Applying Castryck-Decru Attack on the Masked Torsion Point Images SIDH variant

Jesús-Javier Chi-Domínguez[1]

Cryptography Research Center, Technology Innovation Institute, Abu Dhabi, UAE
jesus.dominguez@tii.ae

Keywords: Cryptanalysis · Castryck-Decru Attack · Isogeny-based cryptography · Masked-SIDH

Abstract. This paper illustrates that masking the torsion point images does not guarantee Castryck-Decru attack does not apply. Our experiments over SIDH primes hint that any square root concerning the Weil pairing on the masked public key helps to recover Bob's private key via the Castryck-Decru attack.

1 Introduction

Castryck and Decru provided in [2] a heuristically polynomial SIDH key-recovery Attack, which relies on the knowledge of

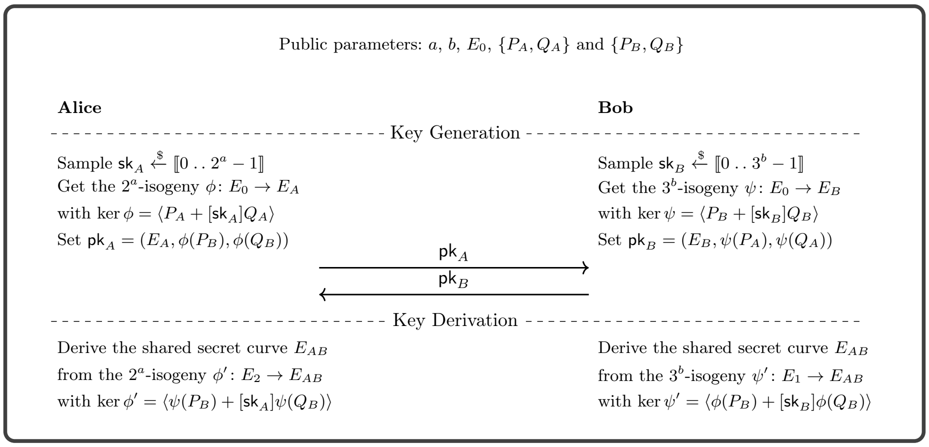
- The isogeny degree;
- The image of coprime torsion points; and
- The endomorphism ring of the isogeny domain curve.

Maino and Martindale in [7] gave an algorithm that works without knowing the endomorphism ring of the domain curve. In contrast, Robert demonstrated the existence of a polynomial key-recovery attack on SIDH [10]. The results in [7,10] are still theoretical, but [2] shared a Magma code of their attack improved by the sagemath code of Oudompheng and Pope in [9].

To mitigate the Castryck-Decru attack, Fouotsa and Moriya independently proposed solutions for SIDH. Fouotsa suggested masking the torsion point images [4] and Moriya hiding the isogeny degree [8]. This works only analyze Fouotsa's countermeasure given in [4].

2 SIDH framework

We strongly recommend that the readers go through [5,3,1] for details concerning SIDH. Let us first center on the following SIDH setup. Let $\mathbb{F}_{p^2} = \mathbb{F}_p[i]/(i^2+1)$ be a quadratic field extension of $\mathbb{F}_p$ along with $p = 2^a 3^b - 1$. We set as starting supersingular curve $E_0 \colon y^2 = x^3 + 6x^2 + x$ [1]. Let $\{P_A, Q_A\}$ a basis for the 2^a-torsion subgroup $E_0[2^a]$, and $\{P_B, Q_B\}$ for the 3^b-torsion subgroup $E_0[3^b] = \langle P_B, Q_B \rangle$.



