1 \leq \cdots \leq N_B \leq z+1$ for some $z \in \mathbb{Z}$. One can also omit one or more large primes ℓ_j by taking each $N_j = 1$ and $m_j = 0$.

Within the full two-layer greedy algorithm, each N considered at the upper layer involves $B(B-1)$ calls to the lower layer, the optimization of 1-norm bounds. Recall that each call to the lower layer involves at least $B(B-1)$ evaluations of the cost function. Overall there are nearly B^4 evaluations of the cost function.

Numerical examples. Table 1 shows examples of outputs of the above search. For each B, the "N"/"m" column shows the final (N, m) found, and the "cost" column shows the cost function for that (N, m), to two digits after the decimal point.

We used a server with two 64-core AMD EPYC 7742 CPUs, but limited each search to 32 cores running in parallel. We parallelized only the upper layer of the search; often fewer than 32 cores were used since some calls to the lower layer were slower than others. For each B, "wall" shows the seconds of real time used for the search, and "CPU" shows the total seconds of CPU time (across all cores, user time plus system time) used for the search.

7 Constant-time software for the action

We have built a self-contained high-performance software package, `high-ctidh`, that includes implementations of all of the operations needed by CSIDH users for whichever parameter set is selected: constant-time key generation, constant-time computation of the CSIDH action, and validation of (claimed) public keys. The package uses the CTIDH key space and CTIDH algorithms to set new cycle-count records for constant-time CSIDH.

The `high-ctidh` source code is in C, with assembly language for field arithmetic. Beyond the performance benefits, using low-level languages is helpful for ensuring constant-time behavior, as explained below. Measuring the performance of a full C implementation

Table 1: Results of searches, for various choices of B, for CTIDH parameters with at least 2^{256} keys for the CSIDH-512 prime. See text for description.

B	wall CPU	cost	N m
1	1.27 1.25	3462230.00	74 153
2	1.55 1.80	1483388.79	36 38 64 96
3	2.59 4.78	990766.14	23 27 24 40 62 62
4	5.14 22.17	755266.87	14 19 20 21 29 45 46 46
5	4.65 22.15	649002.35	13 15 15 17 14 23 37 38 38 35
6	13.29 150.29	583256.02	10 11 12 12 15 14 19 31 31 32 32 30
7	24.98 334.31	537496.27	7 10 10 12 12 14 9 17 27 28 28 28 28 22
8	65.90 1141.82	504984.23	5 9 9 10 10 11 11 9 15 24 25 25 25 26 26 16
9	138.65 2763.28	485052.29	5 7 8 8 8 7 10 12 9 14 22 23 23 23 23 24 24 13
10	393.63 8209.63	471184.70	5 7 8 8 8 7 9 10 11 1 13 20 22 22 22 22 22 22 22 1
11	966.91 21740.60	451105.76	3 5 6 7 7 8 7 9 10 11 1 11 18 19 20 20 21 20 21 21 21 1
12	1484.31 36763.23	448573.04	3 4 6 6 6 6 7 7 8 10 10 1 10 16 18 18 19 19 19 19 19 19 19 2
13	2301.94 55252.51	445054.10	3 4 4 6 6 6 7 7 8 7 7 8 1 10 16 17 18 18 18 18 18 18 18 17 14 1
14	6509 161371.00	437985.55	2 3 4 4 5 5 6 7 7 8 8 6 8 1 10 14 16 17 17 17 18 18 18 18 18 13 13 1
15	8341 211336.80	440201.56	3 4 3 4 4 5 5 5 6 6 6 7 7 8 1 9 14 15 15 16 16 16 16 16 16 16 16 15 13 1
16	18060 491547.34	442718.29	2 3 4 4 5 5 5 5 6 6 7 8 4 1 8 1 9 13 15 16 16 17 17 17 17 16 17 17 7 1 16 1
17	29733 808639.64	450343.88	2 3 4 4 5 5 5 5 5 5 5 5 7 5 3 5 1 8 12 14 15 15 15 16 15 16 16 14 13 16 11 6 10 1
18	73925 2012125.98	443412.54	2 2 3 3 4 4 5 5 5 5 5 6 6 6 3 1 8 1 8 11 13 14 14 15 15 15 15 15 15 15 14 14 7 2 15 1
19	103825 2961123.56	447506.32	2 2 3 3 3 4 4 5 5 5 6 4 6 1 8 4 7 1 1 8 12 14 15 15 15 15 16 16 16 16 10 16 2 16 7 14 1 1
20	167114 4794006.52	455328.80	2 2 3 3 4 4 5 5 5 5 5 4 7 7 1 3 6 1 1 1 9 12 14 14 16 16 16 16 16 16 16 11 16 16 2 5 12 1 1 1
21	278646 7981372.99	460901.00	2 2 3 3 4 4 5 5 5 5 5 7 2 1 1 3 1 7 7 1 1 9 13 15 16 16 16 16 17 17 17 16 17 5 2 2 6 1 17 11 1 1

also resolves the concerns raised by using multiplications as a predictor of performance, such as concerns that some subroutines could be difficult to handle in constant time and that improved multiplication counts could be outweighed by overhead.

The software is freely available online. This section describes the software. Section 8 reports the software speeds and compares to previous speeds.

7.1 Processor selection and field arithmetic

The original CSIDH paper reported clock cycles for variable-time CSIDH-512 software on an Intel Skylake CPU core. Skylake is also the most common CPU choice in followup papers on CSIDH software speed. We similarly focus on Skylake to maximize comparability.

The original `csidh-20180826` software from [10] included a small assembly-language library for Intel chips (Broadwell and newer) to perform arithmetic modulo the CSIDH-512 prime. The same library has been copied, with minor tweaks and generalizations to other primes, into various subsequent software packages, including `high-ctidh`. Code above the field-arithmetic level, decomposing isogenies into multiplications etc., are written in C, so porting the software to another CPU is mainly a matter of writing an efficient Montgomery multiplier for that CPU. Beware that each CPU will have different cycle counts, and possibly a different ranking of algorithmic choices.

The `velusqrt-asm` software from [3] includes an adaptation of the same library to CSIDH-1024. The `sqale-csidh-velusqrt` software from [12] includes adaptations to larger sizes, all automatically generated by a code generator that takes p as input. The `high-ctidh` package includes a similar code generator, with some small improvements in the details: for example, we use less arithmetic for conditional subtraction, and we avoid `cmov` instructions with memory operands out of concern that they could have data-dependent timings.

7.2 Computing one isogeny

The middle layer of `high-ctidh` computes an ℓ-isogeny for one prime ℓ; it also includes auxiliary functions such as multiplying by the scalar ℓ. We built this layer as follows.

We started with the `xISOG` function in `velusqrt-asm`. As in `csidh-20180826`, this function takes a curve and a point P of order ℓ, and returns the corresponding ℓ-isogenous curve. It also takes a point T, and returns the image of that point under the isogeny.

We extended the function interface to take lower and upper bounds on ℓ—the smallest and largest prime in the batch containing ℓ—and we modified the software to take time depending only on these bounds, not on the secret ℓ. The Matryoshka-doll structure of the computation (see Section 5.2.2) meant that very little code had to change. Each loop to ℓ is replaced by a loop to the upper bound, with constant-time conditional selection of the results relevant to ℓ; and ℓ is replaced by the lower bound as input to the $\sqrt{\text{élu}}$ parameter selection. An upper bound was used the same way in [5]; the use of the lower bound for a Matryoshka-doll $\sqrt{\text{élu}}$ is new here.

We reused the automatic $\sqrt{\text{élu}}$ parameter-tuning mechanisms from `velusqrt-asm`. These mechanisms offer the option of tuning for multiplication counts or tuning for cycles. Since most CSIDH-related papers report multiplication counts while fewer report cycles, we chose to tune for multiplication counts for comparability, but this makes only a small difference: cycle counts and multiplication counts are highly correlated.

We made more changes to incorporate known optimizations, including an observation from [5] regarding the applicability of multiexponentiation, and an observation from [1] regarding reciprocal polynomials. Computing a 587-isogeny and pushing a point through takes 2108 multiplications in this software (counting squarings as multiplications); for comparison, [1] took 3.4% more, and `velusqrt-asm` took 8.9% more.

More importantly for the high-level algorithms, we extended the interface to allow an array of points T to be pushed through the isogeny—e.g., two or zero points rather than one. We also incorporated shorter differential addition chains, as in [11], for scalar multiplications, and standard addition chains for the constant-time exponentiation inside Legendre-symbol computation.

There would be marginal speedups from tuning the $\sqrt{\text{élu}}$ parameters separately for each number of points. Taking parameters $(6, 3)$ for 0 points instead of $(0, 0)$ saves 2 out of 328 multiplications for $\ell = 79$; 2 out of 344 multiplications for $\ell = 83$; and 8 out of 368 multiplications for $\ell = 89$. Parameter adjustments also save 3 multiplications for 0 points for each $\ell \in \{557, 587, 613\}$. However, we did not find such speedups for most primes, and we did not find such speedups for the much more common case of 2 points.

7.3 Computing the action

The top layer of `high-ctidh` is new, and includes the core CTIDH algorithms described earlier in this paper. The key space is $\mathcal{K}_{N,m}$, allowing any vector with 1-norm at most m_1 for the first N_1 primes, 1-norm at most m_2 for the first N_2 primes, etc. Constant-time generation of a length-N_i vector of 1-norm at most m_i works as follows:

- Generate $N_i + m_i$ uniform random b-bit integers.
- Set the bottom bit of each of the first N_i integers, and clear the bottom bit of each of the last m_i integers.
- Sort the integers. (We reused existing constant-time sorting software from [2].)
- If any adjacent integers are the same outside the bottom bit, start over. (Otherwise the integers were distinct outside the bottom bit, so sorting them applies a uniform random permutation.)
- Extract the bottom bit at each position. (This is a uniform random bit string of length $N_i + m_i$ with exactly N_i bits set.)
- Consider the entries as integers. Add the first entry to the second, then add the resulting second entry to the third, etc. (Now there are maybe some 0s, then at least one 1, then at least one 2, and so on through at least one N_i.)
- Count, in constant time, the number e_0 of 0, the number e_1 of 1, and so on through the number e_{N_i-1} of $N_i - 1$. (These tallies add up to at most $N_i + m_i - 1$, since the number of N_i was not included. Each of $e_1, \ldots, e_{N_i-1}$ is positive, and e_0 is nonnegative.)
- Subtract 1 from each of $e_1, \ldots, e_{N_i-1}$. (Now $e_0, \ldots, e_{N_i-1}$ is a uniform random string of N_i nonnegative integers with sum at most m_i.)
- Generate a uniform random N_i-bit string $s_0, \ldots, s_{N_i-1}$.
- Compute, in constant time, whether any j has $s_j = 1$ and $e_j = 0$. If so, start over.
- Replace each e_j with $-e_j$ if $s_j = 1$.

