CTIDH: constant-time CSIDH
An introduction to the math ideas

Jana Sotáková
email: ja.sotakova@gmail.com

University of Amsterdam/QuSoft

April 6, 2022

Abstract

This is a written exposition of our paper " CTIDH: constant-time CSIDH" [BBC+21] following the talks I gave at CHES 2021 (online) and at the ACCESS seminar on April 5, 2022.

Contents

CTIDH: Faster constant-time CSIDH CSIDH [CLM+18] is a post-quantum isogeny-based non-interactive key exchange protocol.

It uses a group action on a certain set of elliptic curves.

- Secret keys sampled from some keyspace $\mathsf{sk} \in \mathcal{K}$ give group elements,
- Public keys are elliptic curves obtained by evaluating the group action $\star$

$$\mathsf{pk} = \mathsf{sk} \star \mathsf{E}$$

CTIDH is a new keyspace and a new constant-time algorithm for the group action in CSIDH.

- constant-time claims verified using `valgrind`
- speedups compared to previous best work:
  - `CSIDH-512`: 438006 multiplications (best previous 789000); 125.53 million Skylake cycles (best previous more than 200 million).

1 CSIDH and the group action

Notation and generality We specialize everything to the CSIDH situation. Not all statements generalize to all (supersingular) elliptic curves, and we will try to make explicit which statements are general and which are specific to the constructions in CSIDH. The prime number p should be thought of as being of cryptographic size $p \approx 2^{512}$ (or larger). All numbers denoted ℓ or ℓ_i will always be small odd primes (typically, between $3 \leq \ell \leq 587$), and will always be assumed to divide $p+1$.

Choice of prime Start with a prime $p = 4 \cdot (\ell_1 \cdot \cdots \cdot \ell_n) - 1$ with $\ell_1, \ldots, \ell_n$ distinct odd primes.

Supersingular elliptic curves in Montgomery form

$$E_A : y^2 = x^3 + Ax^2 + x \quad A \in \mathbb{F}_p;$$

It is a fact that these curves always have $p+1$ points, and their groups of rational points are cyclic. You can take this as your definition (we will not use anything else from supersingularity): cyclic group and $p+1$ points.

Denote the set of such elliptic curves $\mathcal{E} = \{E_A : y^2 = x^3 + Ax^2 + x \text{ with } p+1 \text{ points over } \mathbb{F}_p\}$.

Properties

- ✓ Abelian group with an algebraic group law,
- ✓ Montgomery form enables x-only arithmetic,
- ! The group of rational points is cyclic:

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

  This is the reason for the very special choice of the prime p. Note that in particular, there is a unique subgroup of order ℓ_i for every $\ell_i \mid p+1$. One way to think of this: every point on $E(\mathbb{F}_p)$ can be written uniquely as a sum over the indices i of points of order dividing ℓ_i plus some point of order dividing 4.

Isogenies Whenever have a subgroup of E, we can construct an isogeny. For convenience (and the only case we need), we assume that this subgroup is generated by a point $P \in E(\mathbb{F}_p)$ of order ℓ, so we will get an ℓ-isogeny: a morphism of elliptic curves

$$\varphi : E_A \to E_{A'}$$

with kernel $\langle P \rangle$.

Unraveling the definition

- The isogeny φ is given by rational maps in the x, y of E with coefficient in $\mathbb{F}_p$;
- The isogeny φ is a group homomorphism: for all points Q and R we have

$$\varphi(Q+R) = \varphi(Q) + \varphi(R)$$

- the kernel of φ is the subgroup of E_A generated by P and has size ℓ, so it is tempting to think of isogenies as quotient maps that kill the subgroup $\langle P \rangle$, however, the better analogy is:
- ! the isogeny acts like a "power-ℓ-map" on $E(\mathbb{F}_p)$:

  if Q has order $\ell \cdot N$ (with $\gcd(\ell, N) = 1$), then $\varphi(Q)$ has order N on $E_{A'}$. If ℓ does not divide the order of the point Q, then $\varphi(Q)$ has the same order.

In the decomposition of points as in (1), we see that exactly the part of the point coming from the appropriate subgroup of points of order ℓ gets killed in the isogeny.

