On Lions and Elligators: An Efficient Constant-Time Implementation of CSIDH

Michael Meyer[1,2(✉)], Fabio Campos[1], and Steffen Reith[1]

[1] Department of Computer Science, University of Applied Sciences, Wiesbaden, Germany
{Michael.Meyer,FabioFelipe.Campos,Steffen.Reith}@hs-rm.de
[2] Department of Mathematics, University of Würzburg, Würzburg, Germany

Abstract. The recently proposed CSIDH primitive is a promising candidate for post quantum static-static key exchanges with very small keys. However, until now there is only a variable-time proof-of-concept implementation by Castryck, Lange, Martindale, Panny, and Renes, recently optimized by Meyer and Reith, which can leak various information about the private key. Therefore, we present an efficient constant-time implementation that samples key elements only from intervals of nonnegative numbers and uses dummy isogenies, which prevents certain kinds of side-channel attacks. We apply several optimizations, e.g. Elligator and the newly introduced SIMBA, in order to get a more efficient implementation.

Keywords: CSIDH · Isogeny-based cryptography · Post-quantum cryptography · Constant-time implementation

1 Introduction

Isogeny-based cryptography is the most juvenile family of the current proposals for post-quantum cryptography. The first cryptosystem based on the hardness of finding an explicit isogeny between two given isogenous elliptic curves over a finite field was proposed in 1997 by Couveignes [10], eventually independently rediscovered by Rostovtsev and Stolbunov [19] in 2004, and therefore typically called CRS. Childs, Jao, and Soukharev [7] showed in 2010 that CRS can be broken using a subexponential quantum algorithm by solving an abelian hidden shift problem. To avoid this attack, Jao and De Feo [13] invented a new isogeny-based scheme SIDH (supersingular isogeny Diffie-Hellman) that works with supersingular curves over $\mathbb{F}_{p^2}$. The current state-of-the-art implementation is SIKE [12], which was submitted to the NIST post-quantum cryptography competition [17].

De Feo, Kieffer and Smith optimized CRS in 2018 [11]. Their ideas led to the development of CSIDH by Castryck, Lange, Martindale, Panny, and Renes [6],

This work was partially supported by Elektrobit Automotive, Erlangen, Germany.

J. Ding and R. Steinwandt (Eds.): PQCrypto 2019, LNCS 11505, pp. 307–325, 2019.
https://doi.org/10.1007/978-3-030-25510-7_17

who adapted the CRS scheme to supersingular curves and isogenies defined over a prime field $\mathbb{F}_p$. They implemented the key exchange as a proof-of-concept, which is efficient, but does not run in constant time, and can therefore leak information about private keys. We note that building an efficient constant-time implementation of CSIDH is not as straightforward as in SIDH, where, speaking of running times, only one Montgomery ladder computation depends on the private key (see [9]).

In this paper we present a constant-time implementation of CSIDH with many practical optimizations, requiring only a small overhead of factor 3.03 compared to the fastest variable-time implementation from [14].

Organization. The rest of this paper is organized as follows. The following section gives a brief algorithmic introduction to CSIDH [6]. Leakage scenarios based on time, power analysis, and cache timing are presented in Sect. 3. In Sect. 4, we suggest different methods on how to avoid these leakages and build a constant-time implementation. Section 5 contains a straightforward application of our suggested methods, and various optimizations. Thereafter, we provide implementation results in Sect. 6 and give concluding remarks in Sect. 7. Appendices A and B give more details about our implementations and algorithms.

Note that there are two different notions of constant-time implementations, as explained in [3]. In our case, it suffices to work with the notion that the running time does not depend upon the choice of the private key, but may vary due to randomness. The second notion specifies strict constant time, meaning that the running time must be the same every time, independent from private keys or randomness. Throughout this paper, 'constant time' refers to the first notion described above.

Related Work. In [3], Bernstein, Lange, Martindale, and Panny describe constant-time implementations in the second notion from above, which is required for quantum attacks. In this paper, we follow the mentioned different approach for an efficient constant-time implementation, but reuse some of the techniques from [3].

2 CSIDH

We only cover the algorithmic aspects of CSIDH here, and refer to [6] for the mathematical background and a more detailed description.

We first choose a prime of the form $p = 4 \cdot \ell_1 \cdot \ldots \cdot \ell_n - 1$, where the ℓ_i are small distinct odd primes. We work with supersingular curves over $\mathbb{F}_p$, which guarantees the existence of points of the orders ℓ_i, that enable us to compute ℓ_i-isogenies from kernel generator points by Vélu-type formulas [20].

A private key consists of a tuple $(e_1, \ldots, e_n)$, where the e_i are sampled from an interval $[-B, B]$. The absolute value $|e_i|$ specifies how many ℓ_i-isogenies have to be computed, and the sign of e_i determines, whether points on the current

Algorithm 1. Evaluating the class group action.

Input : $a \in \mathbb{F}_p$ such that $E_a : y^2 = x^3 + ax^2 + x$ is supersingular, and a list of integers $(e_1, ..., e_n)$ with $e_i \in \{-B, ..., B\}$ for all $i \leq n$.
Output: $a' \in \mathbb{F}_p$, the curve parameter of the resulting curve $E_{a'}$.

```
1  while some e_i ≠ 0 do
2      Sample a random x ∈ F_p.
3      Set s ← +1 if x^3 + ax^2 + x is a square in F_p, else s ← −1.
4      Let S = {i | sign(e_i) = s}.
5      if S = ∅ then
6          Go to line 2.
7      P = (x : 1), k ← ∏_{i∈S} ℓ_i, P ← [(p + 1)/k]P.
8      foreach i ∈ S do
9          K ← [k/ℓ_i]P.
10         if K ≠ ∞ then
11             Compute a degree-ℓ_i isogeny φ : E_a → E_{a'} with ker(φ) = ⟨K⟩.
12             a ← a', P ← φ(P), k ← k/ℓ_i, e_i ← e_i − s.
```

curve or on its twist have to be used as kernel generators. One can represent this graphically: Over $\mathbb{F}_p$, the supersingular ℓ_i-isogeny graph consists of distinct cycles. Therefore, we have to walk $|e_i|$ steps through the cycle for ℓ_i, and the sign of e_i tells us the direction.