As required by the constant-time property, the two rejection steps in this algorithm are independent of the secrets produced as output. The first rejection step is very unlikely to occur when b is chosen so that 2^b is on a larger scale than $(N_i + m_i)^2$. The second rejection step occurs more frequently. Sign variations for vectors of Hamming weight k contribute 2^k by Lemma 1 and thus the rejection correctly happens more frequently for smaller k.

In the case $N_i > m_i$, `high-ctidh` saves time by skipping (in constant time) the $s_j = 1$ rejection test for the first $N_i - m_i$ values of j having $e_j = 0$. There are always at least $N_i - m_i$ such values of j. This increases each acceptance chance by a factor $2^{N_i - m_i}$, preserving uniformity of the final output.

Once a private key is generated, the action is computed by a series of restricted square-free ABs. As in Section 4, the first AB handles one prime from each batch, the next AB handles one prime from each batch that might have something left to do, etc.

Within each AB, Elligator is used twice to generate two independent points; see Appendix B. Specifically, Elligator is used to generate a point on the first curve E_A: a point in $\tilde{E}_A(\mathbb{F}_p)$ if the first isogeny has negative sign, otherwise in $E_A(\mathbb{F}_p)$. This point is pushed through the first isogeny. Elligator is then used again to generate an independent point on the second curve $E_{A'}$: a point in $\tilde{E}_{A'}(\mathbb{F}_p)$ if the first isogeny had positive sign, otherwise in $E_{A'}(\mathbb{F}_p)$. Both choices are secret. These two points (T_0, T_1) are then pushed through subsequent isogenies as in Algorithm 4, except that no points are pushed through the last isogeny and only one point is pushed through the isogeny before that. The AB thus pushes $1, 2, 2, 2, \ldots, 2, 2, 2, 1, 0$ points through isogenies. The software permutes the $b \leq B$ batches in the AB to use primes $\ell_{b-1}, \ell_{b-3}, \ell_{b-4}, \ldots, \ell_1, \ell_{b-2}, \ell_b$ in that order.

Each AB selects one prime from each batch in the block and tries to compute an isogeny of total degree D, the product of the selected primes; $D = r'$ in Algorithm 4. Each point is multiplied by 4 and then by all primes outside D immediately after the point is generated by Elligator, so that the order of the point divides D. There are two types of primes outside D (compare Steps 2 and 3 of Algorithm 4):

- The batches in the AB are public. Primes outside these batches are publicly outside D.
- Primes that are inside the batches in the AB, but that are not the secretly selected prime per batch, are secretly outside D.

For scalar multiplication by a product of secret primes, [5] uses a Montgomery ladder, with the number of ladder steps determined by the maximum possible product. For public primes, [11] does better using a precomputed differential addition chain for each prime. Our `high-ctidh` software also uses these chains for secret primes, taking care to handle the incompleteness of differential-addition formulas and to do everything in constant time. The primes in a batch usually vary slightly in chain length, so the software always runs to the maximum length.

Each ℓ-isogeny then clears ℓ from the order of the point that was used to compute the isogeny. As in line 14 of Algorithm 4, the software multiplies the point by ℓ anyway (again using a constant-time differential addition chain), just in case this was a dummy isogeny, i.e., there was secretly nothing left to do in the batch. This extra scalar multiplication could be merged with the isogeny computation, but the $\sqrt{\text{élu}}$ structure seems to make this somewhat more complicated than in [16], and the extra scalar multiplication accounts for only about 3% of the CSIDH-512 computation. The other point is also multiplied by ℓ.

Recall that an AB successfully handling a batch is a public event, visible in timing: it means that a (real or dummy) ℓ-isogeny is computed now for some ℓ in the batch, publicly decreasing the maximum 1-norm of the batch. This event occurs with probability $1 - 1/\ell_{i,1}$, where $\ell_{i,1}$ is the smallest prime in the batch containing $\ell = \ell_{i,j}$. As in Section 5.2.1, the software creates this event exactly when there is a conjunction of a natural success and an artificial success. A natural success, probability $1 - 1/\ell$, means that cofactor multiplication produces a point of order ℓ rather than order 1. An artificial success, probability $\gamma = (1 - 1/\ell_{i,1})/(1 - 1/\ell)$, is determined by a γ-biased coin toss.

One obvious way to generate a γ-biased coin is to (1) generate a uniform random integer modulo $\ell_{i,1}(\ell - 1)$ and (2) compute whether the integer is smaller than $\ell(\ell_{i,1} - 1)$. The second step is easy to do in constant time. For the first step, the software generates a uniform random 256-bit integer and, in constant time, reduces that modulo $\ell_{i,1}(\ell - 1)$; the resulting distribution is indistinguishable from uniform. One could instead use rejection sampling to compute a uniform random integer modulo $\ell_{i,1}M$, where M is the least common multiple of $\ell - 1$ across primes ℓ in the batch, and then reduce the integer modulo $\ell_{i,1}(\ell - 1)$, to obtain an exactly uniform distribution; the reason to use M here rather than just one $\ell - 1$ is to avoid having the secret ℓ influence the rejection probability.

7.4 Automated constant-time verification

We designed and analyzed every step of the CTIDH algorithm to be constant time, leaking nothing about the input through timing; this is the basis for our claim that the algorithm is in fact constant time. We also designed and reviewed every new line of code in `high-ctidh` to be constant time, and reviewed every line of reused code for the same property; this is the basis for our claim that the software is in fact constant time. These analyses are complete—but, as in most papers on constant-time algorithms, are entirely done by hand, raising the question of what protections there are against human error.

For extra assurance, we designated an internal auditor to use automated tools to verify the constant-time claims. This subsection is an audit report to support external auditing. This report describes what the tools verified, describes various limitations of this verification, and describes various steps that the auditor took to compensate for those limitations.

From a risk-management perspective, a timing leak in `high-ctidh` would have to be at the intersection of (1) human error in this paper's manual analysis and (2) limitations of the automated verification. One might hope for automated verification without any limitations, eliminating the need for manual analysis (assuming correctness of the verification tools), but one risk identified below is beyond the current state of the art in automated verification. The automated verification is nevertheless useful in reducing risks overall.

An automated test using `valgrind`. There is a standard tool, `valgrind` [19], that runs a specified binary, watching each instruction for memory errors—in particular, branches and array indices derived from undefined data. It is not a new observation that if secret data in cryptographic software is marked as undefined then simply running `valgrind` will automatically check whether there is any data flow from secrets to branches and array indices.

Because `valgrind` works at the binary level, this analysis includes any optimizations that might have been introduced by the compiler. A compiler change could generate a different binary with timing leaks, but `valgrind` is fast enough to be systematically run on all compiled cryptographic software before the software is deployed.

The auditor wrote a simple `checkct` program using `high-ctidh` to perform a full CSIDH key exchange; this program is included in the `high-ctidh` package. For example, running `valgrind ./checkct512default` takes under 30 seconds on a 3GHz Skylake core, where `checkct512default` performs a full CSIDH-512 key exchange. The underlying `randombytes` function marks all of its output as undefined, so `valgrind` is checking for any possible data flow from randomness to branches or to array indices. For each size, `valgrind` completes successfully, indicating that there is no such data flow.

Limitations of the automated test, and steps to address the limitations. The following paragraphs ask, from the auditor's perspective, what could have been missed *by this automated test*—for example, the auditor asks what would happen if private keys were actually generated by OpenSSL's `RAND_bytes` rather than `randombytes`. Everything is covered by the paper's manual analysis—for example, we had already checked that all code was generating all randomness via `randombytes`—but the question addressed here is the level of *extra* assurance provided by the automated analysis.

If the code generates randomness such as private keys via `RAND_bytes` rather than `randombytes`, then private keys will not be marked as undefined, so `valgrind` will not track data flow from private keys to branches or to array indices. To address this, the auditor skimmed `high-ctidh` to check the (very limited) set of C library functions being used, and double-checked the list of functions output by `nm checkct512default`.

If private keys are actually deterministic then they will not be marked as undefined. To address this, the auditor added a step to `checkct` to mark Alice's private key as undefined

before Alice handles Bob's public key. The auditor also checked examples of private keys and saw them varying.

The `valgrind` analysis is dynamic, tracing through one run of code. Perhaps CSIDH-1024 triggers code paths that are not used by CSIDH-512 and that leak secret data. To address this, the auditor tried all sizes of interest.

Different runs could still follow different code paths because of the declassification described below. To address this, the auditor tried many runs of each size, but there is still a risk that all of the runs missed some code path. In theory it should be possible to combine `valgrind` with static code-coverage analysis for binaries that follow standard calling conventions, but as far as we know no tools are available for this. There are tools for static constant-time analysis of C code, and those tools could be applied to a modified version of `high-ctidh` that replaces the assembly-language portions with reference C code.

The `valgrind` analysis checks that all array indices and all branch conditions are defined, but does not check that division inputs are defined. Division instructions take variable time in most CPUs, and should *not* be modeled as taking constant time. To address this, the auditor skimmed the assembly-language code for any use of division instructions, and skimmed the C code for any operations likely to be compiled into division instructions. Patching `valgrind` to limit the set of acceptable instructions would reduce risks here.

Finally, the `high-ctidh` package has six `crypto_declassify` lines explicitly marking certain pieces of data as *defined*, meaning that `valgrind` allows branches and array indices derived from that data. Perhaps this declassification leaks secrets. This is the most important risk, the risk that would require advances in automated verification to address.

Five of the six lines are in rejection-sampling loops: one in generating uniform random integers modulo p (rejecting numbers $\geq p$), two in Elligator (rejecting random numbers $0, 1, -1$), and two in the subroutine described above to generate vectors of bounded 1-norm. The sixth, and the most worrisome, declassifies the success of a batch in an AB. Analyzing the safety of this declassification requires analyzing everything that influences the success probability, including

- the logic concluding that the natural failure probability of generating a curve point of order $\ell_{i,j}$ is exactly $1/\ell_{i,j}$,
- the coin toss artificially increasing the failure probability to exactly $1/\ell_{i,1}$, and
- the use of Elligator as a substitute for uniform random points, assuming (as previously conjectured; see Appendix B) indistinguishability of the point orders.