Public parameters: a, b, E_0, $\{P_A, Q_A\}$ and $\{P_B, Q_B\}$

Alice **Bob**

Key Generation

Sample $\mathsf{sk}_A \xleftarrow{\$} [\![0 \mathinner{..} 2^a - 1]\!]$
Get the 2^a-isogeny $\phi\colon E_0 \to E_A$
with $\ker \phi = \langle P_A + [\mathsf{sk}_A] Q_A \rangle$
Set $\mathsf{pk}_A = (E_A, \phi(P_B), \phi(Q_B))$

Sample $\mathsf{sk}_B \xleftarrow{\$} [\![0 \mathinner{..} 3^b - 1]\!]$
Get the 3^b-isogeny $\psi\colon E_0 \to E_B$
with $\ker \psi = \langle P_B + [\mathsf{sk}_B] Q_B \rangle$
Set $\mathsf{pk}_B = (E_B, \psi(P_A), \psi(Q_A))$

$\xrightarrow{\mathsf{pk}_A}$
$\xleftarrow{\mathsf{pk}_B}$

Key Derivation

Derive the shared secret curve E_{AB}
from the 2^a-isogeny $\phi'\colon E_2 \to E_{AB}$
with $\ker \phi' = \langle \psi(P_B) + [\mathsf{sk}_A] \psi(Q_B) \rangle$

Derive the shared secret curve E_{AB}
from the 3^b-isogeny $\psi'\colon E_1 \to E_{AB}$
with $\ker \psi' = \langle \phi(P_B) + [\mathsf{sk}_B] \phi(Q_B) \rangle$

Fig. 1: General description of the SIDH protocol.

3 Sketch of the Castryck-Decru Attack

We encourage readers to carefully check [2] for details concerning the Castryck-Decru attack. We let assume Eve wants to recover the secrete 3^b-isogeny $\psi \colon E_0 \to E_B$ given $P = \psi(P_A)$ and $Q = \psi(Q_A)$

Eve starts by fixing two integers $\alpha_1 \geq 0$ and $\beta_1 \geq 1$ such that $c = 2^{a-\alpha_1} - 3^{b-\beta_1} > 0$ splits as a product of prime factors, each congruent to 1 modulo 4. Next, she proceeds by guessing the first 3^{β_1}-isogeny $\kappa_1 \colon E_0 \to E_1$. To decide whether κ_1 is a correct guess, she performs the following: let $\mathbf{i} \colon (x, y) \mapsto (-x, iy)$ be the distorsion map on $E_{-1} \colon y^2 = x^3 + x$, then she

1. Sets $P_1 = \kappa_1([2^{\alpha_1}]P_A)$ and $Q_1 = \kappa_1([2^{\alpha_1}]Q_A)$. Notice $E_1[2^{a-\alpha_1}]\langle P_1, Q_1 \rangle$;
2. Constructs the isogeny $\tau_1 \colon E_0 \to C_1$ with kernel $\gamma_0(\ker \kappa_1)$ where $\gamma_0 = [u] + [v] \circ 2\mathbf{i} \colon E_0 \to E_0$ for some integers u, v as in [2, Remark 1] [2];
3. Computes the c-isogeny $\gamma_1 = \frac{\tau_1 \circ \gamma_0 \circ \widehat{\kappa}_1}{3^{\beta_1}} \colon E_1 \to C_1$, and evaluates $\kappa_1(P_A)$ and $\kappa_1(Q_A)$ to get

[1] We choose the same E_0 as in SIKE proposal [1], but it can be any different curve with known endomorphism ring.

[2] Here, $2\mathbf{i}$ refers $\widehat{\rho} \circ \mathbf{i} \circ \rho \colon E_0 \to E_0$ with $\rho \colon E_0 \to E_{-1}$ being the 2-isogeny that connects E_0 and E_{-1}.

$$P_{c_1} = \gamma_1(P_A) = [2^{\alpha_1}]\tau_1(\gamma_0(P_A)) \text{ and}$$
$$Q_{c_1} = \gamma_1(Q_A) = [2^{\alpha_1}]\tau_1(\gamma_0(Q_A));$$