Since this class group action is commutative, it allows a basic Diffie-Hellman-type key exchange: Starting from a supersingular curve E_0, Alice and Bob choose a private key as described above, and compute their public key curves E_A resp. E_B via isogenies, as described in Algorithm 1. Then Alice repeats her computations, this time starting at the curve E_B, and vice versa. Both parties then arrive at the same curve E_{AB}, which represents their shared secret. Furthermore, public keys can be verified efficiently in CSIDH (see [6]). Therefore, a static-static key-exchange is possible.

However, the quantum security is still an open problem. For our implementation we use CSIDH-512, the parameter set from [6], that is conjectured to satisfy NIST security level 1. In the light of the subexponential quantum attack on CRS and CSIDH [7], more analysis on CSIDH has been done in [3–5].

3 Leakage Scenarios

It is clear and already mentioned in [6] that the proof-of-concept implementation of CSIDH is not side-channel resistant. In this paper we focus on three scenarios that can leak information on the private key. Note that the second scenario features a stronger attacker. Further, there will of course be many more scenarios for side-channel attacks.

Timing Leakage. As the private key in CSIDH specifies how many isogenies of each degree have to be computed, it is obvious that this (up to additional effort for point multiplications due to the random choice of points) determines the running time of the algorithm. As stated in [14], the worst case running time occurs for the private key $(5, 5, \ldots, 5)$, and takes more than 3 times as much as in the average case. The other extreme is the private key $(0, 0, \ldots, 0)$, which would require no computations at all. However, in a timing-attack protected implementation, the running time should be independent from the private key.

Power Analysis. Instead of focusing on the running time, we now assume that an attacker can measure the power consumption of the algorithm. We further assume that from the measurement, the attacker can determine blocks which represent the two main primitives in CSIDH, namely point multiplication and isogeny computation, and can separate these from each other. Now assume that the attacker can separate the loop iterations from each other. Then the attacker can determine which private key elements share the same sign from the isogeny blocks that are performed in the same loop, since they have variable running time based on the isogeny degree. This significantly reduces the possible key space and therefore also the complexity of finding the correct key.

Cache Timing Attacks. In general, data flow from the secret key to branch conditions and array indices must be avoided in order to achieve protection against cache timing attacks [1]. Our implementation follows these guidelines to avoid vulnerabilities against the respective possible attacks.

4 Mitigating Leakage

In this section we give some ideas on how to fix these possible leakages in an implementation of CSIDH. We outline the most important ideas here, and give details about how to implement them efficiently in CSIDH-512 in Sect. 5.

Dummy Isogenies. First, it seems obvious that one should compute a constant number of isogenies of each degree ℓ_i, and only use the results of those required by the private key, in order to obtain a constant running time. However, in this case additional multiplications are required, if normal isogenies and unused isogenies are computed in the same loop[1]. We adapt the idea of using dummy isogenies from [14] for that cause. Meyer and Reith propose to design dummy isogenies, which instead of updating the curve parameters and evaluating the curve point P, compute $[\ell_i]P$ in the degree-ℓ_i dummy isogeny. Since the isogeny algorithm computes $[\frac{\ell_i - 1}{2}]K$ for the kernel generator K, one can replace K by P there, and perform two more differential additions to compute $[\ell_i]P$. The curve parameters remain unchanged.

[1] This is required, since otherwise, an attacker in the second leakage scenario can determine the private key easily.

In consequence, a dummy isogeny simply performs a scalar multiplication. Therefore, the output point $[\ell_i]P$ then has order not divisible by ℓ_i, which is important for using this point to compute correct kernel generators in following iterations. Further, one can design the isogeny and dummy isogeny algorithms for a given degree ℓ_i such that they perform the same number and sequence of operations with only minor computational overhead compared to the isogenies from [14]. This is important to make it hard for side-channel attackers to distinguish between those two cases, since conditionally branching can be avoided with rather small overhead.

Balanced vs. Unbalanced Private Keys. Using dummy isogenies to spend a fixed time on isogeny computations in not enough for a constant-time implementation, however. Another problem lies in the point multiplications in line 7 and 9 of Algorithm 1. We use an observation from [14] to illustrate this. They consider the private keys $(5, 5, 5, \ldots)$ and $(5, -5, 5, -5, \ldots)$ and observe that for the first key, the running time is 50% higher than for the second key. The reason for this is that in the first case in order to compute one isogeny of each degree, the multiplication in line 7 is only a multiplication by 4, and the multiplication in line 9 has a factor of bitlength 509 in the first iteration, 500 in the second iteration, and so on.

For the second key, we have to perform one loop through the odd i and one through the even i in order two compute one isogeny of each degree ℓ_i. Therefore, the multiplications in line 7 are by 254 resp. 259 bit factors, while the bitlengths of the factors in the multiplications in line 9 are $252, 244, \ldots$, resp. 257, 248, and so on (see Fig. 1). In total, adding up the bitlengths of all factors, we can measure the cost of all point multiplications for the computation of one isogeny per degree, where we assume that the condition in line 10 of Algorithm 1 never fails, since one Montgomery ladder step is performed per bit. For the first key, we end up with 16813 bits, while for the second key we only have 9066 bits.