(In general, ℓ-torsion is two-dimensional, isomorphic to $\mathbb{Z}/\ell \times \mathbb{Z}/\ell$, and an ℓ-isogeny kernel is a one-dimensional subspace, and then ℓ-isogenies only remove the ℓ from the order from points whose "decomposition" contains points in the kernel).

Computing an isogeny from a point Suppose $P \in E$ is a point of order ℓ. We want to compute the isogeny with kernel $\langle P \rangle$:

$$\varphi : E_A \to E_{A'}$$

We can follow the following general recipe:

1. Collect the points $\{[i]P : i \in S\}$ for some index set S,
2. Compute the product
$$h(X) = \prod_{i \in S} (X - x([i]P)),$$
3. Recover A' from $h(X)$ (by evaluating the polynomial at some numbers, and computing a simple expression; this part is easy).

- Vélu's formulas [Vél71] use $S = \{1, 2, \ldots, \frac{\ell-1}{2}\}$;

  cost 6ℓ mult

- New $\sqrt{\text{élu}}$ formulas [BDFLS20] use $S = \{1, 3, 5, \ldots, \ell - 2\}$

  cost $\tilde{O}(\sqrt{\ell})$ mult
  The speedup comes from the fact that they do not process the big product $H(X)$ by multiplying the factors one by one, but by a baby-step-giant-step strategy.

This recipe allows us to recover the coefficient A' from the target curve, but the formulas can also be used to map points through the isogeny φ, at a cost of approx $1/3\times$ the total cost of this recipe. Soon we will want to map some points through the isogenies but we don't have to worry about the cost of that.

CSIDH magic Recall our prime of choice $p = 4 \cdot (\ell_1 \ldots \ell_n) - 1$ and that we are restricting ourselves to the elliptic curves in the set $\mathcal{E} = \{E_A : y^2 = x^3 + Ax^2 + x \text{ with } p + 1 \text{ points}\}$.

Every such curve $E_A \in \mathcal{E}$ has one distinguished[1] isogeny ℓ_i-isogeny: the one that comes from the unique subgroup of order ℓ_i in $E_A(\mathbb{F}_p)$.

So, for every $E_A \in \mathcal{E}$ and every $\ell \mid p + 1$, we can construct an ℓ-isogeny $\varphi : E_A \to E_{A'}$ using the points defined over $\mathbb{F}_p$:

$$E_A \longrightarrow E_{A'}$$

Claim We have $E_{A'} \in \mathcal{E}$, that is, computing these isogenies (with all the points in the kernel being defined over $\mathbb{F}_p$) preserves the properties we imposed on E_A: it is still a supersingular elliptic curve (true for all isogenies) and the group of rational points $E_{A'}(\mathbb{F}_p)$ is still cyclic (not true for all isogenies).

[1] Not a technical term, just for our purposes. The isogeny is still distinguished and can be picked out easily/algebraically from the set of all ℓ-isogenies.

Complex multiplication magic There is a finite abelian group G with a group action on $\mathcal{E}$ with the following properties:

- the action $E \mapsto g \star E$ is free and transitive action:

  For every E and E' in $\mathcal{E}$, there exists a unique element $g \in G$ sending E to E', that is,

  $$E' = g \star E.$$

- For every $\ell_i \mid p+1$, there exists a group element g_i such that if $\varphi : E_A \to E_{A'}$ is the distinguished ℓ_i-isogeny from before, then
  $$g_i \star E_A = E_{A'}.$$

- Because the group G is abelian, it only matters how many times we step in a particular direction, not the order in which we compute the isogenies.

The group in question is the ideal class group $\mathrm{Cl}(\mathbb{Z}[\sqrt{-p}])$, and the ring $\mathbb{Z}[\sqrt{-p}]$ is the $\mathbb{F}_p$-endomorphism ring of the curves in $\mathcal{E}$, so it's not out of the blue. But we don't need the details to understand CSIDH.