4. If the Richelot $(2^{a-\alpha_1}, 2^{a-\alpha_1})$-isogeny $\Psi_1 \colon C_1 \times E_B \to A_1$ with kernel
$$\langle (P_{c_1}, [2^{\alpha_1}]P), (Q_{c_1}, [2^{\alpha_1}]Q) \rangle$$
has codomain A_1 being a product of elliptic curves, then the guess κ_1 is correct. If not, she tries with a different 3^{β_1}-isogeny κ_1 and returns to Item 1.
5. At this point, she already gets $\psi = \psi_1 \circ \kappa_1$ with unknown $3^{b-\beta_1}$-isogeny $\psi_1 \colon E_1 \to E_B$. She continues by recovering the next $3^{\beta_2-\beta_1}$-isogeny $\kappa_2 \colon E_1 \to E_2$ with $\beta_2 > \beta_1$ and $\alpha_2 > \alpha_1$ and having the same conditions for $c_2 = 2^{a-\alpha_2} - 3^{b-\beta_2}$ as c_1. She applies the same methodology as below, but this time using the knowledge of κ_1. She iteratively recovers the 3^b-isogeny $\psi \colon E_0 \to E_B$ until she reaches the desired degree 3^b. That is, Eve recovers the secret 3^b-isogeny by finding small 3^{β_j}-isogeny factors of ψ. Notice that we implicitly assume $\alpha_j < a$ and $\beta_j < b$, so the last step reduces to a brute-force search on small values of β_j.

4 The masked SIDH from [4]

The countermeasure from [4] proposes masking the torsion point images in pk_A and pk_B as

- Alice samples a random integer $r_A \in [\![0 \mathrel{..} 3^b - 1]\!]$ coprime to 3. Next, she sets a her public key $\mathsf{pk}_A = (E_A, [r_A]\phi(P_B), [r_A]\phi(Q_B))$.
- Bobs samples a random integer $r_B \in [\![0 \mathrel{..} 2^a - 1]\!]$ coprime to 2. Next, he sets a his public key $\mathsf{pk}_B = (E_B, [r_B]\psi(P_A), [r_B]\psi(Q_A))$.

[4] claims the above masking proposal is enough for the Castryck-Decru attack to fail; we need precisely the image of the torsion points to mount the Castryck-Decru attack. In concrete, an attacker can get both $(r_A)^2 \bmod 3^b$ and $(r_B)^2 \bmod 2^b$ via discrete logarithms concerning the following Weil pairing equations

$$e_{3^b}\big([r_A]\phi(P_B), [r_A]\phi(Q_B)\big) = \left(e(P_B, Q_B)^{2^a}\right)^{(r_A)^2} \text{ and}$$
$$e_{2^a}\big([r_B]\psi(P_A), [r_B]\psi(Q_A)\big) = \left(e(P_A, Q_A)^{3^b}\right)^{(r_B)^2}.$$

On that basis, [4] suggests a large prime such that the number of square roots for $(r_A)^2$ and $(r_B)^2$ is about 2^λ, thus finding the correct r_A and r_B ensure λ-bits of security.

5 Applying Castryck-Decru Attack over SIDH-like primes

For simplicity, we center on analyzing Bob's public key (just as in [2]), but it easily extends to Alice's scenario. This section justifies and experimentally illustrates that any square root of $r = (r_B)^2$ helps to make the Castryck-Decru works.

Let r' be a square root of r, and let $\vartheta_{\zeta'}\colon P \mapsto [\zeta']\psi(P)$ be the isogeny being $\zeta' = \tilde{r}r_B$ with $\tilde{r}$ the multiplicative inverse of r' modulo 2^a. Next, we let $\widehat{\vartheta}_{\zeta'}\colon P \mapsto [\zeta']\widehat{\psi}(P)$ describes the dual isogeny of $\vartheta_{\zeta'}$. Consequently, we get $\vartheta_{\zeta'} \circ \widehat{\vartheta}_{\zeta'} = \left[3^b{\zeta'}^2\right]$ and $\widehat{\vartheta}_{\zeta'} \circ \vartheta_{\zeta'} = \left[3^b{\zeta'}^2\right]$. Notice that ${\zeta'}^2 = 1 \bmod 2^a$, so let us assign the square root of unity $\zeta = (\zeta' 3^b\mu_b + 2^a\mu_a)$ modulo $(p+1)$ with $3^b\mu_b = 1 \bmod 2^a$ and $2^a\mu_a = 1 \bmod 3^b$, which coincides with ζ' modulo 2^a.