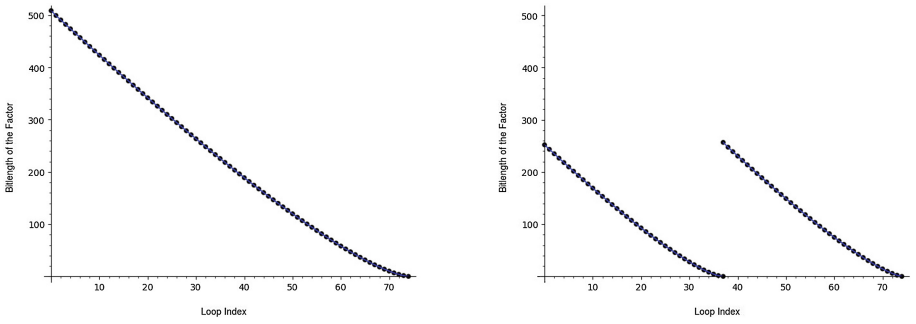

Fig. 1. Bitlengths of factors for computing one isogeny per degree for the keys $(5, 5, \ldots, 5)$ (left) and $(5, -5, 5, -5, \ldots)$ (right).

This can be generalized to any private key: The more the key elements (or the products of the respective ℓ_i) are unbalanced, i.e. many of them share the same sign, the more the computational effort grows, compared to the perfectly balanced case from above. This behavior depends on the private key and can therefore leak information. Hence, it is clear that we have to prevent this in order to achieve a constant-time implementation.

One way to achieve this is to use constant-time Montgomery ladders that always run to the maximum bitlength, no matter how large the respective factor is. However, this would lead to a massive increase in running time. Another possibility for handling this is to only choose key elements of a fixed sign. Then we have to adjust the interval from which we sample the integer key elements, e.g. from $[-5, 5]$ to $[0, 10]$ in CSIDH-512. This however doubles the computational effort for isogenies (combined normal and dummy isogenies). We will return to this idea later.

Determining the Sign Distribution. In our second leakage scenario, an attacker might determine the sign distribution of the key elements by identifying blocks of isogeny resp. dummy isogeny computations. One way of mitigating this attack would be to let each degree-ℓ_i isogeny run as long as a ℓ_{max}-isogeny, where ℓ_{max} is the largest ℓ_i. As used in [3], this is possible because of the Matryoshka-doll structure of the isogeny algorithms. This would allow an attacker in the second leakage scenario to only determine the number of positive resp. negative elements, but not their distribution, at the cost of a large increase of computational effort. We can also again restrict to the case that we only choose nonnegative (resp. only nonpositive) key elements. Then there is no risk of leaking information about the sign distribution of the elements, since in this setting the attacker knows this beforehand, at the cost of twice as many isogeny computations.

Limitation to Nonnegative Key Elements. Since this choice eliminates both of the aforementioned possible leakages, we use the mentioned different interval to sample private key elements from. In CSIDH-512, this means using the interval $[0, 10]$ instead of $[-5, 5]$. One might ask if this affects the security properties of CSIDH. As before, there are 11^{74} different tuples to choose from in CSIDH-512. Castryck et al. argue in [6] that there are multiple vectors $(e_1, e_2, \ldots, e_n)$, which represent the same ideal class, meaning that the respective keys are equivalent. However, they assume by heuristic arguments that the number of short representations per ideal class is small, i.e. the 11^{74} different keys $(e_1, e_2, \ldots, e_n)$, where all e_i are sampled from the interval $[-5, 5]$, represent not much less than 11^{74} ideal classes. If we now have two equivalent keys $e \neq f$ sampled from $[-5, 5]$, then we have a collision for our shifted interval as well, since shifting all elements of e and f by $+5$ results in equivalent keys $e' \neq f'$ with

elements in $[0, 10]$, and vice versa. Therefore, our shifted version is equivalent to CSIDH-512 as defined in [6][2].

In the following sections we focus on optimized implementations, using the mentioned countermeasures against attacks, i.e. sampling key elements from the interval $[0, 10]$ and using dummy isogenies.

5 Efficient Implementation

5.1 Straightforward Implementation

First, we describe the straightforward implementation of the evaluation of the class group action in CSIDH-512 with the choices from above, before applying various optimizations. We briefly go through the implementation aspects of the main primitives, i.e. point multiplications, isogenies and dummy isogenies, and explain why this algorithm runs in constant time, and does not leak information about the private key.

Parameters. As described in [6], we have a prime number $p = 4 \cdot \ell_1 \cdot \ell_2 \cdot ... \cdot \ell_n - 1$, where the ℓ_i are small distinct odd primes. We further assume that we have $\ell_1 > \ell_2 > ... > \ell_n$. In CSIDH-512 we have $n = 74$, and we sample the elements of private keys $(e_1, e_2, \ldots, e_n)$ from $[0, 10]$.

Handling the Private Key. Similar to the original implementation of Castryck et al., we copy the elements of the private key in an array $e = (e_1, e_2, \ldots, e_n)$, where e_i determines how many isogenies of degree ℓ_i we have to compute. Furthermore, we set up another array $f = (10-e_1, 10-e_2, \ldots, 10-e_n)$, to determine how many dummy isogenies of each degree we have to compute. As we go through the algorithm, we compute all the required isogenies and dummy isogenies, reducing e_i resp. f_i by 1 after each degree-ℓ_i isogeny resp. dummy isogeny. We therefore end up with a total of 10 isogeny computations (counting isogenies and dummy isogenies) for each ℓ_i.

Sampling Random Points. In Algorithm 2 line 3, we have to find curve points on the current curve that are defined over $\mathbb{F}_p$ instead of $\mathbb{F}_{p^2} \backslash \mathbb{F}_p$. As in [6] this can be done by sampling a random $x \in \mathbb{F}_p$, and computing y^2 by the curve equation $y^2 = x^3 + ax^2 + x$. We then check if y is defined over $\mathbb{F}_p$ by a Legendre symbol computation, i.e. by checking if $(y^2)^{(p-1)/2} \equiv 1 \pmod{p}$. If this is not the case, we simply repeat this procedure until we find a suitable point. Note that we require the curve parameter a to be in affine form. Since a will typically be in projective form after isogeny computations, we therefore have to compute the affine parameter each time before sampling a new point.