We again emphasize that all of this analysis is included in this paper. The challenge for the future is to *automate* the analysis.

8 Software speeds

This section reports various measurements of the `high-ctidh` software from Section 7, and compares the measurements to previous speeds for constant-time CSIDH.

8.1 Selecting a CSIDH size and collecting performance data

For comparability to previous speed reports, we focus here on CSIDH-512 with a key space of 2^{256} vectors. After some searching we took the (N, m) shown for $B = 14$ in Figure 1. This (N, m) has approximately $2^{256.009}$ keys. Our cost calculator claimed that this (N, m) would use approximately 437986 multiplications on average.

We chose parameters a and c, and performed a action computations for each of c different private keys on a 3GHz Intel Xeon E3-1220 v5 (Skylake) CPU with Turbo Boost disabled. This CPU does not support hyperthreading, and to limit noise we used only one

core. For each of the ac computations, we recorded a cycle count, a total multiplication count including squarings, a separate count of squarings, and a total addition count including subtractions. We also tracked, for each key and each batch of primes, the success probability of that batch in ABs for the computations for that key.

Choosing $c = 65$, as in [3], and $a = 16383$ meant that experiments completed quickly, half a day on one core. We did not detect any deviations from the null hypothesis that the software performance is independent of the private key. For example, as discussed below, the per-key success probability of the batch having smallest prime $\ell_{i,1}$ was not statistically distinguishable from $1 - 1/\ell_{i,1}$.

One can easily justify spending further computer time on experiments. Larger a or c would make the total statistics more robust. Larger a would make the per-key statistics more robust. Larger c would be useful if there were a set of, say, 1 in every 1000 keys that somehow leaked information through timing. On the other hand, predictable CTIDH implementation errors such as taking a coin with probability $1 - \gamma$ rather than γ would have been caught by our experiments.

8.2 Performance results for the selected CSIDH size

We use the standard notation $\mathbf{M}$ for multiplications not including squarings, $\mathbf{S}$ for squarings, and $\mathbf{a}$ for additions including subtractions. One common metric in the literature is $(\mathbf{M}, \mathbf{S}, \mathbf{a}) = (1, 1, 0)$, counting the total number of multiplications while ignoring the costs of addition and ignoring possible squaring speedups. Another common metric is $(\mathbf{M}, \mathbf{S}, \mathbf{a}) = (1, 0.8, 0.05)$.

Across all 1064895 experiments, the average cycle count was 125.53 million, standard deviation 3.01 million. The average $\mathbf{M}$ was 321207, standard deviation 6621. The average $\mathbf{S}$ was 116798, standard deviation 4336. The average $\mathbf{a}$ was 482311, standard deviation 9322. The average cost in the $(\mathbf{M}, \mathbf{S}, \mathbf{a}) = (1, 1, 0)$ metric was 438006. The average cost in the $(\mathbf{M}, \mathbf{S}, \mathbf{a}) = (1, 0.8, 0.05)$ metric was 438762.

For the first key in particular, the averages were 125.55 million, 321270, 116837, and 482399 respectively. The gaps between these per-key averages and the overall averages are +0.90%, +0.94%, +0.89%, +0.94%, respectively, of a standard deviation, which is unsurprising for 16383 experiments per key. The gaps for the next four keys are −0.61%, −0.58%, −0.56%, −0.59%, −0.06%, −0.02%, −0.02%, −0.02%, −1.44%, −1.39%, −1.37%, −1.39%, +0.09%, +0.20%, +0.19%, +0.20%, respectively, of a standard deviation. The per-batch success probability for the first key, divided by the expected $1 - 1/\ell_{i,1}$, was 0.999670 for the first batch, 1.000785 for the second batch, 0.999688 for the third batch, etc.; for the second key, 1.001005, 0.998374, 1.000442, etc.; for the third key, 0.999304, 1.000131, 0.999820, etc.; for the fourth key, 1.000476, 0.998714, 0.999224, etc.; for the fifth key, 1.001030, 1.001379, 1.000829, etc. The first-batch gaps from the predicted average (namely 1) for the first 10 keys are −0.05%, +0.14%, −0.10%, +0.07%, +0.15%, −0.35%, +0.13%, +0.01%, +0.12%, −0.21% of the predicted standard deviation (namely $\sqrt{1/2}$); note that each of the 16383 experiments involves around 20 first-batch tries.

To understand the performance results in more detail, we plotted the distribution of all ac multiplication counts as the red curve in Figure 1. We also computed, for each key, the distribution of the 16383 multiplication counts for that key; there are five blue curves in Figure 1, showing the minimum, first quartile, median, third quartile, and maximum of these 65 distributions. The green curves, with a larger spread, are like the blue curves but are limited to the last 255 multiplication counts for each key.

Each curve has a stair-step shape. Another step upwards reflects another AB in the computation, with (typically) two Elligator calls and two large scalar multiplications. Any number of ABs *can* appear—for example, $\ell = 3$ can fail again and again—but with exponentially low probability. One can extrapolate the budget needed for an application

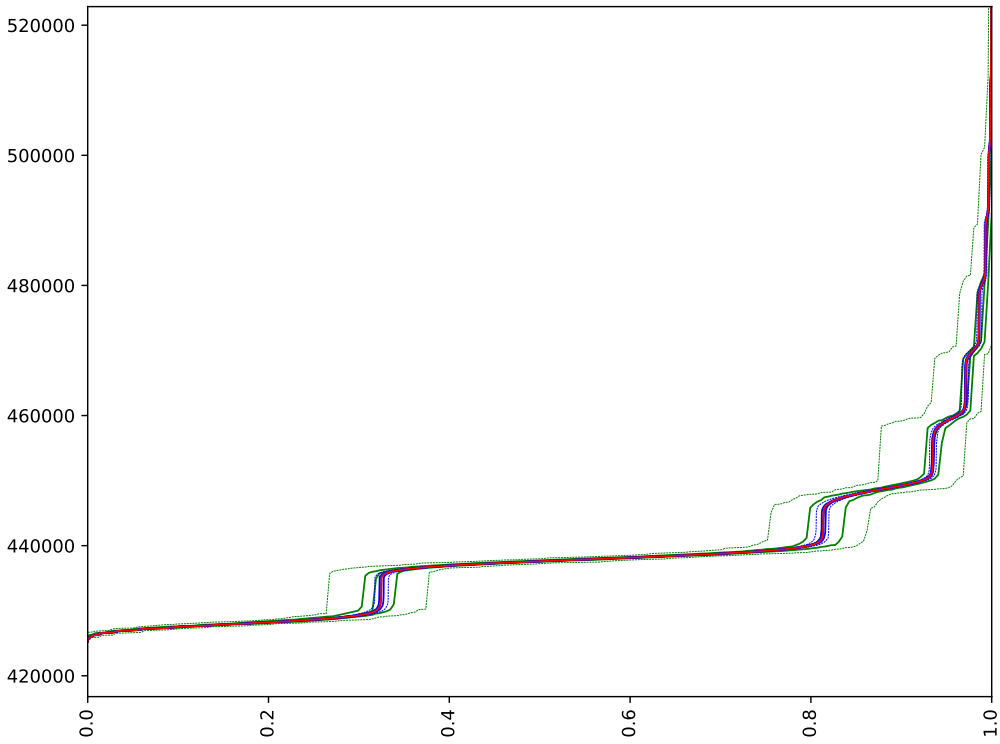

Figure 1: Distributions of costs in the $(\mathbf{M}, \mathbf{S}, \mathbf{a}) = (1, 1, 0)$ metric, i.e., total multiplication counts. Five green curves: minimum, first quartile, median, third quartile, and maximum of 65 per-key distributions of multiplication counts in the last 255 experiments. Five blue curves: minimum, first quartile, median, third quartile, and maximum of 65 per-key distributions of multiplication counts in all 16383 experiments. Red curve: distribution of multiplication counts in all experiments across all keys.

that needs to run for a fixed time (e.g., [5]) with a failure probability of, e.g., 2^{-100}; in this scenario the time would be lower if the smallest primes were left out of all batches.

It is unsurprising to see that the green curves have a spread of step positioning on the scale of 10%, given that these curves consider only 255 experiments for each of the 65 keys. Similar comments apply to the blue curves, with more experiments and a narrower spread. As a visual illustration that the spread is what one would expect, Figure 2 replaces the green and blue curves from Figure 1 with random simulations based on the red curve.

To gain more confidence that the distributions match, one could run more experiments for each key and watch for a further narrowing from the blue curves towards the red curve. Note that graphing the complete distributions provides much more information than computing a single number such as a t-statistic. Similar comments apply to cost metrics beyond multiplications: for example, Figure 3 shows cycle counts, and one could similarly plot other statistics such as the time of the first failed batch.

We also measured the cost of validation of a public key: median 14680 multiplications (after some easy speedups), around 4.09 million cycles. Note that validation takes variable time: an invalid key fails much more quickly. Finally, we measured the cost of generating a private key: typically under 1 million cycles, a negligible cost compared to generating the corresponding public key.

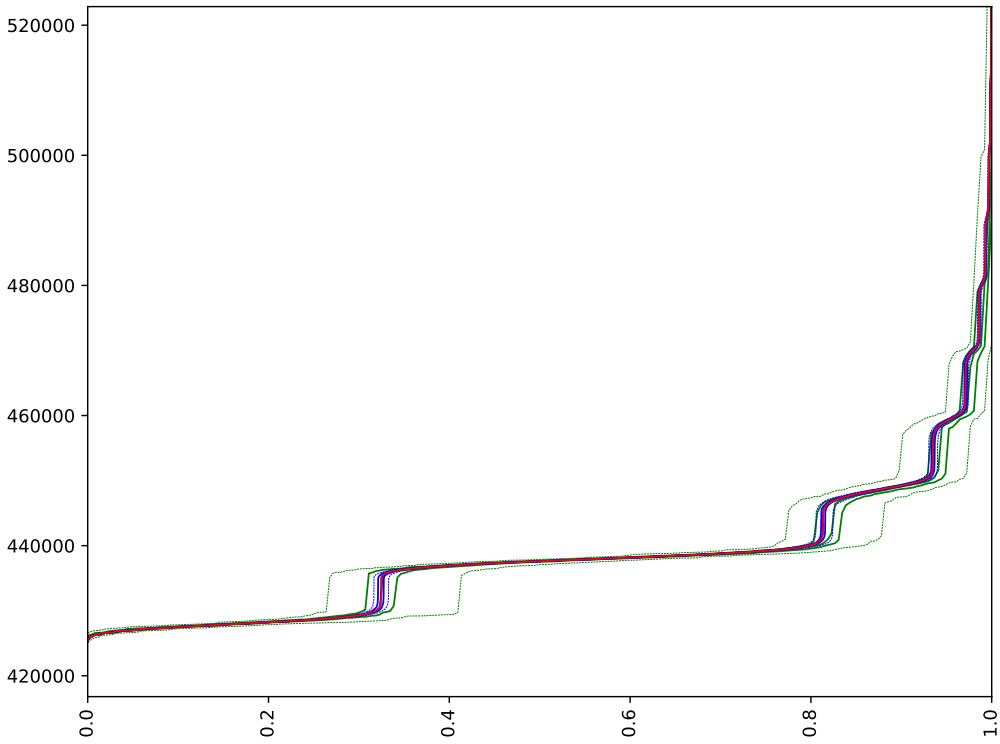

Figure 2: Simulation of Figure 1. The red curve is copied from Figure 1. The green and blue curves replace the underlying per-key data with random samples from the red curve.