Exponent vector $(e_1, \ldots, e_n) \in \mathbb{Z}^n$ encodes how many times we perform each isogeny.

$$(e_1, \ldots, e_n): \qquad E_{A'} = \left(\prod_{i=1}^{n} g_i^{e_i} \right) \star E_A.$$

Going back with isogenies For every curve in $\mathcal{E}$ and every $\ell_i \mid p+1$, we have one ℓ_i-isogeny going forward, but also one going back:

$$E_A \xrightarrow{g_i} E_{A'} \xrightarrow{g_i^{-1}} E_A$$

Because acting with inverses should get us back$g_i^{-1} \star ((g_i) \star E_A) = (g_i^{-1} g_i) \star E_A = E_A$. The isogeny corresponding to g_i^{-1} is also easy to compute. One description of the isogeny corresponding to g_i is that its kernel are points of order ℓ with rational coefficients (x, y) for $x, y \in \mathbb{F}_p$, a description of the isogeny for g_i^{-1} can be that if we write $\mathbb{F}_{p^2} = \mathbb{F}_p(\sqrt{-1})$ (which we can because $p \equiv 3 \bmod 4$), then this isogeny is generated by points of order ℓ with coordinates $(x, \sqrt{-1}y)$ for $x, y \in \mathbb{F}_p$.

This is why we allow the exponent vectors $(e_1, \ldots, e_n) \in \mathbb{Z}^n$ and not just non-negative integers.

CSIDH key exchange Diffie-Hellman flow Alice and Bob agree on a starting curve $E_0 \in \mathcal{E}$:

1. Alice samples random exponent vector (e_i); Bob samples (f_i);
2. They compute action on E_0 as $E_A = (\prod g_i^{e_i}) \star E_0$ and $E_B = \left(\prod g_i^{f_i}\right) \star E_0$;
3. Exchange public keys: E_A, E_B;
4. They compute action on the curve just received:

$$\left(\prod g_i^{e_i}\right) \star E_B = \left(\prod g_i^{e_i + f_i}\right) \star E_0 = \left(\prod g_i^{f_i}\right) \star E_A$$

2 Constant-time evaluation

Constant-time evaluation Secret keys $(e_1, \ldots, e_n) \in \mathbb{Z}^n$ tell us how many different ℓ_i-isogenies we need to compute to evaluate the action

$$E_{A'} = \left(\prod_{i=1}^{n} g_i^{e_i} \right) \star E_A.$$

When evaluating the group action, every step (naively) consists of:

1. finding a point of order ℓ on some curve $E \in \mathcal{E}$,
2. an ℓ-isogeny computation from E.

However, if we just apply Vélu's formulas directly, each of the ℓ_i-isogenies takes a different amount of time. So, somebody timing the computation and seeing which steps occur, will be able to read off which isogenies we computed ... which is exactly the secret key. So, we want to compute the isogeny group action in a way that would not leak this information. A bit more formal definition follows below:

Constant-time evaluation of the group action 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.

Computing the group action Let's compute one step in a bit more detail: we want to compute the group action $E_{A'} = g_i \star E_A$ as an ℓ_i-isogeny:

1. find a point P of order ℓ_i on E_A:
   (a) generate a point T of order $p+1$ on E_A,
   (b) multiply $P = [\frac{p+1}{\ell_i}]T$.
2. Compute the ℓ_i-isogeny $\varphi : E_A \to E_{A'}$ with kernel P:
   (a) enumerate the multiples $[i]P$ of the point P for $i \in S$,
   (b) construct a polynomial $h(X) = \prod_{i \in S}(X - x([i]P))$,
   (c) Compute the coefficient A' from $h(X)$.

We need to comment on Step 1a. We do not know how to sample random points directly, however, we do know how to sample points directly: either just sample a random x and hope that $x^3 + Ax^2 + x$ is a square, or use more sophisticated ways like the Elligator map. But once we have a random point T, it will not have exact point of order $p+1$. This failure is a problem we will ignore today completely, but has to be taken into account when implementing isogenies – until we find a way to reliably sample points of a certain order.

In Step 1b, once we have a random point of order $p+1$, we need to compute one scalar multiplication by a number close to $p+1$; the cost of this step absolutely dominates the isogeny computation, because it's on the order of $\approx 11 \log_2(p)$ multiplication in $\mathbb{F}_p$. For `CSIDH-512`, this is about 5500 multiplications, whereas the largest $\ell_i = 587$ and computing the 587-isogeny closer to 2000 multiplications (and 587 itself is an outlier, larger than other primes, so most isogeny computations are a lot cheaper). We therefore want to compute as few of these big scalar multiplications as possible.