[2] One could also think of using the starting curve E', which is the result of applying the key $(5, 5, \ldots, 5)$ to the curve E_0. Then for a class group action evaluation using key elements from $[-5, 5]$ and the starting curve E' is equivalent to using key elements from $[0, 10]$ and the starting curve E_0.

Algorithm 2. Constant-time evaluation of the class group action in CSIDH-512.

Input : $a \in \mathbb{F}_p$ such that $E_a : y^2 = x^3 + ax^2 + x$ is supersingular, and a list of integers $(e_1, ..., e_n)$ with $e_i \in \{0, 1, .., 10\}$ for all $i \leq n$.
Output: $a' \in \mathbb{F}_p$, the curve parameter of the resulting curve $E_{a'}$.

Initialize $k = 4$, $e = (e_1, ..., e_n)$ and $f = (f_1, ..., f_n)$, where $f_i = 10 - e_i$.
while some $e_i \neq 0$ or $f_i \neq 0$ **do**
  Sample random values $x \in \mathbb{F}_p$ until we have some x where $x^3 + ax^2 + x$ is a square in $\mathbb{F}_p$.
  Set $P = (x : 1)$, $P \leftarrow [k]P$, $S = \{i \mid e_i \neq 0 \text{ or } f_i \neq 0\}$.
  foreach $i \in S$ **do**
    Let $m = \prod_{j \in S, j > i} \ell_i$.
    Set $K \leftarrow [m]P$.
    if $K \neq \infty$ **then**
      if $e_i \neq 0$ **then**
        Compute a degree-ℓ_i isogeny $\varphi : E_a \rightarrow E_{a'}$ with $ker(\varphi) = \langle K \rangle$.
        $a \leftarrow a'$, $P \leftarrow \varphi(P)$, $e_i \leftarrow e_i - 1$.
      else
        Compute a degree-ℓ_i dummy isogeny:
        $a \leftarrow a$, $P \leftarrow [\ell_i]P$, $f_i \leftarrow f_i - 1$.
      if $e_i = 0$ *and* $f_i = 0$ **then**
        Set $k \leftarrow k \cdot \ell_i$.

Elliptic Curve Point Multiplications. Since we work with Montgomery curves, using only projective XZ-coordinates, and projective curve parameters $a = A/C$, we can use the standard Montgomery ladder as introduced in [15], adapted to projective curve parameters as in [9]. This means that per bit of the factor, one combined doubling and differential addition is performed.

Isogenies. For the computation of isogenies, we use the formulas presented by Meyer and Reith in [14]. They combine the Montgomery isogeny formulas by Costello and Hisil [8], and Renes [18] with the twisted Edwards formulas by Moody and Shumow [16], in order to obtain an efficient algorithm for the isogeny computations in CSIDH. For a ℓ_i-isogeny, this requires a point K of order ℓ_i as kernel generator, and the projective parameters A and C of the current curve. It outputs the image curve parameters A' and C', and the evaluation of the point P. As mentioned before, the algorithm computes all multiples of the point K up to the factor $\frac{\ell_i - 1}{2}$. See e.g. [3] for more details.

Dummy Isogenies. As described before, we want the degree-ℓ_i dummy isogenies to output the scalar multiple $[\ell_i]P$ instead of an isogeny evaluation of P. Therefore, we interchange the points K and P in the original isogeny algorithm,

such that it computes $[\frac{\ell_i-1}{2}]P$. We then perform two more differential additions, i.e. compute $[\frac{\ell_i+1}{2}]P$ from $[\frac{\ell_i-1}{2}]P$, P, and $[\frac{\ell_i-3}{2}]P$, and compute $[\ell_i]P$ from $[\frac{\ell_i+1}{2}]P$, $[\frac{\ell_i-1}{2}]P$, and P.

As mentioned before, we want isogenies and dummy isogenies of degree ℓ_i to share the same code in order to avoid conditionally branching. Hence, the two extra differential additions are also performed in the isogeny algorithm, without using the results. In our implementation, a conditional point swapping based on a bitmask ensures that the correct input point is chosen. This avoids conditionally branching that depends on the private key in line 9 of Algorithm 2 (and lines 11 and 27 of Algorithm 5).

If one is concerned that a side-channel attacker can detect that the curve parameters A and C are not changed for some time (meaning that a series of dummy isogenies is performed), one could further rerandomize the projective representation of the curve parameter A/C by multiplying A and C by the same random number[3] $1 < \alpha < p$.

5.2 Running Time

We now explain why this algorithm runs in constant time. As already explained, we perform 10 isogeny computations (counting isogenies and dummy isogenies) for each degree ℓ_i. Furthermore, isogenies and dummy isogenies have the same running time, since they share the same code, and conditionally branching is avoided. Therefore the total computational effort for isogenies is constant, independent from the respective private key. We also set the same condition (line 8 of Algorithm 2) for the kernel generator for the computation of a dummy isogeny, in order not to leak information.

Sampling random points and finding a suitable one doesn't run in constant time in Algorithm 2. However, the running time only depends on randomly chosen values, and does not leak any information on the private key.

Now for simplicity assume that we always find a point of full order, i.e. a point that can be used to compute one isogeny of each degree ℓ_i. Then it is easy to see that the total computational effort for scalar multiplications in Algorithm 2 is constant, independent from the respective private key. If we now allow random points, we will typically not satisfy the condition in line 8 of Algorithm 2 for all i. Therefore, additional computations (sampling random points, and point multiplications) are required. However, this does not leak information about the private key, since this only depends on the random choice of curve points, but not on the private key.