Corollary 1. *Over $\mathbb{F}_{p^2}$, the isogeny $\vartheta_\zeta\colon P \mapsto [\zeta]\psi(P)$ and its dual $\widehat{\vartheta}_\zeta\colon P \mapsto [\zeta]\widehat{\psi}(P)$ satisfy $\vartheta_\zeta \circ \widehat{\vartheta}_\zeta = \left[3^b\right] = \widehat{\vartheta}_\zeta \circ \vartheta_\zeta$.*

Corollary 1 implicitly says $\vartheta_\zeta\colon E_0 \to E_B$ and its dual $\widehat{\vartheta}_\zeta\colon E_B \to E_0$ looks like 3^b-isogenies over $\mathbb{F}_{p^2}$. We point out that ϑ_ζ and $\widehat{\vartheta}_\zeta$ does not behave as 3^b-isogenies over extensions fields of $\mathbb{F}_{p^2}$ but we do not care about that for attacking the masked SIDH construction. In particular, locally over $\mathbb{F}_{p^2}$, we get a high chance that the Castryck-Decru attack succeeds for ϑ_ζ.

Now we sketch how to recover Bob's secret 3^b-isogeny $\psi\colon E_0 \to E_B$. Given the masked public key $\mathsf{pk}_B = (E_B, [r_B]\psi(P_A), [r_B]\psi(Q_A))$, we proceed as follows:

1. Parse $(E_B, P', Q') \leftarrow \mathsf{pk}_B$;
2. Get $r = (r_B)^2$ from the Weil pairing equation in Section 4;
3. Compute any square root r' of r;
4. Calculate the multiplicative inverse $\tilde{r}$ of r' modulo 2^a;
5. Set $\mathsf{pk}'_B = (E_B, [\tilde{r}]P', [\tilde{r}]Q')$;
6. Feed the Castryck-Decru attack with pk'_B to find the isogeny $\vartheta_\zeta\colon P \mapsto [\zeta]\psi(P)$.
7. Derive sk_B from ϑ_ζ.

Notice that the square roots of unity modulo 2^a are $\zeta_{\pm 1} = \pm 1$ and $\zeta_{\pm 2} = (\pm 1 + 2^{a-1})$. Then, the new input order-2^a points $P \leftarrow [\tilde{r}]'$ and $Q \leftarrow [\tilde{r}]Q'$ required in Item 4 of the Castryck-Decru attack satisfy

$$\begin{aligned}
[2^{\alpha_j}]P &= [2^{\alpha_j}]([\tilde{r}]P') = [2^{\alpha_j}]([\zeta]\psi(P_A)) = \pm[2^{\alpha_j}]\psi(P_A) \text{ and}\\
[2^{\alpha_j}]Q &= [2^{\alpha_j}]([\tilde{r}]Q') = [2^{\alpha_j}]([\zeta]\psi(Q_A)) = \pm[2^{\alpha_j}]\psi(Q_A)
\end{aligned}$$

for each $a > \alpha_j > 0$. Now, the sign ± does not affect the computations of the Richelot $(2,2)$-isogeny chain since the kernels have torsion two. In other words, since the Castryck-Decru attack only uses the image points when computing the Richelot $(2, 2)$-isogeny, the attack should recover the secret isogeny $\theta_\zeta \colon E_0 \to E_B$ without issue. We validate the above procedure using the sagemath code from [9] along with the below code [3] and illustrate that we do not need the correct square root r_B of $r = (r_B)^2$. Our experiments randomly generate instances with the following SIDH parameters:

- Baby SIDHp64 from [9]: $a = 33$ and $b = 19$,
- $IKEp217: $a = 110$ and $b = 67$;
- SIKEp377 from [6]: $a = 191$ and $b = 117$;
- SIKEp546 from [6]: $a = 273$ and $b = 172$;

6 Applying Castryck-Decru Attack over CSIDH-like primes

Now, let us replace the prime $p = 2^a 3^b - 1$ by $p = 4AB - 1$ with $(A, B) = 1$, and the isogeny degrees 2^a and 3^b by A and B, respectively. We assume $(A, 2) = 1$ and $(B, 2) = 1$, and AB splits as the product of different small odd primes ℓ_j's. Then, similar reasoning from above holds. Let r' be a square root of r, and let $\vartheta_{\zeta'} \colon P \mapsto [\zeta']\psi(P)$ be the isogeny being $\zeta' = \tilde{r} r_B$ with $\tilde{r}$ the multiplicative inverse of r' modulo A. Next, we set the square root of unity $\zeta = (\zeta'(4B)\mu_b + 4A\mu_a + AB\mu)$ modulo $(p + 1)$ with

$$\begin{aligned}(4B)\mu_b &= 1 \bmod A,\\ (4A)\mu_a &= 1 \bmod B \text{ and}\\ (AB)\mu &= 1 \bmod 4.\end{aligned}$$

which coincides with ζ' modulo A.

Corollary 2. *Over $\mathbb{F}_{p^2}$, the isogeny $\vartheta_\zeta \colon P \mapsto [\zeta]\psi(P)$ and its dual $\widehat{\vartheta}_\zeta \colon P \mapsto [\zeta]\widehat{\psi}(P)$ satisfy $\vartheta_\zeta \circ \widehat{\vartheta}_\zeta = [B] = \widehat{\vartheta}_\zeta \circ \vartheta_\zeta$.*

6.1 Square roots of unity modulo A on the (ℓ, ℓ)-isogeny kernels

Let us write $A = \prod_{j=1}^{k} \ell_j$ with ℓ_j a small odd prime, and let μ_j the multiplicative inverse of $\frac{A}{\ell_j}$ modulo ℓ_j. This time the root of unity ζ modulo A has the shape

$$\zeta = \sum_{j=1}^{k} (-1)^{e_j} \left(\frac{A}{\ell_j}\right) \mu_j \bmod A$$

[3] We take the script `baby_SIDH.sage` from [9] as a baseline and replace it according to SIKE parameters. Our code is shared in Appendix A.

where $e_j \in \{0, 1\}$ for each $j \coloneqq 1, \ldots, k$. Now, let us discuss below how to decompose any order-A point R as the sum of order-ℓ_j points. The idea is to use such point decomposition to illustrate the shape of each (ℓ_j, ℓ_j)-isogeny required in the Castrcyk-Decru attack. For any order-A elliptic curve point R, we have

$$R = \sum_{j=1}^{k} [\mu_j] R_j \quad \text{and} \quad [\zeta]R = \sum_{j=1}^{k} [(-1)^{e_j} \mu_j] R_j \quad \text{with} \quad R_j = \left[\frac{A}{\ell_j}\right] R.$$