Amortize the cost Let us go trough an example of how to compute 3 different isogenies using one big scalar multiplication (and some (a lot smaller) overhead). Think of this as an exercise, try to fill in the correct orders for the various points (if you're uncertain, see the same computation below 2)

Take the exponent vector $(1, 1, 1, 0, \ldots, 0)$: compute ℓ_i-isogenies for $\ell_1 = 3$ and $\ell_2 = 5$ and $\ell_3 = 7$:

1. Find a suitable point:
   (a) Generate a random point T of order $p + 1$,
   (b) Compute $T_1 = \left[\frac{p+1}{3 \cdot 5 \cdot 7}\right] T$ has exact order ___
2. Compute the isogenies:
   (a) 3-isogeny:
      i. Compute $P_1 = [5 \cdot 7]T_1$ has order ___
      ii. Use P_1 to construct 3-isogeny φ_1,
      iii. Point $T_2 = \varphi_1(T_1)$ has order ___ on the new curve,
   (b) 5-isogeny:
      i. Compute $P_2 = [7]T_2$ has order ___,
      ii. Construct 5-isogeny φ_2 with kernel P_2,
      iii. The point $T_3 = \varphi_2(T_2)$ has order ___ on the new curve,
   (c) 7-isogeny: construct the isogeny φ_3 with kernel $P_3 = T_3$ which has order ___

Towards atomic blocks Now we modify the procedure above to evaluate the action by the exponent vector $(1, 0, 1, 0, \ldots, 0)$: compute ℓ_i-isogenies for $\ell_1 = 3$ and $\ell_3 = 7$ but no 5-isogeny:

1. Find a suitable point:
   (a) Generate a random point T of order $p + 1$,
   (b) Compute $T_1 = \left[\frac{p+1}{3 \cdot 5 \cdot 7}\right] T$ has exact order $3 \cdot 5 \cdot 7$,
2. Compute the isogenies:
   (a) 3-isogeny:
      i. Compute $P_1 = [5 \cdot 7]T_1$ has order 3,
      ii. Use P_1 to construct 3-isogeny φ_1,
      iii. Point $T_2 = \varphi_1(T_1)$ has order $5 \cdot 7$ on the new curve,
   (b) No 5-isogeny:
      i. Compute the isogeny as before but throw away the results,
      ii. Adjust to code to always compute $[5]T_2$,
      iii. The point $T_3 = [5]T_2$ has order 7 on the same curve,
   (c) 7-isogeny: construct the isogeny φ_3 with kernel $P_3 = T_3$.

If you believe that this can be made to work to not leak information about whether or not you computed a 5-isogeny in the process, you can quickly extend this to a procedures that allows you to compute either of the $3, 5$ or 7-isogenies without leaking (timing) information on which of the isogenies were actually computed.

3 Atomic blocks

The discussion above lead us to the following (simplified) definition of atomic blocks:

Definition 3.1 (Atomic Blocks)**.** *Let $I \subset \{1, \ldots, n\}$ be a subset of indices and write $I = (i_1, \ldots, i_k)$. An atomic block of length k is a probabilistic algorithm α_I:*

- *taking inputs A and $\epsilon \in \{0,1\}^k$,*
- *returning $A' \in \mathbb{F}_p$ such that $E_{A'} = (\prod_{j=1}^{k} g_{i_j}^{\epsilon_j}) \star E_A$,*
- *the time distribution of α_I is independent of ϵ.*

On the previous slide, we saw an atomic block α_I with $I = (1, 2, 3)$ that computes

$$E_{A'} = g_1^{\epsilon_1} g_2^{\epsilon_2} g_3^{\epsilon_3} \star E_A$$

for $(\epsilon_1, \epsilon_2, \epsilon_3) \in \{0,1\}^3$ without leaking timing information about $(\epsilon_1, \epsilon_2, \epsilon_3)$.

Why atomic blocks? Because:

1. Previous CSIDH implementations are using atomic blocks implicitly;
2. Simpler framework to compute the group action:
   (a) split the computation into atomic blocks independent of the secret;
   (b) make sure each atomic block is constant-time.