Hence, we conclude that the implementation of Algorithm 2 as described here prevents the leakage scenarios considered in Sect. 3. It is however quite slow compared to the performance of variable-time CSIDH-512 in [6,14]. In the following section, we focus on how to optimize and speed up the implementation.

[3] One could actually use an intermediate value $\alpha \in \mathbb{F}_p \backslash \{0, 1\}$ of the isogeny computation, since the factor is not required to be truly random.

5.3 Optimizations

Sampling Points with Elligator. In [3] Bernstein, Lange, Martindale, and Panny pointed out that Elligator [2], specifically the Elligator 2 map, can be used in CSIDH to be able to choose points over the required field of definition. Since we only need points defined over $\mathbb{F}_p$, this is especially advantageous in our situation. For $a \neq 0$ the Elligator 2 map works as follows (see [3]):

- Sample a random $u \in \{2, 3, \ldots, (p-1)/2\}$.
- Compute $v = a/(u^2 - 1)$.
- Compute e, the Legendre symbol of $v^3 + av^2 + v$.
- If $e = 1$, output v. Otherwise, output $-v - a$.

Therefore, for all $a \neq 0$, we can replace the search for a suitable point in line 3 of Algorithm 2, at the cost of an extra inversion. However, as explained by Bernstein et al., one can precompute $1/(u^2 - 1)$ for some values of u, e.g. for $u \in \{2, 3, 4, \ldots\}$. Then the cost is essentially the same as for the random choice of points, but we always find a suitable point this way, compared to the probability of $1/2$ when sampling random points. This could, however, potentially lead to the case that we cannot finish the computation: Consider that we only have one isogeny of degree ℓ_i left to compute, but for all of the precomputed values of u, the order of the corresponding point is not divided by ℓ_i. Then we would have to go back to a random choice of points to finish the computation. However, our experiments suggest that it is enough to have 10 precomputed values. Note that the probability for actually finding points of suitable order appears to be almost unchanged when using Elligator instead of random points, as discussed in [3].

For $a = 0$, Bernstein et al. also show how to adapt the Elligator 2 map to this case, but also argue that one could precompute a point of full order (or almost full order, i.e. divided by all ℓ_i) and simply use this point whenever $a = 0$. We follow their latter approach.

SIMBA (Splitting Isogeny Computations into Multiple Batches). In Sect. 4, we analyzed the running time of variable-time CSIDH-512 for the keys $e_1 = (5, 5, \ldots, 5)$ and $e_2 = (5, -5, 5, -5, \ldots)$. For the latter, the algorithm is significantly faster, because of the smaller multiplications during the loop (line 9 of Algorithm 1), see Fig. 1. We adapt and generalize this observation here, in order to speed up our constant-time implementation.

Consider for our setting the key $(10, 10, \ldots, 10)$ and that we can again always choose points of full order. To split the indices in two sets (exactly as Algorithm 1 does for the key e_2), we define the sets $S_1 = \{1, 3, 5, \ldots, 73\}$ and $S_2 = \{2, 4, 6, \ldots, 74\}$. Then the loops through the ℓ_i for $i \in S_1$ resp. $i \in S_2$ require significantly smaller multiplications, while only requiring to compute $[4k]P$ with $k = \prod_{i \in S_2} \ell_i$ resp. $k = \prod_{i \in S_1} \ell_i$ beforehand. We now simply perform 10 loops for each set, and hence this gives exactly the same speedup over Algorithm 2, as Algorithm 1 gives for the key e_2 compared to e_1, by using two batches of indices instead of only one.

One might ask if splitting the indices in two sets already gives the best speedup. We generalize the observation from above, now splitting the indices into m batches, where $S_1 = \{1, m+1, 2m+1, \ldots\}$, $S_2 = \{2, m+2, 2m+2, \ldots\}$, and so on[4]. Before starting a loop through the indices $i \in S_j$ with $1 \leq j \leq m$, one now has to compute $[4k]P$ with $k = \prod_{h \notin S_j} \ell_h$. The number and size of these multiplications grows when m grows, so we can expect that the speedup turns into an increasing computational effort when m is too large.

To find the best choice for m, we computed the total number of Montgomery ladder steps during the computation of one isogeny of each degree in CSIDH-512 for different m, with the assumptions from above. We did not take into account here that when m grows, we will have to sample more points (which costs at least one Legendre symbol computation each), since this depends on the cost ratio between Montgomery ladder steps and Legendre symbol computations in the respective implementation. Table 1 shows that the optimal choice should be around $m = 5$.

Table 1. Number of Montgomery ladder steps for computing one isogeny of each degree in CSIDH-512 for different numbers of batches m.

m	1	2	3	4	5	6	7
Ladder steps	16813	9066	6821	5959	5640	5602	5721

If we now come back to the choice of points through Elligator, the assumption from above does not hold anymore, and with very high probability, we will need more than 10 loops per index set. Typically, soon after 10 loops through each batch the large degree isogenies will be finished, while there are some small degree isogenies left to compute. In this case our optimization backfires, since in this construction, the indices of the missing ℓ_i will be distributed among the m different batches. We therefore need large multiplications in order to only check a few small degrees per set. Hence it is beneficial to define a number $\mu \geq 10$, and merge the batches after μ steps, i.e. simply going back to Algorithm 2 for the computation of the remaining isogenies. We dub this construction SIMBA-m-μ.