8.3 Other CSIDH sizes

We also evaluated two further sizes, running $65 \cdot 16383$ experiments per size, to illustrate performance variations in two different dimensions of the CSIDH parameter space.

First, to understand the effect of having a larger list of ℓ, we switched from the CSIDH-512 prime to the CSIDH-1024 prime, while keeping the same size of key space. After some searching we took CTIDH batch sizes $2, 3, 5, 4, 6, 6, 6, 6, 6, 7, 7, 7, 6, 7, 7, 5, 6, 5, 10, 3, 10, 5, 1$, with bounds $2, 4, 5, 5, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 5, 5, 3, 6, 2, 6, 2, 0$. There are approximately $2^{256.066}$ keys.

Across all 1064895 experiments, the average cycle count was 469.52 million, standard deviation 15.29 million. The average $\mathbf{M}$ was 287739, standard deviation 7420. The average $\mathbf{S}$ was 87944, standard deviation 4961. The average $\mathbf{a}$ was 486764, standard deviation 10525. The average cost in the $(\mathbf{M}, \mathbf{S}, \mathbf{a}) = (1, 1, 0)$ metric was 375683. The average cost in the $(\mathbf{M}, \mathbf{S}, \mathbf{a}) = (1, 0.8, 0.05)$ metric was 382432.

As in [1, Tables 1 and 2], the CSIDH-1024 prime uses fewer multiplications than the CSIDH-512 prime (although, unsurprisingly, the larger prime makes each multiplication slower). One might think that a larger list of ℓ needs more multiplications, since one needs to clear more cofactors; but the larger list also means that one can use smaller exponents for the same size key space, and in our experiments this turns out to have a larger effect.

Second, to understand the effect of changing the size of the key space, we took CSIDH-512 with only 2^{220} keys, as in the software from [12]. Specifically, we took CTIDH batch sizes $2, 3, 4, 4, 5, 5, 5, 5, 5, 7, 7, 8, 7, 6, 1$, with bounds $6, 9, 11, 11, 12, 12, 12, 12, 12, 12, 12, 12, 8, 6, 1$. There are approximately $2^{220.004}$ keys. This option is labeled "511" in `high-ctidh`.

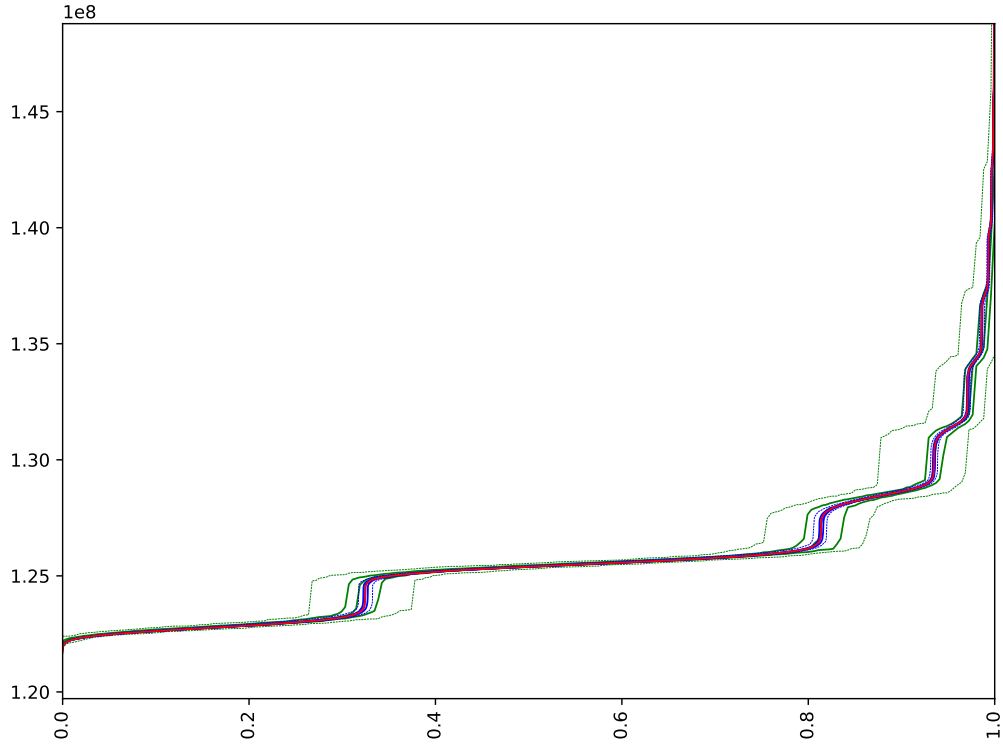

Figure 3: Same as Figure 1, but replacing multiplication counts with cycle counts scaled by 10^8.

Across all 1064895 experiments, the average cycle count was 89.11 million, standard deviation 2.37 million. The average $\mathbf{M}$ was 228780, standard deviation 5186. The average $\mathbf{S}$ was 82165, standard deviation 3428. The average $\mathbf{a}$ was 346798, standard deviation 7344. The average cost in the $(\mathbf{M}, \mathbf{S}, \mathbf{a}) = (1, 1, 0)$ metric was 310945. The average cost in the $(\mathbf{M}, \mathbf{S}, \mathbf{a}) = (1, 0.8, 0.05)$ metric was 311852.

8.4 Comparisons

There have been several previous speed reports [16, 20, 11, 15, 13, 1, 12] for constant-time CSIDH. CSIDH-512 with a key space of 2^{256} vectors is almost always included, and for this size the lowest multiplication count we have found in the literature is 789000: this is from [1, Table 1, "hvelu", "OAYT-style"], which reports $624000\mathbf{M} + 165000\mathbf{S} + 893000\mathbf{a}$. All Skylake cycle counts we have found are above 200 million. The `high-ctidh` speeds are much faster, and have the added feature of constant-time verification using `valgrind`.

Sometimes the latest software speeds are not fully reflected in the relevant papers, so we downloaded the latest versions of `csidh_withstrategies` (dated July 2020) from [13] and `sqale-csidh-velusqrt` (dated December 2020) from [12], and ran their benchmarking tools—with one modification to change the number of experiments ("`its`") from 1024 to 65536—on the same Skylake machine that we had used for measuring `high-ctidh`. Some care is required in comparisons, for at least three reasons: first, some tools report the time for an action *plus* key validation; second, different benchmarking frameworks could be measuring different things (e.g., our impression is that the costs of Elligator were omitted from the multiplication counts reported in [1]); third, the 512-bit parameters in

Table 2: Comparison of speed reports for constant-time CSIDH actions. The CSIDH size is specified by "pub" (512 for the CSIDH-512 prime, 1024 for the CSIDH-1024 prime) and "priv" (k where private keys are chosen from a space of approximately 2^k vectors). "DH" is the Diffie–Hellman stage: "1" for computing a public key (computing the CSIDH action), "2" for computing a shared secret (validating a public key and then computing the CSIDH action). "Mcyc" is millions of Skylake cycles (not shown for Python software); "**M**" is the number of multiplications not including squarings; "**S**" is the number of squarings; "**a**" is the number of additions including subtractions; "$1, 1, 0$" and "$1, 0.8, 0.05$" are combinations of **M**, **S**, and **a**. See text for measurement details and standard deviations.

pub	priv	DH	Mcyc	**M**	**S**	**a**	$1, 1, 0$	$1, 0.8, 0.05$	
512	220	1	89.11	228780	82165	346798	310945	311852	new
512	220	1	190.92	447000	128000	626000	575000	580700	[12]
512	220	2	93.23	238538	87154	361964	325692	326359	new
512	256	1	125.53	321207	116798	482311	438006	438762	new
512	256	1	—	624000	165000	893000	789000	800650	[1]
512	256	2	129.64	330966	121787	497476	452752	453269	new
512	256	2	218.42	665876	189377	691231	855253	851939	[13]
512	256	2	238.51	632444	209310	704576	841754	835121	[15]
512	256	2	239.00	657000	210000	691000	867000	859550	[11]
512	256	2	—	732966	243838	680801	976804	962076	[20]
512	256	2	395.00	1054000	410000	1053000	1464000	1434650	[16]
1024	256	1	469.52	287739	87944	486764	375683	382432	new
1024	256	1	—	552000	133000	924000	685000	704600	[1]
1024	256	2	511.19	310154	99371	521400	409525	415721	new

`sqale-csidh-velusqrt` use a key space of size only 2^{220}, as noted above. Note that for CSIDH-1024 there is even more variation in the literature in the size of key space; e.g., the original CSIDH-1024 software from [10] used $5^{130} > 2^{300}$ keys.

The `csidh_withstrategies` tools, using `BITLENGTH_OF_P=512 TYPE=WITHDUMMY_2 APPROACH=STRATEGY`, reported averages of 218.42 million clock cycles (standard deviation 3.39 million), 691231**a** (standard deviation 12554), 189377**S** (standard deviation 4450), and 665876**M** (standard deviation 7888); in other words, 855253 multiplications, or 851939 counting $(\mathbf{M}, \mathbf{S}, \mathbf{a}) = (1, 0.8, 0.05)$.