Furthermore, due to the nature of $C \times E$, the order-(A, A) points (P_c, P) and (Q_c, Q) on $C \times E$ satisfy

$$\begin{aligned}
(P_c, P) &= \sum_{j=1}^{k} \big([\mu_j]P_{c,j}, [\mu_j]P_j\big) = \sum_{j=1}^{k} [\mu_j]\big(P_{c,j}, P_j\big),\\
(P_c, [\zeta]P) &= \sum_{j=1}^{k} \big([\mu_j]P_{c,j}, [(-1)^{e_j}\mu_j]P_j\big) = \sum_{j=1}^{k} [\mu_j]\big(P_{c,j}, [(-1)^{e_j}]P_j\big),\\
(Q_c, Q) &= \sum_{j=1}^{k} \big([\mu_j]Q_{c,j}, [\mu_j]Q_j\big) = \sum_{j=1}^{k} [\mu_j]\big(Q_{c,j}, Q_j\big) \text{ and}\\
(Q_c, [\zeta]Q) &= \sum_{j=1}^{k} \big([\mu_j]Q_{c,j}, [(-1)^{e_j}\mu_j]Q_j\big) = \sum_{j=1}^{k} [\mu_j]\big(Q_{c,j}, [(-1)^{e_j}]Q_j\big),
\end{aligned}$$

where $P_j = \left[\frac{A}{\ell_j}\right] P$, $P_{c,j} = \left[\frac{A}{\ell_j}\right] P_c$, $Q_j = \left[\frac{A}{\ell_j}\right] Q$ and $Q_{c,j} = \left[\frac{A}{\ell_j}\right] Q_c$. It is worth highlighting that the generators $\big(P_{c,j}, \pm P_j\big)$ and $\big(Q_{c,j}, \pm Q_j\big)$ have order ℓ_j. We illustrate the above with the sagemath code given in Appendix B. Additionally, for each (ℓ_j, ℓ_j)-isogeny factor Ψ_j of $\Psi \colon C \times E \to A$ we have

$$\begin{aligned}
\ker \Psi_1 &= \langle \big(P_{c,1}, P_1\big), \big(Q_{c,1}, Q_1\big) \rangle,\\
\ker \Psi_2 &= \langle \Psi_1\left(\big(P_{c,2}, P_2\big)\right), \Psi_1\big((Q_{c,2}, Q_2)\big) \rangle,\\
&\vdots\\
\ker \Psi_j &= \langle \Psi_{j-1} \circ \cdots \circ \Psi_1\left(\big(P_{c,j}, P_j\big)\right), \Psi_{j-1} \circ \cdots \circ \Psi_1\big((Q_{c,j}, Q_j)\big) \rangle \text{ for } j \geq 3.
\end{aligned}$$

In constrast, concerning the isogeny chain determined by $\langle \big(P_c, [\zeta]P\big), \big(Q_c, [\zeta]Q\big) \rangle$, we get a slightly different sequence of kernels for each (ℓ_j, ℓ_j)-isogeny factor Ψ'_j of $\Psi' \colon C \times E \to A$:

$$\begin{aligned}
\ker \Psi'_1 &= \langle \big(P_{c,1}, [(-1)^{e_1}]P_1\big), \big(Q_{c,1}, [(-1)^{e_1}]Q_1\big)\rangle,\\
\ker \Psi'_2 &= \langle \Psi'_1\left(\big(P_{c,2}, [(-1)^{e_2}]P_2\big)\right), \Psi'_1\big((Q_{c,2}, [(-1)^{e_2}]Q_2)\big)\rangle,\\
&\;\;\vdots\\
\ker \Psi'_j &= \langle \Psi'_{j-1} \circ \cdots \circ \Psi'_1\left(\big(P_{c,j}, [(-1)^{e_j}]P_j\big)\right), \Psi_{j-1} \circ \cdots \circ \Psi_1\big((Q_{c,j}, [(-1)^{e_j}]Q_j)\big)\rangle.
\end{aligned}$$

The only difference between the kernels $\ker \Psi_j$ and $\ker \Psi'_j$ is a (possible) shifted sign of the second coordinate points P_j and Q_j.

6.2 Open questions

Could the attack from section 5 work for other primes like $p = 4AB - 1$?? Or do we need precisely the square root r_B? What can we say between the kernels of the (ℓ_j, ℓ_j)-isogenies Ψ_j and Ψ'_j from section 6.1?