Sampling Private Key Elements from Different Intervals. Instead of sampling all private key elements from the interval $[0, 10]$, and in total computing 10 isogenies of each degree, one could also consider to choose the key elements from different intervals for each isogeny degree, as done in [11]. For a private key $e = (e_1, e_2, \ldots, e_n)$, we can choose an interval $[0, B_i]$ for each e_i, in order to e.g. reduce the number of expensive large degree isogenies at the cost of computing

[4] Note that in [3] a similar idea is described. However, in their algorithm only two isogeny degrees are covered in each iteration. Our construction makes use of the fact that we restrict to intervals of nonnegative numbers for sampling the private key elements.

more small degree isogenies. We require $\prod_i (B_i+1) \approx 11^{74}$, in order to obtain the same security level as before. For the security implication of this choice, similar arguments as in Sect. 4 apply.

Trying to find the optimal parameters B_i leads to a large integer optimization problem, which is not likely to be solvable exactly. Therefore, we heuristically searched for parameters likely to improve the performance of CSIDH-512. We present them in Sect. 6 and Appendix A.

Note that if we choose $B = (B_1, \ldots, B_n)$ differently from $B = (10, 10, \ldots, 10)$, the benefit of our optimizations above will change accordingly. Therefore, we changed the parameters m and μ in our implementation according to the respective B.

Skip Point Evaluations. As described before, the isogeny algorithms compute the image curve parameters, and push a point P through the isogeny. However, in the last isogeny per loop, this is unnecessary, since we choose a new point after the isogeny computation anyway. Therefore, it saves some computational effort, if we skip the point evaluation part in these cases.

Application to Variable-Time CSIDH. Note that many of the optimizations from above are also applicable to variable-time CSIDH-512 implementations as in [14] or [6]. We could therefore also speed up the respective implementation results using the mentioned methods.

6 Implementation Results

We implemented our optimized constant-time algorithm in C, using the implementation accompanying [14], which is based on the implementation from the original CSIDH paper by Castryck et al. [6]. For example the implementation of the field arithmetic in assembly is the one from [6]. Our final algorithm, containing all the optimizations from above, can be found in Appendix B.

Since we described different optimizations that can influence one another, it is not straightforward to decide which parameters B, m, and μ to use. Therefore, we tested various choices and combinations of parameters B, m, and μ, assuming $\ell_1 > \ell_2 > ... > \ell_n$. The parameters and implementation results can be found in Appendix A. The best parameters we found are given by

$$B = [5, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 8, 8, 8, 8, 8, 8, 8, 11, 11, 11,$$
$$11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 13, 13, 13, 13, 13, 13, 13$$
$$13, 13, 13, 13, 13, 13, 13, 13, 13, 13, 13, 13, 13, 13, 13, 13, 13, 13, 13, 13, 13, 13, 13]$$

using SIMBA-5-11, where the key element e_i is chosen from $[0, B_i]$. We do not claim that these are the best parameters; there might be better choices that we did not consider in our experiments.

We further tried to rearrange the order of the primes ℓ_i in the different loops. As pointed out in [14], it is beneficial to go through the ℓ_i in descending order. However, if we suppress isogeny point evaluations in the last iteration per loop, this means that these savings refer to small ℓ_i, and therefore the impact of this is rather small. Hence, we put a few large primes at the end of the loops, therefore requiring more computational effort for point multiplications, which is however in some situations outweighed by the larger savings from not evaluating points.

In this way, the best combination we found for CSIDH-512 is $\ell_1 = 349$, $\ell_2 = 347$, $\ell_3 = 337, \ldots, \ell_{69} = 3, \ell_{70} = 587, \ell_{71} = 373, \ell_{72} = 367, \ell_{73} = 359$, and $\ell_{74} = 353$, using SIMBA-5-11 and B from above, where the B_i are swapped accordingly to the ℓ_i.

Table 2. Performance of one class group action evaluation in CSIDH-512 with the mentioned parameters. All timings were measured on an Intel Core i7-6500 Skylake processor running Ubuntu 16.04 LTS, averaged over 1 000 runs.

Clock cycles $\times 10^8$	Wall clock time
3.145	121.3 ms

In Table 2, we give the cycle count and running time for the implementation using the parameters from above. The code is freely available at https://zenon.cs.hs-rm.de/pqcrypto/constant-csidh-c-implementation.

To give a comparison that mainly shows the impact of SIMBA and the different choice of B, we also ran the straightforward implementation according to Algorithm 2 with $B = [10, 10, \ldots, 10]$, also using Elligator. In this case, we measured 621.5 million clock cycles in the same setting as above.

Compared to the performance of the variable-time implementation from [14], the results from Table 2 mean a slowdown of factor 3.03. However, as mentioned, also the variable-time implementation can benefit from the optimizations from this paper, so this comparison should not be taken too serious.

7 Conclusion

We present the first implementation of CSIDH that prevents certain side-channel attacks, such as timing leakages. However, there might be more leakage models, depending on how powerful the attacker is. There is also more work to be done on making this implementation as efficient as possible. It may e.g. be possible to find a CSIDH-friendly prime p that allows for faster computations in $\mathbb{F}_p$.

Also the security features of CSIDH remain an open problem. More analysis on this is required, to show if the parameters are chosen correctly for the respective security levels.

We note that our results depend on the parameters from CSIDH-512. However, it is clear that the described optimizations can be adapted to other parameter sets and security levels as well.

Acknowledgments. This work was partially supported by Elektrobit Automotive, Erlangen, Germany. We thank Joost Renes for answering some questions during the preparation of this work, and the anonymous reviewers for their helpful and valuable comments.