The `sqale-csidh-velusqrt` tools, using `BITS=512 STYLE=wd2`, reported averages of 190.921 million cycles (standard deviation 4.32 million), 626000**a** (standard deviation 13000), 128000**S** (standard deviation 5000), and 447000**M** (standard deviation 9000); i.e., 575000 multiplications. For comparison, `high-ctidh` takes 89.11 million cycles (310945 multiplications) as noted above, plus 4.09 million cycles for validation.

Finally, Table 2 summarizes the measurements listed above for `high-ctidh`, for the software from [13], and for the software from [12]; the measurements stated in [20, 11, 15, 1] for the software in those papers; and the measurements stated in [11] for the software in [16]. For [15] the reported processor is an Intel Core i7-7500k, which is Kaby Lake rather than Skylake, but Kaby Lake cycle counts generally match Skylake cycle counts. The table omits cycle counts for [1], which used Python, and [20], which used C but had measurements affected by an unknown amount of Turbo Boost.

References

[1] Gora Adj, Jesús-Javier Chi-Domínguez, and Francisco Rodríguez-Henríquez. On new Vélu's formulae and their applications to CSIDH and B-SIDH constant-time implementations, 2020. https://eprint.iacr.org/2020/1109.

[2] Daniel J. Bernstein. djbsort, 2018. https://sorting.cr.yp.to.

[3] Daniel J. Bernstein, Luca De Feo, Antonin Leroux, and Benjamin Smith. Faster computation of isogenies of large prime degree. In Steven D. Galbraith, editor, *Proceedings of the Fourteenth Algorithmic Number Theory Symposium*, pages 39–55. Mathematics Sciences Publishers, 2020. https://eprint.iacr.org/2020/341.

[4] Daniel J. Bernstein, Mike Hamburg, Anna Krasnova, and Tanja Lange. Elligator: elliptic-curve points indistinguishable from uniform random strings. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*, pages 967–980. ACM, 2013. https://eprint.iacr.org/2013/325.

[5] Daniel J. Bernstein, Tanja Lange, Chloe Martindale, and Lorenz Panny. Quantum circuits for the CSIDH: optimizing quantum evaluation of isogenies. In Yuval Ishai and Vincent Rijmen, editors, *Advances in Cryptology - EUROCRYPT 2019 - 38th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Darmstadt, Germany, May 19-23, 2019, Proceedings, Part II*, volume 11477 of *Lecture Notes in Computer Science*, pages 409–441. Springer, 2019. https://eprint.iacr.org/2018/1059.

[6] Daniel J. Bernstein and Bo-Yin Yang. Fast constant-time gcd computation and modular inversion. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2019(3):340–398, 2019. https://eprint.iacr.org/2019/266.

[7] Xavier Bonnetain and André Schrottenloher. Quantum security analysis of CSIDH. In Canteaut and Ishai [9], pages 493–522. https://eprint.iacr.org/2018/537.

[8] Fabio Campos, Matthias J. Kannwischer, Michael Meyer, Hiroshi Onuki, and Marc Stöttinger. Trouble at the CSIDH: protecting CSIDH with dummy-operations against fault injection attacks. In *17th Workshop on Fault Detection and Tolerance in Cryptography, FDTC 2020, Milan, Italy, September 13, 2020*, pages 57–65. IEEE, 2020. https://eprint.iacr.org/2020/1005.

[9] Anne Canteaut and Yuval Ishai, editors. *Advances in Cryptology - EUROCRYPT 2020 - 39th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, May 10-14, 2020, Proceedings, Part II*, volume 12106 of *Lecture Notes in Computer Science*. Springer, 2020.

[10] Wouter Castryck, Tanja Lange, Chloe Martindale, Lorenz Panny, and Joost Renes. CSIDH: an efficient post-quantum commutative group action. In Thomas Peyrin and Steven D. Galbraith, editors, *Advances in Cryptology - ASIACRYPT 2018 - 24th International Conference on the Theory and Application of Cryptology and Information Security, Brisbane, QLD, Australia, December 2-6, 2018, Proceedings, Part III*, volume 11274 of *Lecture Notes in Computer Science*, pages 395–427. Springer, 2018. https://eprint.iacr.org/2018/383.

[11] Daniel Cervantes-Vázquez, Mathilde Chenu, Jesús-Javier Chi-Domínguez, Luca De Feo, Francisco Rodríguez-Henríquez, and Benjamin Smith. Stronger and faster side-channel protections for CSIDH. In Peter Schwabe and Nicolas Thériault, editors, *Progress in Cryptology - LATINCRYPT 2019 - 6th International Conference on Cryptology and Information Security in Latin America, Santiago de Chile, Chile, October 2-4, 2019, Proceedings*, volume 11774 of *Lecture Notes in Computer Science*, pages 173–193. Springer, 2019. https://eprint.iacr.org/2019/837.

[12] Jorge Chávez-Saab, Jesús-Javier Chi-Domínguez, Samuel Jaques, and Francisco Rodríguez-Henríquez. The SQALE of CSIDH: square-root Vélu quantum-resistant isogeny action with low exponents, 2020. https://eprint.iacr.org/2020/1520.

[13] Jesús-Javier Chi-Domínguez and Francisco Rodríguez-Henríquez. Optimal strategies for CSIDH, 2020. https://eprint.iacr.org/2020/417.

[14] Luca De Feo, David Jao, and Jérôme Plût. Towards quantum-resistant cryptosystems from supersingular elliptic curve isogenies. *J. Math. Cryptol.*, 8(3):209–247, 2014. https://eprint.iacr.org/2011/506.

[15] Aaron Hutchinson, Jason T. LeGrow, Brian Koziel, and Reza Azarderakhsh. Further optimizations of CSIDH: A systematic approach to efficient strategies, permutations, and bound vectors. In Mauro Conti, Jianying Zhou, Emiliano Casalicchio, and Angelo Spognardi, editors, *Applied Cryptography and Network Security - 18th International Conference, ACNS 2020, Rome, Italy, October 19-22, 2020, Proceedings, Part I*, volume 12146 of *Lecture Notes in Computer Science*, pages 481–501. Springer, 2020. https://eprint.iacr.org/2019/1121.

[16] Michael Meyer, Fabio Campos, and Steffen Reith. On Lions and Elligators: An efficient constant-time implementation of CSIDH. In Jintai Ding and Rainer Steinwandt, editors, *Post-Quantum Cryptography - 10th International Conference, PQCrypto 2019, Chongqing, China, May 8-10, 2019 Revised Selected Papers*, volume 11505 of *Lecture Notes in Computer Science*, pages 307–325. Springer, 2019. https://eprint.iacr.org/2018/1198.

[17] Michael Meyer and Steffen Reith. A faster way to the CSIDH. In Debrup Chakraborty and Tetsu Iwata, editors, *Progress in Cryptology - INDOCRYPT 2018 - 19th International Conference on Cryptology in India, New Delhi, India, December 9-12, 2018, Proceedings*, volume 11356 of *Lecture Notes in Computer Science*, pages 137–152. Springer, 2018. https://eprint.iacr.org/2018/782.

[18] Kohei Nakagawa, Hiroshi Onuki, Atsushi Takayasu, and Tsuyoshi Takagi. L_1-norm ball for CSIDH: Optimal strategy for choosing the secret key space, 2020. https://eprint.iacr.org/2020/181.

[19] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In Jeanne Ferrante and Kathryn S. McKinley, editors, *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10–13, 2007*, pages 89–100. ACM, 2007. https://www.valgrind.org/docs/valgrind2007.pdf.

[20] Hiroshi Onuki, Yusuke Aikawa, Tsutomu Yamazaki, and Tsuyoshi Takagi. (Short paper) A faster constant-time algorithm of CSIDH keeping two points. In Nuttapong Attrapadung and Takeshi Yagi, editors, *Advances in Information and Computer Security - 14th International Workshop on Security, IWSEC 2019, Tokyo, Japan, August 28-30, 2019, Proceedings*, volume 11689 of *Lecture Notes in Computer Science*, pages 23–33. Springer, 2019. https://eprint.iacr.org/2019/353.

[21] Chris Peikert. He gives c-sieves on the CSIDH. In Canteaut and Ishai [9], pages 463–492. https://eprint.iacr.org/2019/725.

[22] Jacques Vélu. Isogénies entre courbes elliptiques. *Comptes Rendus de l'Académie des Sciences de Paris*, 273:238–241, 1971. https://gallica.bnf.fr/ark:/12148/cb34416987n/date.

A Dummy-free algorithms

The main body of this paper focuses on protecting against timing attacks. This appendix considers the extra challenge of protecting against faults.

Fault-injection attacks on constant-time CSIDH implementations are discussed in [11, 8]. Dummy operations are dangerous in this context: a "safe-error attack" faults an operation and, if the output is unchanged, concludes that the operation was a dummy operation. The literature thus aims for "dummy-free" algorithms as a step towards protecting against faults. A dummy-free constant-time group-action algorithm, based on the 2-point approach of [20], was proposed in [11]. This algorithm uses the modified key space

$$\tilde{\mathcal{K}}_m := \prod_{i=1}^{n} \{-m_i, -m_i+2, \ldots, m_i-2, m_i\} \qquad \text{with} \qquad \#\tilde{\mathcal{K}}_m = \prod_{i=1}^{n} (m_i+1)\,.$$

The action evaluation computes e_i isogenies of degree ℓ_i as usual, followed by $(m_i - |e_i|)/2$ isogenies in both the positive *and* negative directions. These isogenies effectively cancel each other, and we obtain the same resulting curve as when computing $|e_i|$ isogenies and $m_i - |e_i|$ dummy isogenies. This requires a total of m_i isogenies per degree, as in [20], but m_i has to be chosen twice as large for the same size of key space, so overall the dummy elimination costs a factor 2.

Algorithm 5 presents the dummy-free group action from [11] in terms of dummy-free ABs—which are just the usual squarefree ABs from Section 5, but with $R = \{-1, 1\}$ (so that no dummy isogenies are computed). The similarity to Algorithm 1 is clear.

Algorithm 5: The dummy-free constant-time group action evaluation from [11].