4 New Keyspace

Keyspace Remember what we want: for $(e_1, \ldots, e_n) \in \mathbb{Z}^n$, evaluate the group action

$$E_{A'} = \left(\prod_{i=1}^{n} g_i^{e_i} \right) \star E_A.$$

- We want to put constraints on what exponent vectors we can expect: $(e_1, \ldots, e_n)$ are sampled from some keyspace $\mathcal{K} \subset \mathbb{Z}^n$;
- This keyspace needs to be large enough $\#\mathcal{K} \approx 2^{256}$ to prevent

Example of keyspaces

1. Original CSIDH [CLM+18]: $|e_i| \leq m$ for all i with $(2m+1)^n \approx 2^{256}$, so $m = 5$ for `CSIDH-512`;
2. [MCR19] use $0 \leq e_i \leq 10$ for `CSIDH-512`;
3. you can allow the m_i to vary for efficiency, such as in [CDRH20].

Batching Take `CSIDH-512` prime $p = 4 \cdot (3 \cdot 5 \cdot \cdots \cdot 373 \cdot 587) - 1$. Consider the exponent vector:

primes	3	5	7	11	13	17	19	23	29	31	...
exponent vector	1	-2	0	3	-1	1	0	2	-1	0	...

And note that this vector also lives in the space given by the constraints:

1. $|e_1| + |e_2| + |e_3| \leq 3$,
2. $|e_4| + |e_5| + |e_6| \leq 5$,
3. $|e_7| + \cdots + |e_{10}| \leq 5$, ...

New batching key space For B batches: For $N \in \mathbb{Z}_{>0}^B$ and $m \in \mathbb{Z}_{\geq 0}^B$, we define

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

That is, we split the primes into B batches, and for each batch, we bound the total number of isogenies we are willing to compute for primes from this batch.

Example for 6 primes (Only considering non-negative exponents for simplicity.) Take 6 primes and split them into batches of 3 primes each, and allow up to 3 isogenies per batch: we want vectors

$$(\{e_1, e_2, e_3\}\{e_4, e_5, e_6\})$$

constrained by

$$0 \leq e_i, \qquad e_1 + e_2 + e_3 \leq 3, \qquad e_4 + e_5 + e_6 \leq 3.$$

It is fun counting practice to check that there are $20 \cdot 20 = 400$ such vectors.

To get as many vectors with " balanced exponents" (bounding each entry individually with similar bounds), we could take for instance bounds $(3, 3, 3, 3, 2, 2)$, giving us 324 keys.

(As a short sidenote, the most efficient way to have as many keys with as small entries as possible is to put all the primes in one batch, and consider the L_1-ball, see [NOTT20].)

Constant time However, we also want to compute things in constant time. Most previous constant-time approaches essentially compute the same number of ℓ_i-isogenies no matter what the exponent vector was, so in the balanced exponent example would compute $3+3+3+3+2+2 = 16$ isogenies. However, in CTIDH, we push the constant-time computation into the batch: all the isogenies in the batch will be computed with the same code, and hence not leaking information on the degree - any more information than what we already know from the definition of the keyspace. This includes some overhead, so our isogenies are more expensive. But the speedup of CTIDH comes from the fact, that we still only need to compute $3 + 3 = 6$ isogenies.

Atomic blocks for batches Suppose we have batches $\{3, 5, 7\}, \{11, 13, 17\}, \ldots$ And we want to compute one 5-isogeny and one 11-isogeny, i.e. exponent vector $(0, 1, 0, 1, 0, 0, 0, \ldots)$. Let's try to copy the atomic blocks from 2

1. Find a suitable point:
   (a) Generate a random point T of order $p + 1$,
   (b) Compute $T_1 = \left[\frac{p+1}{(3\cdot5\cdot7)(11\cdot13\cdot17)}\right] T$ has order $(3 \cdot 5 \cdot 7)(11 \cdot 13 \cdot 17)$.
2. Compute the isogenies:
   (a) $\{3, 5, 7\}$-isogeny:
      i. Compute $P_1 = [(11 \cdot 13 \cdot 17)]T_1$ has order $(3 \cdot 5 \cdot 7)$,
      ii. Use $[3 \cdot 7]P_1$ of order 5 to construct 5 -isogeny φ_1,
      iii. Point $T_2 = [3 \cdot 7]\varphi_1(T_1)$ has order $11 \cdot 13 \cdot 17$ on the new curve,
   (b) $\{11, 13, 17\}$-isogeny:
      i. Compute $P_2 = [13 \cdot 17]T_2$ has order 11,
      ii. Construct 11-isogeny φ_2 with kernel P_2.