A Implementation Results

We tested several parameters in a dynamical implementation, as explained in the paper. The setting is the same as in Sect. 6. For the parameters $B_0, \ldots, B_4$ we chose

$$B_0 = [10, 10, 10, ..., 10],$$

$$\begin{aligned} B_1 = [&1, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 8, 8, 8, 8, 8, 8, 8,\\ &12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 14, 14, 14, 14, 14,\\ &14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14,\\ &14, 14, 14, 14, 14, 14, 14, 14], \end{aligned}$$

$$B_2 = [5,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,8,8,8,8,8,8,8,$$
$$11,11,11,11,11,11,11,11,11,11,11,11,11,11,11,11,11,11,11,13,$$
$$13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,$$
$$13,13,13,13,13,13,13,13],$$

$$B_3 = [2,4,4,4,4,4,4,4,4,4,4,4,4,6,6,6,6,6,6,6,6,6,6,6,6,10,10,$$
$$10,10,10,10,10,10,10,10,10,16,16,16,16,16,16,16,16,16,16,16,$$
$$16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,$$
$$16,16,16,16,16,16,16,16,16,16],\ \text{and}$$

$$B_4 = [2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,12,20,20,$$
$$20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,$$
$$20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,$$
$$20,20,20,20,20,20,20,20,20,20].$$

In an earlier version of our implementation we measured many different combinations with different m and μ, running SIMBA-m-μ as described above, averaging the running time over 1 000 runs per parameter set, given in 10^6 clock cycles. For each B_i, we present the three best combinations we found in the Table 3.

Table 3. Performance of one class group action evaluation in CSIDH-512 with different combinations of parameters. All timings are given in 10^6 clock cycles, and were measured on an Intel Core i7-6500 Skylake processor running Ubuntu 16.04 LTS, averaged over 1 000 runs.

B	1st		2nd		3rd	
0	$\mu = 10$ $m = 5$	338.1	$\mu = 10$ $m = 6$	343.5	$\mu = 11$ $m = 5$	343.7
1	$\mu = 12$ $m = 4$	329.3	$\mu = 14$ $m = 4$	330.6	$\mu = 13$ $m = 4$	330.8
2	$\mu = 11$ $m = 5$	326.5	$\mu = 12$ $m = 5$	327.0	$\mu = 11$ $m = 4$	327.6
3	$\mu = 16$ $m = 4$	333.8	$\mu = 17$ $m = 4$	337.6	$\mu = 16$ $m = 3$	339.3
4	$\mu = 20$ $m = 3$	397.5	$\mu = 20$ $m = 4$	399.0	$\mu = 21$ $m = 3$	399.5

We further tried to rearrange the order of the primes ℓ_i in the different loops, as described in Sect. 6. However, the fastest parameter set from above was the best choice in all our tests.

B Algorithms

In this section we describe our constant-time algorithm, containing the optimizations from above. We split the application of SIMBA in two parts: SIMBA-I splits the isogeny computations in m batches, and SIMBA-II merges them after μ rounds. Note that in our implementation, it is actually not required to generate all the arrays from SIMBA-I.

Algorithm 5 shows the full class group action evaluation. Due to many loops and indices, it looks rather complicated. We recommend to additionally have a look at our implementation, provided in Sect. 6.

Algorithm 3. SIMBA-I.

Input : $e = (e_1, ..., e_n)$, $B = (B_1, ..., B_n)$, m.
Output: $e^i = (e_1^i, ..., e_n^i)$, $f^i = (f_1^i, ..., f_n^i)$, k_i for $i \in \{0, ..., m-1\}$.

Initialize $e^i = f^i = (0, 0, ..., 0)$ and $k_i = 4$ for $i \in \{0, ..., m-1\}$
foreach $i \in \{1, ..., n\}$ **do**
  $e_i^{i\%m} \leftarrow e_i$
  $f_i^{i\%m} \leftarrow B_i - e_i$
  foreach $j \in \{1, ..., m\}$ **do**
    if $j \neq (i\%m)$ **then**
      $k_i \leftarrow k_i \cdot \ell_i$

Algorithm 4. SIMBA-II.

Input : $e^i = (e_1^i, ..., e_n^i)$ and $f^i = (f_1^i, ..., f_n^i)$ for $i \in \{0, ..., m-1\}$, m.
Output: $e = (e_1, ..., e_n)$, $f = (f_1, ..., f_n)$, and k.

Initialize $e = f = (0, 0, ..., 0)$, and $k = 4$.
foreach $i \in \{1, ..., n\}$ **do**
  $e_i \leftarrow e_i^{i\%m}$
  $f_i \leftarrow f_i^{i\%m}$
  if $e_i = 0$ *and* $f_i = 0$ **then**
    $k \leftarrow k \cdot \ell_i$

Algorithm 5. Constant-time evaluation of the class group action in CSIDH-512.

Input : $a \in \mathbb{F}_p$ such that $E_a : y^2 = x^3 + ax^2 + x$ is supersingular, a list of integers $(e_1, ..., e_n)$ with $0 \leq e_i \leq B_i$ for all $i \leq n$, $B = (B_1, ..., B_n)$, m, μ.

Output: $a' \in \mathbb{F}_p$, the curve parameter of the resulting curve $E_{a'}$.