Parameters: $m = (m_1, \ldots, m_n)$
Input: $A \in \mathcal{M}$, $e = (e_1, \ldots, e_n) \in \tilde{\mathcal{K}}_m$
Output: A' with $E_{A'} = (\prod_i \mathfrak{l}_i^{e_i}) \star E_A$

$(\mu_1, \ldots, \mu_n) \leftarrow (m_1, \ldots, m_n)$
while $(\mu_1, \ldots, \mu_n) \neq (0, \ldots, 0)$ **do**
  Let $I = (I_1, \ldots, I_k)$ s.t. $I_1 < \cdots < I_k$ and $\{I_1, \ldots, I_k\} = \{1 \le i \le n \mid \mu_i > 0\}$
  Choose $e^+ \in \mathbb{Z}^n_{\ge 0}$ and $e^- \in \mathbb{Z}^n_{\le 0}$ such that $e_i^+ + e_i^- = e_i$ and $|e_i^+| + |e_i^-| = m_i$ for $1 \le i \le n$
  for $1 \le i \le k$ **do**
    $\epsilon_i \leftarrow \begin{cases} 1 & \text{if } e^+_{I_i} \neq 0 \\ -1 & \text{if } e^+_{I_1} = 0 \end{cases}$
  $(A, f) \leftarrow \alpha_{R,I}(A, (\epsilon_1, \ldots, \epsilon_k))$ `// Square-free AB`
  for $1 \le i \le k$ **do**
    $\mu_{I_i} \leftarrow \mu_{I_i} - f_i$
    if $\epsilon_i = 1$ **then**
      $e^+_{I_i} \leftarrow e^+_{I_i} - \epsilon_i \cdot f_i$
    else
      $e^-_{I_i} \leftarrow e^-_{I_i} - \epsilon_i \cdot f_i$
return A

CTIDH adapts easily to a dummy-free variant. Algorithm 6, a generalization of Algorithm 5 for $\tilde{\mathcal{K}}_m$, uses restricted square-free ABs with $R = \{-1, 1\}$ to handle keys in

$$\tilde{\mathcal{K}}_{N,m} := \left\{(e_1, \ldots, e_n) \in \mathcal{K}_{N,m} \mid \textstyle\sum_{j=1}^{N_i} |e_{i,j}| \equiv m_i \pmod 2 \text{ for all } i\right\},$$

a batching-oriented generalization of $\tilde{\mathcal{K}}_m$. We have $\#\tilde{\mathcal{K}}_{N,m} = \prod_{i=1}^{B} \tilde{\Phi}(N_i, m_i)$, where $\tilde{\Phi}$ sums $\Phi(N_i, j) - \Phi(N_i, j-1)$ for $j = m_i$, $j = m_i - 2$, etc., analogously to Lemma 1.

Algorithm 6: A constant-time group action for keys in $\tilde{\mathcal{K}}_{N,m}$ based on restricted squarefree ABs with $R = \{-1, 1\}$.

Parameters: N, m, B
Input: $A \in \mathcal{M}$, $e = (e_1, \ldots, e_n) \in \tilde{\mathcal{K}}_{N,m}$
Output: A' with $E_{A'} = (\prod_i \mathfrak{l}_i^{e_i}) \star E_A$

$(\mu_1, \ldots, \mu_B) \leftarrow (m_1, \ldots, m_B)$
Choose $e^+ \in \mathbb{Z}^n_{\geq 0}$ and $e^- \in \mathbb{Z}^n_{\leq 0}$ s.t. $e_i^+ + e_i^- = e_i$ and $\sum_{j=1}^{N_i}(|e_{i,j}^+| + |e_{i,j}^-|) = m_i$ for $1 \leq i \leq n$
while $(\mu_1, \ldots, \mu_B) \neq (0, \ldots, 0)$ **do**
  Let $I = (I_1, \ldots, I_k)$ s.t. $I_1 < \cdots < I_k$ and $\{I_1, \ldots, I_k\} = \{1 \leq i \leq B \mid \mu_i > 0\}$
  for $1 \leq i \leq k$ **do**
    Choose J_i such that $e^+_{I_i,J_i} \neq 0$ or $e^-_{I_i,J_i} \neq 0$
    $\epsilon_i \leftarrow \begin{cases} 1 & \text{if } e^+_{I_i,J_i} \neq 0 \\ -1 & \text{if } e^+_{I_i,J_i} = 0 \end{cases}$
  $(A, f) \leftarrow \beta_{R,I}(A, (\epsilon_1, \ldots, \epsilon_k), J)$ `// Restricted square-free AB`
  for $1 \leq i \leq k$ **do**
    $\mu_{I_i} \leftarrow \mu_{I_i} - f_i$
    if $\epsilon_i = 1$ **then**
      $e^+_{I_i,J_i} \leftarrow e^+_{I_i,J_i} - \epsilon_i \cdot f_i$
    else
      $e^-_{I_i,J_i} \leftarrow e^-_{I_i,J_i} - \epsilon_i \cdot f_i$
return A

Batching improves dummy-free operation counts even more than it improves constant-time operation counts. However, various subroutines inside ABs need to be redone to avoid lower-level dummy operations or to double-check, preferably at low cost, that the operations are being performed correctly. For example, the constant-time differential addition chains in our software involve dummy differential additions; it should be possible to avoid these by precomputing chains of the same length for all of the primes in a batch. As another example, the Matryoshka-doll structure involves dummy operations, and it would be interesting to explore adaptations of the countermeasures of [8] to this context.

B Elligator safety

The literature on algorithms for the CSIDH action frequently uses Elligator outputs as cheaper replacements for the uniform random points generated in these algorithms. This appendix analyzes the question of whether this is secure. The conclusion, in a nutshell, is that it seems reasonable to conjecture indistinguishability of the orders of Elligator outputs for large p from the orders of uniform random points.

The cost of UniformRandomPoints. The obvious way to generate a point in $E_A(\mathbb{F}_p)$ is to generate a uniform random $x \in \mathbb{F}_p$ and compute $y = \pm\sqrt{x^3 + Ax^2 + x}$, trying again if $x^3 + Ax^2 + x$ is not a square. The distribution is not exactly uniform, but one can easily adjust the procedure to correct this (see [5, Section 4.1]), or simply accept the distribution as being statistically indistinguishable from uniform.

The standard way to try to compute a square root, given that $p \equiv 3 \pmod 4$, is to compute a $(p+1)/4$ power. One more squaring then reveals whether the input was a square. Generating a point in $E_A(\mathbb{F}_p)$ in this way takes two exponentiations on average.

Before trying to compute y one can check the Legendre symbol $\left(\frac{x^3+Ax^2+x}{p}\right)$. The square-root attempt will succeed if and only if the symbol is not -1. This reduces two exponentiations to two Legendre-symbol computations and one exponentiation, saving time if a Legendre-symbol computation is more than twice as fast as an exponentiation.

Similar comments apply to $\tilde{E}_A(\mathbb{F}_p)$, producing an average `UniformRandomPoints` cost of four exponentiations, or four Legendre-symbol computations and two exponentiations. One can easily reduce the cost to three exponentiations, or two Legendre-symbol computations and two exponentations, by taking the first x as generating a point in $E_A(\mathbb{F}_p)$ in half of the cases and generating a point in $\tilde{E}_A(\mathbb{F}_p)$ in the other half of the cases.

Conventional algorithms for Montgomery-curve computations, including the isogeny computations needed in CSIDH, work only with x and do not need to inspect y. One can thus reduce the cost of `UniformRandomPoints` to three Legendre-symbol computations on average.

Our `high-ctidh` software follows previous CSIDH work in computing a Legendre symbol as a $(p-1)/2$ power, so the speed is the same as computing a square root, but it would be interesting to investigate faster algorithms. One can use blinding to guarantee constant-time Legendre-symbol computation:

- If $x^3 + Ax^2 + x$ is 0, set a bit indicating this, and replace the 0 with 1.
- Multiply by $\pm r^2$ where r is a uniform random nonzero element of $\mathbb{F}_p$.
- Use any Legendre-symbol algorithm.
- Adjust the output according to the 0 bit and the $\pm$ bit.

It would also be interesting to investigate whether the techniques of [6] can be adapted to this context, avoiding the costs of blinding.

Elligators everywhere. The literature generally takes a different approach, using the Elligator 2 [4] map. This approach has the advantage of using just one Legendre-symbol computation to generate a point in $E_A(\mathbb{F}_p)$ and, with no extra cost, a point in $\tilde{E}_A(\mathbb{F}_p)$. The disadvantage is that each point produced is distinguishable from uniform, covering only $(p-3)/2$ out of the $p+1$ possible points. Perhaps the *orders* of these points are distinguishable from the orders of uniform random points.

Elligator was first used in the CSIDH context in [5], which analyzed algorithms to compute CSIDH in superposition as a subroutine inside quantum attacks. That paper mentioned experiments suggesting that Elligator outputs have "failure chance almost exactly $1/\ell$" and that the higher-level algorithms in [5] performed as predicted. However, the security question for constructive CSIDH applications, namely the order-indistinguishability question, did not arise in [5]. A measurable deviation in orders could easily have avoided detection by the experiments in [5].

Elligator was first used for constructive CSIDH applications in [16] to generate an element of $E_A(\mathbb{F}_p)$. It was then used in [20] to generate an element of $E_A(\mathbb{F}_p) \times \tilde{E}_A(\mathbb{F}_p)$. Subsequent CSIDH software has also used Elligator. It is conceivable, however, that information about A is leaked via the distribution of orders of the points that are generated by Elligator.

Elligator tracking. To directly address the order-distinguishability question, we collected complete data for various small primes p. Specifically, for each $k \in \{1,2,3,4,5\}$, we took the smallest prime $p \equiv 3 \pmod 8$ for which $(p+1)/4$ factors into exactly k distinct primes.

For each p, we enumerated all $A \in \mathcal{M}$. For each (p, A), we enumerated all Elligator outputs $T_0 \in E_A(\mathbb{F}_p)$ and computed the exact distribution of the order of $[4]T_0$. We then

compared this to the uniform model: the exact distribution of orders for uniform random elements of $\mathbb{Z}/((p+1)/4)\mathbb{Z}$. Specifically, we computed the total-variation distance between these two distributions; recall that the total-variation distance between D and E, the conventional form of statistical distance, is $\sum_o |D_o - E_o|/2$.

For example, for $k = 1$ and $p = 11 = 4 \cdot 3 - 1$, the Elligator order distribution for each $A \in \{0, 5, 6\}$ is 100% order 3: if T_0 is output by Elligator then $[4]T_0$ always has order 3. The uniform model is that orders 3 and 1 appear with probability 2/3 and 1/3 respectively. The total-variation distance is $(|1 - 2/3| + |0 - 1/3|)/2 = 1/3$.