It is easy to check that the only steps that leak information on the primes in the batch are the small scalar multiplications $[3\cdot7]P_1, [13\cdot17]T_2$ (which are easily fixed to be in constant time), but most importantly the actual computations of 5- and 11-isogenies. This is what we fix next.

5 New algorithm and Matryoshka Isogeny

Let's examine the procedure to compute the 11-isogeny:

1. enumerate the multiples $[i]P$ of the point P for $i \in S$ with $S = \{1, 2, \ldots, 5\}$
2. construct $h(X) = \prod_{i=1}^{5}(x - x([i]P))$,
3. Compute the coefficient A' from $h(X)$.

Now compute the ~~11~~ 13-isogeny:

1. enumerate the multiples $[i]P$ of the point P for $i \in S$ with $S = \{1, 2, \ldots, 5, 6\}$
2. construct $h(X) = \prod_{i=1}^{5}(x - x([i]P)) \cdot (x - x([6]P))$,
3. Compute the coefficient A' from $h(X)$.

And now the ~~11~~ ~~13~~ 17-isogeny

1. enumerate the multiples $[i]P$ of the point P for $i \in S$ with $S = \{1, 2, \ldots, 5, 6, 7, 8\}$
2. construct $h(X) = \prod_{i=1}^{5}(x - x([i]P)) \cdot (x - x([6]P)) \cdot (x - x([7]P))(x - x([8]P))$,
3. Compute the coefficient A' from $h(X)$.

We see that the code to compute a 17-isogeny naturally includes everything we need to compute a 11- or 13-isogeny. So, with small overhead, we can use the same code to compute isogenies in the same batch. This also justifies why we split our primes in batches: we don't want to pay the price for the 587-isogeny for all the small isogenies. Carefully selecting the batches has major impact on performance, and is an optimization problem we haven't been able to solve in full generality.

Matryoshka isogenies

- We can compute the isogeny for any prime in the batch with the same code,
- at the cost of computing isogeny for the largest prime,
- requiring using dummy computations.

This was know for Vélu formulas [BLMP19] but our new observation is that this also works for $\sqrt{\text{élu}}$ from [BDFLS20], and we newly used this for batching.

Matryoshka for $\sqrt{\text{élu}}$ We need to evaluate the polynomial

$$h(X) = \prod_{i \in S}(X - x([i]P)),$$

for $S = \{1, 3, \ldots, \ell - 2\}$.

Visual explanation for 29 and 31 In √élu, we process the product as a " square-box", which is processed using baby-step-giant-step procedure, and a final multiplication by the factors that did not fit into the optimal box. So, we use the procedure for the smallest prime, and add the factors we would need for the largest prime to multiply with as more factors outside of the box:

$$\left|\begin{array}{c|c|c|c} 1 & 9 & 17 & 25 \\ 3 & 11 & 19 & 27 \\ 5 & 13 & 21 & \color{red}{29} \\ 7 & 15 & 23 & \end{array}\right| \longleftrightarrow \left(\prod_{\text{box from 1 to 23}} (X - x([i]P))\right) \times (X - x([25]P)(X - x([27]P))\color{red}{(X - x([29]P))}$$

6 Implementation

Finally, a few brief comments on implementation.

Selection of the parameters We use a greedy algorithm to find efficient batching in Table 1

- For every batch configuration (number of batches, bounds of each batch), we can estimate the cost of the group action evaluation.
- Adaptively change batch configuration to find one with smaller cost (and large enough keyspace).

See in the table that the resulting batches have primes of approximately the same size, so paying the prime for one isogeny at the cost of the largest prime in the batch, is not so much of an overhead.

batch	size	primes	bound
1	2	3, 5	10
2	3	7, 11, 13	14
3	4	17, 19, 23, 29	16
4	4	31, 37, 41, 43	17
5	5	47, 53, 59, 61, 67	17
6	5	71, 73, 79, 83, 89	17
7	6	97, 101, 103, 107, 109, 113	18
8	7	127, 131, 137, 139, 149, 151, 157	18
9	7	163, 167, 173, 179, 181, 191, 193	18
10	8	197, 199, 211, 223, 227, 229, 233, 239	18
11	8	241, 251, 257, 263, 269, 271, 277, 281	18
12	6	283, 293, 307, 311, 313, 317	13
13	8	331, 337, 347, 349, 353, 359, 367, 373	13
14	1	587	1

Table 1: *Batches for the primes used in CSIDH-512*

valgrind constant time verification You can check your code manually for constant-time-ness, or you can use `valgrind` to do it for you, checking that there is no secret data flow influencing branches or indices.