Run SIMBA-I(e, B, m).
foreach $i \in \{1, ..., \mu\}$ **do**
  foreach $j \in \{1, ..., m\}$ **do**
    Run Elligator to find a point P, where $y_P \in \mathbb{F}_p$.
    $P \leftarrow [k_j]P$
    $S = \{\iota \mid e_\iota^j \neq 0 \text{ or } f_\iota^j \neq 0\}$
    foreach $\iota \in S$ **do**
      $\alpha = \prod_{\kappa \in S, \kappa > \iota} \ell_\kappa$
      $K \leftarrow [\alpha]P$.
      if $K \neq \infty$ **then**
        if $e_\iota^j \neq 0$ **then**
          Compute a degree-ℓ_ι isogeny $\varphi : E_a \rightarrow E_{a'}$ with $ker(\varphi) = \langle K \rangle$.
          $a \leftarrow a'$, $P \leftarrow \varphi(P)$, $e_\iota^j \leftarrow e_\iota^j - 1$.
        else
          Compute a degree-ℓ_ι dummy isogeny:
          $a \leftarrow a$, $P \leftarrow [\ell_\iota]P$, $f_\iota^j \leftarrow f_\iota^j - 1$.
        if $e_\iota^j = 0$ *and* $f_\iota^j = 0$ **then**
          Set $k_j = k_j \cdot \ell_\iota$.
Run SIMBA-II(e^i and f^i for $i \in \{0, ..., m-1\}$, m).
while some $e_i \neq 0$ or $f_i \neq 0$ **do**
  Run Elligator to find a point P, where $y_P \in \mathbb{F}_p$.
  Set $P = (x : 1)$, $P \leftarrow [k]P$, $S = \{i \mid e_i \neq 0 \text{ or } f_i \neq 0\}$.
  foreach $i \in S$ **do**
    Let $m = \prod_{j \in S, j < i} \ell_i$.
    Set $K \leftarrow [m]P$.
    if $K \neq \infty$ **then**
      if $e_i \neq 0$ **then**
        Compute a degree-ℓ_i isogeny $\varphi : E_a \rightarrow E_{a'}$ with $ker(\varphi) = \langle K \rangle$.
        $a \leftarrow a'$, $P \leftarrow \varphi(P)$, $e_i \leftarrow e_i - 1$.
      else
        Compute a degree-ℓ_i dummy isogeny:
        $a \leftarrow a$, $P \leftarrow [\ell_i]P$, $f_i \leftarrow f_i - 1$.
      if $e_i = 0$ *and* $f_i = 0$ **then**
        Set $k = k \cdot \ell_i$.

References

1. Bernstein, D.J., Duif, N., Lange, T., Schwabe, P., Yang, B.Y.: High-speed high-security signatures. J. Cryptogr. Eng. **2**(2), 77–89 (2012)
2. Bernstein, D.J., Hamburg, M., Krasnova, A., Lange, T.: Elligator: elliptic-curve points indistinguishable from uniform random strings. In: Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security, pp. 967–980. ACM (2013)
3. Bernstein, D.J., Lange, T., Martindale, C., Panny, L.: Quantum circuits for the CSIDH: optimizing quantum evaluation of isogenies. Cryptology ePrint Archive, Report 2018/1059 (2018). https://eprint.iacr.org/2018/1059
4. Biasse, J.-F., Iezzi, A., Jacobson, M.J.: A note on the security of CSIDH. In: Chakraborty, D., Iwata, T. (eds.) INDOCRYPT 2018. LNCS, vol. 11356, pp. 153–168. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-05378-9_9
5. Bonnetain, X., Schrottenloher, A.: Quantum security analysis of CSIDH and ordinary isogeny-based schemes. Cryptology ePrint Archive, Report 2018/537 (2018). https://eprint.iacr.org/2018/537
6. Castryck, W., Lange, T., Martindale, C., Panny, L., Renes, J.: CSIDH: an efficient post-quantum commutative group action. In: Peyrin, T., Galbraith, S. (eds.) ASIACRYPT 2018. LNCS, vol. 11274, pp. 395–427. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-03332-3_15
7. Childs, A., Jao, D., Soukharev, V.: Constructing elliptic curve isogenies in quantum subexponential time. J. Math. Cryptol. **8**(1), 1–29 (2014)
8. Costello, C., Hisil, H.: A simple and compact algorithm for SIDH with arbitrary degree isogenies. In: Takagi, T., Peyrin, T. (eds.) ASIACRYPT 2017. LNCS, vol. 10625, pp. 303–329. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-70697-9_11
9. Costello, C., Longa, P., Naehrig, M.: Efficient algorithms for supersingular isogeny Diffie-Hellman. In: Robshaw, M., Katz, J. (eds.) CRYPTO 2016. LNCS, vol. 9814, pp. 572–601. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-53018-4_21
10. Couveignes, J.M.: Hard homogeneous spaces. Cryptology ePrint Archive, Report 2006/291 (2006). https://eprint.iacr.org/2006/291
11. De Feo, L., Kieffer, J., Smith, B.: Towards practical key exchange from ordinary isogeny graphs. In: Peyrin, T., Galbraith, S. (eds.) ASIACRYPT 2018. LNCS, vol. 11274, pp. 365–394. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-03332-3_14
12. Jao, D., et al.: Supersingular isogeny key encapsulation. Round 1 submission, NIST Post-Quantum Cryptography Standardization (2017)
13. Jao, D., De Feo, L., Plût, J.: Towards quantum-resistant cryptosystems from supersingular elliptic curve isogenies. J. Math. Cryptol. **8**(3), 209–247 (2014)
14. Meyer, M., Reith, S.: A faster way to the CSIDH. In: Chakraborty, D., Iwata, T. (eds.) INDOCRYPT 2018. LNCS, vol. 11356, pp. 137–152. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-05378-9_8
15. Montgomery, P.L.: Speeding the Pollard and elliptic curve methods of factorization. Math. Comput. **48**(177), 243–264 (1987)
16. Moody, D., Shumow, D.: Analogues of Vélu's formulas for isogenies on alternate models of elliptic curves. Math. Comput. **85**(300), 1929–1951 (2016)
17. National Institute of Standards and Technology (NIST): Submission requirements and evaluation criteria for the post-quantum cryptography standardization process (2016)

18. Renes, J.: Computing isogenies between montgomery curves using the action of (0, 0). In: Lange, T., Steinwandt, R. (eds.) PQCrypto 2018. LNCS, vol. 10786, pp. 229–247. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-79063-3_11
19. Rostovtsev, A., Stolbunov, A.: Public-key cryptosystem based on isogenies. Cryptology ePrint Archive, Report 2006/145 (2006). http://eprint.iacr.org/2006/145
20. Vélu, J.: Isogénies entre courbes elliptiques. C.R. Acad. Sci. Paris Série A **271**, 238–241 (1971)