For $k = 2$ and $p = 59 = 4 \cdot 3 \cdot 5 - 1$, the uniform model is that orders 15, 5, 3, and 1 appear with probability 8/15, 4/15, 2/15, 1/15 respectively. Elligator for $A = 6$ has probability 6/14, 6/14, 2/14, 0 respectively, with total-variation distance $6/35 \approx 0.171429$. Elligator for $A = 11$ has a different distribution from $A = 6$: probability 8/14, 4/14, 1/14, 1/14 respectively. There are 9 choices of A overall, with total-variation distances ranging from ≈0.0619048 to ≈0.171429, averaging ≈0.110582.

Seeing two different values of A with different distributions shows that the result of replacing `UniformRandomPoints` with Elligator is not exactly an atomic block. This does not end the security analysis: for security it is enough to have something indistinguishable from an atomic block. If the total-variation distance drops quickly enough to reach, e.g., 2^{-128} for $p \approx 2^{512}$, then the Elligator orders are indistinguishable from uniform-point orders for every A, and are thus indistinguishable from one A to another.

For $p = 419 = 4 \cdot 3 \cdot 5 \cdot 7 - 1$, there are 27 choices of A, with total-variation distances averaging ≈0.0655745, ranging from ≈0.0357143 to ≈0.119780. For $p = 12011 = 4 \cdot 3 \cdot 7 \cdot 11 \cdot 13 - 1$, there are 195 choices of A, with total-variation distances averaging ≈0.0135444, ranging from ≈0.0063736 to ≈0.0232127. For $p = 78539 = 4 \cdot 3 \cdot 5 \cdot 7 \cdot 11 \cdot 17 - 1$, there are 459 choices of A, with total-variation distances averaging ≈0.00713331, ranging from ≈0.00353921 to ≈0.0115945. For $p = 1021019 = 4 \cdot 3 \cdot 5 \cdot 7 \cdot 11 \cdot 13 \cdot 17 - 1$, there are 1905 choices of A, with total-variation distances averaging ≈0.00493310, ranging from ≈0.00272376 to ≈0.00790233.

To see more information regarding the distributions, we inspected, for each A with $p = 419$, the full distribution of orders of (4 times) Elligator points in $E_A(\mathbb{F}_p)$. The green curves in Figure 4 show the minimum, quartiles, and maximum of the per-A distributions; for comparison, the red curve shows the uniform model. Figure 5 is for $p = 12011$. The green curves are closer to the red curve for $p = 12011$ than for $p = 419$.

Elligator simulators. Consider the following simulator, resampling from the uniform model: for each $A \in \mathcal{M}$, generate a uniform random sequence of $(p-3)/2$ elements of $[4]E_A(\mathbb{F}_p) \cong \mathbb{Z}/((p+1)/4)\mathbb{Z}$, and compute the distribution of orders of these elements.

Obviously this simulator is not exactly Elligator. For example, Elligator produces each element of $[4]E_A(\mathbb{F}_p)$ at most 4 times. More fundamentally, Elligator deterministically produces a particular distribution for each A, while the simulator produces a new random choice each time. As an extreme case, for $p = 11$, Elligator produces 100% order 3 as noted above, whereas for each $A \in \{0, 5, 6\}$ the simulator produces 100% 3 with probability $16/3^4$; 75% 3 and 25% 1 with probability $32/3^4$; 50% 3 and 50% 1 with probability $24/3^4$; 25% 3 and 75% 1 with probability $8/3^4$; and 100% 1 with probability $1/3^4$.

However, within the range of our experiments, this simulator produces similar results to Elligator. Compare Figure 5 to Figure 6, which gives an example of the simulator output for $p = 12011$.

A heuristic analysis of this simulator for arbitrary sizes of p proceeds as follows. For each positive integer d dividing $(p+1)/4$, there are $\varphi(d)$ elements of order d in $\mathbb{Z}/((p+1)/4)\mathbb{Z}$, where φ is Euler's phi function. For example, there is 1 element of order 1, and there are $(\ell_1 - 1) \cdots (\ell_n - 1)$ elements of order $(p+1)/4 = \ell_1 \cdots \ell_n$. Order d thus occurs with probability $4\varphi(d)/(p+1)$.

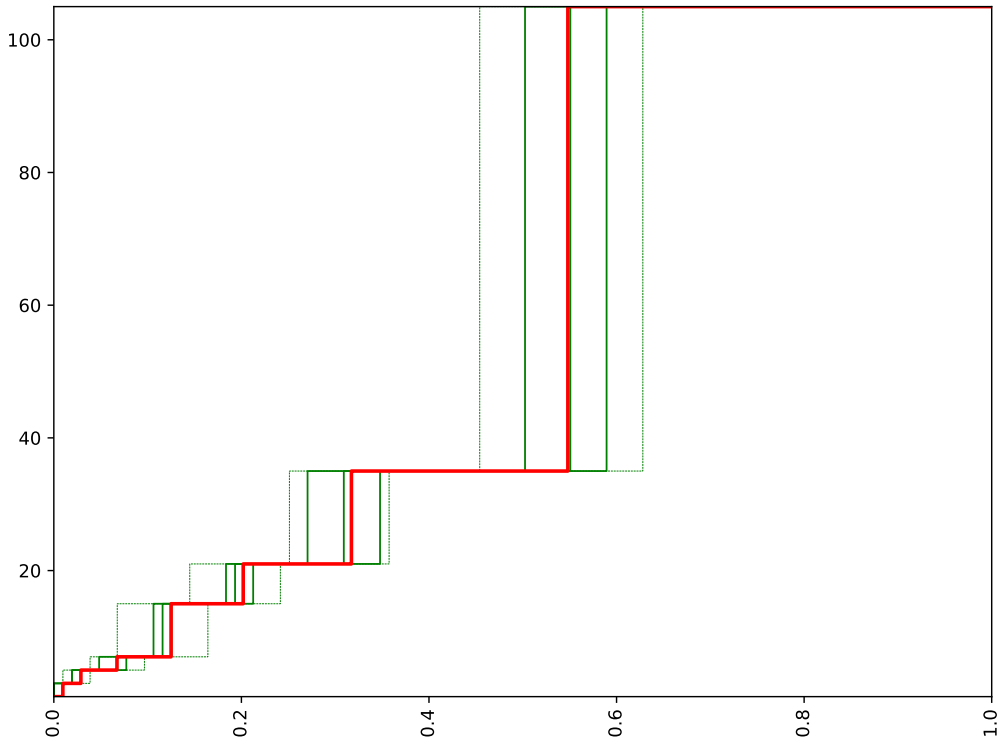

Figure 4: For $p = 419$, distributions of orders of points $[4]T_0$ in $E_A(\mathbb{F}_p)$. Red curve: the uniform model, choosing T_0 uniformly at random from $E_A(\mathbb{F}_p)$. Five green curves: minimum, first quartile, median, third quartile, and maximum of the per-A distributions when T_0 is output by Elligator.

In general, if a trial succeeds with probability q, then the number S of successes in N independent trials has average qN and standard deviation $\sqrt{q(1-q)N}$, so the ratio $(S-qN)/\sqrt{q(1-q)N}$ (assuming $0 < q < 1$) has average 0 and standard deviation 1. The distribution of $(S-qN)/\sqrt{q(1-q)N}$ rapidly approaches a normal distribution as $q(1-q)N$ increases. If the distribution were exactly normal then the half absolute value $|S-qN|/(2\sqrt{q(1-q)N})$ would have average $\sqrt{1/2\pi}$. One thus expects the variation $|S/N-q|/2$ to be approximately $\sqrt{q(1-q)/2N\pi}$ on average.

In particular, write S_d for the number of times that order d occurs among $(p-3)/2$ independent samples from the uniform distribution on $\mathbb{Z}/((p+1)/4)\mathbb{Z}$. Then S_d has average $q_d(p-3)/2$, where $q_d = 4\varphi(d)/(p+1)$, and standard deviation $\sqrt{q_d(1-q_d)(p-3)/2}$. One expects the variation $|2S_d/(p-3) - q_d|$ to be approximately $\sqrt{q_d(1-q_d)/(p-3)\pi}$ on average.

Summing over d says that the total-variation distance is, on average, approximately $\sum_d \sqrt{q_d(1-q_d)/(p-3)\pi}$. This is at most $\sum_d \sqrt{q_d/(p-3)\pi} = X/\sqrt{(p-3)\pi}$ where $X = \sum_d \sqrt{q_d}$. Notice that X factors as $\prod_j(\sqrt{1-1/l_j}+\sqrt{1/l_j})$, which is easy to compute even when p is large. For example, for the CSIDH-512 prime p, this product X is below 2^{11}, while $1/\sqrt{(p-3)\pi}$ is around 2^{-256}.

There are roughly $\sqrt{p}$ choices of A, and they usually vary in total-variation distance. For any particular d, one expects that there exists some A with approximately $\sqrt{\log p}$ standard deviations in the probability that d occurs: e.g., 19 standard deviations for the

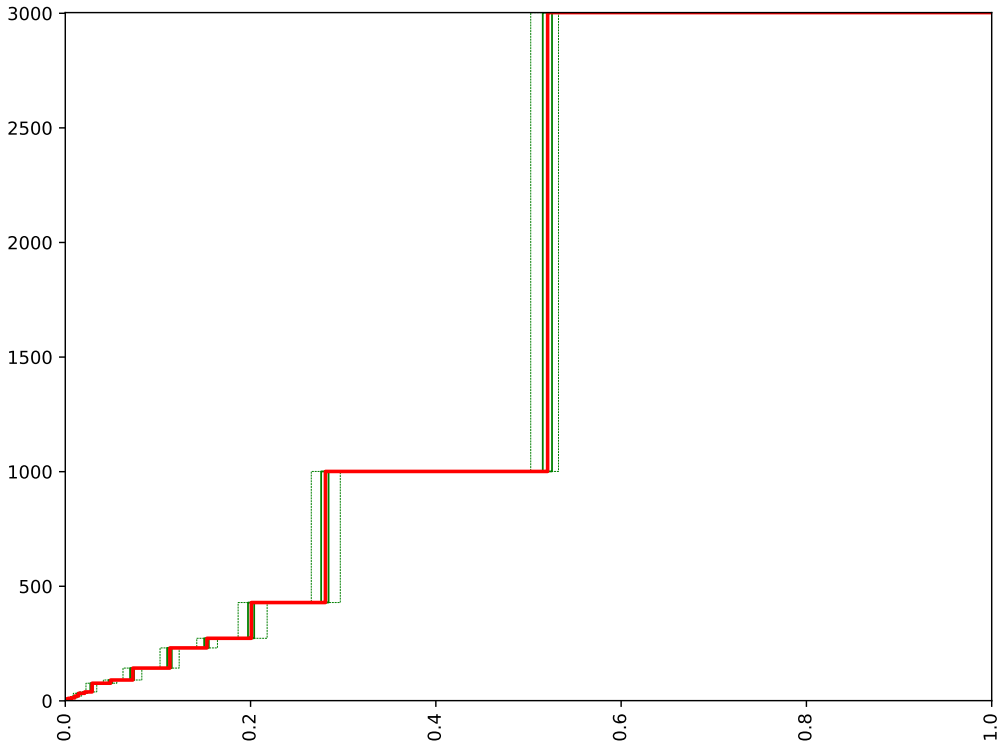

Figure 5: Same as Figure 5 but for $p = 12011$.