Acknowledgements We thank Benjamin Wesolowski, Luca De Feo, and Peter Kutas for their comments and discussion on the applicability of masked SIDH parameters with $p = 4AB - 1$. We also thank Tako Boris Fouotsa for his comments about the shape of the roots of unity modulo powers of two.

A Code concerning SIDH-like primes

```
import public_values_aux
from public_values_aux import *

load('castryck_decru_shortcut.sage')

# SIKEpXXX parameters
a, b = 33, 19
# a, b = 110, 67
# a, b = 191, 117
# a, b = 273, 172

# Set the prime, finite fields and starting curve
# with known endomorphism
p = 2^a*3^b - 1
public_values_aux.p = p

Fp2.<i> = GF(p^2, modulus=x^2+1)
R.<x> = PolynomialRing(Fp2)

E_start = EllipticCurve(Fp2, [0,6,0,1,0])
E_start.set_order((p+1)^2) # Speeds things up in Sage

# Generation of the endomorphism 2i
two_i = generate_distortion_map(E_start)

# Generate public torsion points, for SIKE implementations
# these are fixed but to save loading in constants we can
# just generate them on the fly
P2, Q2, P3, Q3 = generate_torsion_points(E_start, a, b)
check_torsion_points(E_start, a, b, P2, Q2, P3, Q3)
```

```
# Generate Bob's key pair
bob_private_key, EB, PB, QB = gen_bob_keypair(E_start, b, P2, Q2, P3, Q3)
solution = Integer(bob_private_key).digits(base=3)

print(f"Running the attack against SIDHp{p.bit_length()} parameters, which has a prime:
↪  2^{a}*3^{b} - 1")
print(f"If all goes well then the following digits should be found: {solution}")

def mask(pk):
    (EB, PB, QB) = pk
    N = 2^a
    rB = 2 * randint(1, N // 2) + 1
    print(f'mask:\t{rB}')
    return EB, rB * PB, rB * QB

# ==================================
# =====  ATTACK  ===================
# ==================================
def unmask(pk):
    (EB, PB, QB) = pk
    e = P2.weil_pairing(Q2, 2^a)
    e_= PB.weil_pairing(QB, 2^a)
    N = e.order()
    r = discrete_log(e_, e^(3^b))
    assert (e ^ (r * (3^b))) == e_
    assert N == e_.order()
    N = 2^a
    R = IntegerModRing(N)
    square_roots = R(r).sqrt(all=True)
    other = int(square_roots[0])
    dec, tilde, _  = xgcd(other, N)
    assert dec == 1
    tilde = int(R(tilde))
    assert tilde * other % (N) == 1
    PB_ = tilde * PB
    QB_ = tilde * QB
    assert e^(3^b) == (PB_).weil_pairing(QB_, 2^a)
    print(f'unmask:\t{other}')
    return EB, PB_, QB_


EB, PB, QB = mask((EB, PB, QB))
EB, PB, QB = unmask((EB, PB, QB))


def RunAttack(num_cores):
        return CastryckDecruAttack(E_start, P2, Q2, EB, PB, QB, two_i, num_cores=num_cores)


if __name__ == '__main__' and '__file__' in globals():
    if '--parallel' in sys.argv:
        # Set number of cores for parallel computation
        num_cores = os.cpu_count()
        print(f"Performing the attack in parallel using {num_cores} cores")
    else:
        num_cores = 1
    recovered_key = RunAttack(num_cores)
```