Speedups, comparison to previous works We are almost twice as fast as compared to the previous constant-time implementations, see Table 2.

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	[CCJR20]
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	[ACR20]
512	256	2	129.64	330966	121787	497476	452752	453269	new
512	256	2	218.42	665876	189377	691231	855253	851939	[CDRH20]
512	256	2	238.51	632444	209310	704576	841754	835121	[HLKA20]
512	256	2	239.00	657000	210000	691000	867000	859550	[CCC+19]
512	256	2	—	732966	243838	680801	976804	962076	[OAYT19]
512	256	2	395.00	1054000	410000	1053000	1464000	1434650	[MCR19]
1024	256	1	469.52	287739	87944	486764	375683	382432	new
1024	256	1	—	552000	133000	924000	685000	704600	[ACR20]
1024	256	2	511.19	310154	99371	521400	409525	415721	new

Table 2: **pub***: size of p;* **priv***: size of the keyspace;* **DH** *1: group action evaluation,* **DH** *2: group action evaluation and public key validation;* **Mcyc** *millions of cycles on a 3GHz Intel Xeon E3-1220 v5 (Skylake) CPU with Turbo Boost disabled; "***M***" multiplications; "***S***" squarings; "***a***" additions; "*$1,1,0$*" and "*$1,0.8,0.05$*" combinations of* **M**, **S**, *and* **a**.

7 Summary

CTIDH is:

- New keyspace for CSIDH,
- New constant-time algorithm to evaluate the group action in CSIDH,
- Formalization of atomic blocks to compute the isogeny group action,
- constant-time verification using `valgrind`,
- speed records,

Find the article and the code at

https://ctidh.isogeny.org/

References

[ACR20] 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`.

[BBC+21] Gustavo Banegas, Daniel J. Bernstein, Fabio Campos, Tung Chou, Tanja Lange, Michael Meyer, Benjamin Smith, and Jana Sotáková. CTIDH: faster constant-time CSIDH. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2021(4):351–387, August 2021. Artifact available at `https://artifacts.iacr.org/tches/2021/a20`.

[BDFLS20] Daniel J. Bernstein, Luca De Feo, Antonin Leroux, and Benjamin Smith. Faster computation of isogenies of large prime degree, 2020. `https://eprint.iacr.org/2020/341`.

[BLMP19] Daniel J. Bernstein, Tanja Lange, Chloe Martindale, and Lorenz Panny. Quantum circuits for the CSIDH: optimizing quantum evaluation of isogenies, 2019. `https://eprint.iacr.org/2018/1059`.

[CCC+19] 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, 2019. https://eprint.iacr.org/2019/837.

[CCJR20] 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.

[CDRH20] Jesús-Javier Chi-Domínguez and Francisco Rodríguez-Henríquez. Optimal strategies for CSIDH, 2020. https://eprint.iacr.org/2020/417.

[CLM+18] Wouter Castryck, Tanja Lange, Chloe Martindale, Lorenz Panny, and Joost Renes. CSIDH: an efficient post-quantum commutative group action, 2018. https://eprint.iacr.org/2018/383.

[HLKA20] Aaron Hutchinson, Jason T. LeGrow, Brian Koziel, and Reza Azarderakhsh. Further optimizations of CSIDH: A systematic approach to efficient strategies, permutations, and bound vectors, 2020. https://eprint.iacr.org/2019/1121.

[MCR19] Michael Meyer, Fabio Campos, and Steffen Reith. On Lions and Elligators: An efficient constant-time implementation of CSIDH, 2019. https://eprint.iacr.org/2018/1198.

[NOTT20] 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.

[OAYT19] Hiroshi Onuki, Yusuke Aikawa, Tsutomu Yamazaki, and Tsuyoshi Takagi. (Short paper) A faster constant-time algorithm of CSIDH keeping two points, 2019. https://eprint.iacr.org/2019/353.

[Vél71] Jacques Vélu. Isogénies entre courbes elliptiques, 1971. https://gallica.bnf.fr/ark:/12148/cb34416987n/date.