CSIDH-512 prime p (although finding just 10 deviations would be a large computation). This effect is on a smaller scale than X when $(p+1)/4$ has many small prime factors: i.e., the typical variation accumulated by many large divisors d is larger than occasional variation for the largest divisor. One does not expect several standard deviations to occur for several d simultaneously.

To summarize, one expects all total-variation distances from the simulator to be close to $X/\sqrt{(p-3)\pi}$. The ratios in Table 3 show that the actual Elligator total-variation distances are close to $X/\sqrt{(p-3)\pi}$ for $p \in \{11, 59, 419, 12011, 78539, 1021019\}$.

Zero hazards. The original Elligator paper [4] did not define Elligator 2 for $A = 0$. The application of Elligator to CSIDH attacks in [5] suggested handling $A = 0$ by precomputing a point of full order, or, alternatively, replacing the initial $A/(r^2-1)$ with r when $A = 0$.

Our CTIDH software (see Section 7) replaces the initial $A/(r^2-1)$ with $1/(r^2-1)$ when $A = 0$. For constant-time projective computations this seems slightly more efficient than replacing $A/(r^2-1)$ with r. We included this handling of $A = 0$ in the computations described above of the total-variation distance.

Beware that the alternative of precomputing a point of full order would not generally be safe in constructive applications. This precomputation eliminates failure cases for $A = 0$, making $A = 0$ easily distinguishable from other values of A via timing. An attacker that guesses the isogenies used in the victim's first AB, and provides a fake public key that is taken to 0 by those isogenies, can check this guess by watching timings of the victim's second AB. Once the attacker has enough confidence regarding the first isogenies, the attacker can move on to guessing the isogenies used in the victim's second AB. The attacker continues in this way to adaptively target the victim's full private key. This

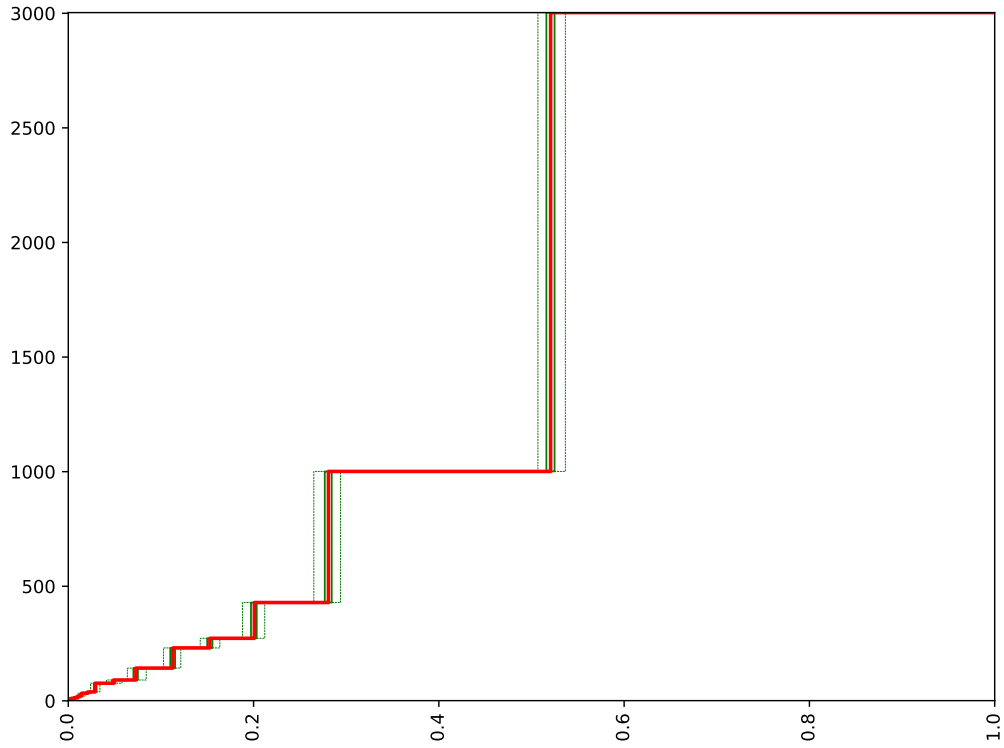

Figure 6: Simulation of Figure 5. The red curve is copied from Figure 5. For the green curves, the Elligator order distributions are replaced by random samples from the red curve.

adaptive timing attack breaks [16, Section 5.3, second paragraph]. On the other hand, always using a large enough number of iterations to reach a negligible failure probability, as in [5], would stop this attack.

One Elligator, or two? The order-distinguishability question for orders in $\tilde{E}_A(\mathbb{F}_p)$ is equivalent to the order-distinguishability question for orders in $E_{-A}(\mathbb{F}_p)$, so it does not need to be analyzed separately. However, a different security question arises if a single `UniformRandomPoints` call is replaced with a single Elligator call, as in [20] and subsequent constant-time CSIDH work. It is conceivable, for example, that the resulting element of $E_A(\mathbb{F}_p) \times \tilde{E}_A(\mathbb{F}_p)$ has a measurable correlation between 3 dividing the order in the $E_A(\mathbb{F}_p)$ part and 5 dividing the order in the $\tilde{E}_A(\mathbb{F}_p)$ part, even if the order in each part separately is indistinguishable from uniform.

Our `high-ctidh` software simplifies the security analysis by using Elligator once to generate an element of $E_A(\mathbb{F}_p)$, and using Elligator again to generate an independent element of $\tilde{E}_A(\mathbb{F}_p)$. It is not clear that this is a slowdown: independently generating two points lets the software save the cost of pushing one of the two points through an isogeny, and Legendre-symbol computations could be fast enough to justify this purely from a speed perspective. (Note that if Legendre-symbol computations have low enough cost then it is easy to argue for incurring the slowdown of using two Legendre-symbol computations on each curve to generate uniform random points, skipping Elligator and further simplifying the security analysis.)

On the other hand, a single Elligator call could be best for speed, and is used in several

Table 3: Heuristic analysis of the Elligator simulator (see text) compared to actual Elligator order distributions. "$\#\mathcal{E}$": number of choices of A. "$\lg X$": logarithm base 2 of $X = \prod_j(\sqrt{1-1/l_j} + \sqrt{1/l_j})$. "$\lg H$": logarithm base 2 of $H = \sum_d \sqrt{q_d/(p-3)\pi} = X/\sqrt{(p-3)\pi}$. "$\lg H_1$": logarithm base 2 of $H_1 = \sum_d \sqrt{q_d(1-q_d)/(p-3)\pi}$. "min" and "avg" and "max": logarithm base 2 of the minimum and average and maximum, over A, of the total-variation distance between Elligator and the uniform model. "ratios": H_1 and minimum and average and maximum, divided by H, without logarithms.

p	$\#\mathcal{E}$	$\lg X$	$\lg H$	$\lg H_1$	min	avg	max				ratios
11	3	0.48	−1.85	−2.41	−1.58	−1.58	−1.58	0.68	1.20	1.20	1.20
59	9	0.90	−2.83	−3.12	−4.01	−3.18	−2.54	0.82	0.44	0.78	1.22
419	27	1.29	−3.89	−4.07	−4.81	−3.93	−3.06	0.89	0.53	0.97	1.78
12011	195	1.50	−6.10	−6.26	−7.29	−6.21	−5.43	0.90	0.44	0.93	1.60
78539	459	1.89	−7.06	−7.16	−8.14	−7.13	−6.43	0.94	0.47	0.95	1.55
1021019	1905	2.20	−8.61	−8.67	−9.52	−8.66	−7.98	0.96	0.53	0.96	1.54

previous papers. So we also studied the joint distribution of $E_A(\mathbb{F}_p) \times \tilde{E}_A(\mathbb{F}_p)$ orders.

For $A = 0$, the Elligator extensions mentioned above produce outputs of the form $((x, y), (-x, iy))$. The "distortion map" from (x, y) to $(-x, iy)$ is compatible with elliptic-curve addition, so it preserves the order of points. This reduces the security question for $E_0(\mathbb{F}_p) \times \tilde{E}_0(\mathbb{F}_p)$ to the security question for $E_0(\mathbb{F}_p)$, which was addressed above.

For nonzero A, our computations did not detect any such correlations. We instead compared the pair of orders to a uniform-pair model, namely the orders of two independent uniform random elements T_0, T_1 of $\mathbb{Z}/((p+1)/4)\mathbb{Z}$. We found total-variation distance averaging ≈0.312619 (maximum ≈0.358730) for $p = 59$, ≈0.207722 (maximum ≈0.260199) for $p = 419$, ≈0.0512755 (maximum ≈0.0678949) for $p = 12011$, and ≈0.0361776 (maximum ≈0.0416506) for $p = 78539$.

It is not surprising that the distance from the joint distribution to the uniform-pair model is generally larger than the distance from the single-point distribution to the uniform model. There are many more possibilities for a pair of orders than for a single order.

To quantify this, consider a joint-distribution simulator that resamples from the uniform-pair model. If d_1 and d_2 are positive integers dividing $(p+1)/4$ then there are $\varphi(d_1)\varphi(d_2)$ pairs of elements T_1, T_2 of orders d_1, d_2 respectively in $\mathbb{Z}/((p+1)/4)\mathbb{Z}$. A heuristic analysis proceeds as before, with q_d replaced by $q_{d_1}q_{d_2}$, and X replaced by X^2, giving the estimate $X^2/\sqrt{(p-3)\pi}$. This estimate is ≈0.263654 for $p = 59$, ≈0.164434 for $p = 419$, ≈0.0410512 for $p = 12011$, and ≈0.0277182 for $p = 78539$. If the actual joint-Elligator distances remain close to $X^2/\sqrt{(p-3)\pi}$ for all CSIDH primes p then the distances are acceptably small for CSIDH-512.