B Code concerning split of points

The following code takes the prime field characteristic P and uses an instance generated through the Magma code from Appendix C.

```
load('speedup.sage')
load('richelot_aux.sage')
```

```
def SplitCoeff(d, f):
    assert d == prod(f)
    mu_list = []
    for j in range(0, len(f), 1):
        tmp, mu, _ = xgcd(d // f[j], f[j])
        if mu < 0:
            mu += f[j]
        mu_list.append(mu)

    return mu_list

def SplitPoint(P, Zero, d):
    f = list(dict(factor(d)).keys())
    e = list(dict(factor(d)).values())
    assert e == [1] * len(f)
    out = [(d // fi)*P for fi in f]
    assert [out[j] != Zero for j in range(0, len(out), 1)] == [True]*len(out)
    assert [f[j] * out[j] == Zero for j in range(0, len(out), 1)] == [True]*len(out)
    mus = SplitCoeff(d, f)
    assert sum([mus[j] * out[j] for j in range(0, len(out), 1)]) == P
    return out, mus

def FromProdToJac_(C, E, P_c, Q_c, P, Q, d):
    # The gluing step only requires all the 2-order points on E and C. So, we can assume d =
    ↪  4 and not 2A
    Fp2 = C.base()
    Rx.<x> = Fp2[]
    p = Fp2.characteristic()

    assert (E.a2()^2 - Fp2(4)).is_square()
    discE = (E.a2()^2 - Fp2(4)).square_root()
    P2 = E((-E.a2() + discE) / Fp2(2), Fp2(0))
    Q2 = E(Fp2(0), Fp2(0))
    assert P2 * 2 == E(0)
    assert Q2 * 2 == E(0)
    assert P2 != Q2
    assert (C.a2()^2 - Fp2(4)).is_square()
    discC = (C.a2()^2 - Fp2(4)).square_root()
    P_c2 = C((-C.a2() + discC) / Fp2(2), Fp2(0))
    Q_c2 = C((-C.a2() - discC) / Fp2(2), Fp2(0))
    assert P_c2 * 2 == C(0)
    assert Q_c2 * 2 == C(0)
    assert P_c2 != Q_c2

    a1, a2, a3 = P_c2[0], Q_c2[0], (P_c2 + Q_c2)[0]
    b1, b2, b3 = P2[0], Q2[0], (P2 + Q2)[0]

    # Compute coefficients
    M = Matrix(Fp2, [
        [a1*b1, a1, b1],
        [a2*b2, a2, b2],
        [a3*b3, a3, b3]])
    R, S, T = M.inverse() * vector(Fp2, [1,1,1])
    RD = R * M.determinant()
    da = (a1 - a2)*(a2 - a3)*(a3 - a1)
    db = (b1 - b2)*(b2 - b3)*(b3 - b1)

    s1, t1 = - da / RD, db / RD
    s2, t2 = -T/R, -S/R

    a1_t = (a1 - s2) / s1
    a2_t = (a2 - s2) / s1
    a3_t = (a3 - s2) / s1
    h = s1 * (x^2 - a1_t) * (x^2 - a2_t) * (x^2 - a3_t)

    H = HyperellipticCurve(h)
    J = H.jacobian()
    def isogeny(pair):
        # Argument members may be None to indicate the zero point.

        # The projection maps are:
        # H->C: (xC = s1/x²+s2, yC = s1 y)
        # so we compute Mumford coordinates of the divisor f^-1(P_c): a(x), y-b(x)
        Pc, P = pair
        if Pc:
            xPc, yPc = Pc.xy()
            JPc = J([s1 * x^2 + s2 - xPc, Rx(yPc / s1)])
        # Same for E
        # H->E: (xE = t1 x² + t2, yE = t1 y/x^3)
        if P:
            xP, yP = P.xy()
            JP = J([(xP - t2) * x^2 - t1, yP * x^3 / t1])
        if Pc and P:
            return JPc + JP
        if Pc:
            return JPc
        if P:
            return JP

    print(f'\nBelow divisors belongs to the {H}')
    imPcP = isogeny((P_c, P))
    imPcP_ = isogeny((P_c, -P))
    print(f'\nimage of (Pc, P):\t{imPcP}')
    print(f'image of (Pc,-P):\t{imPcP_}')
    imQcQ = isogeny((Q_c, Q))
    imQcQ_ = isogeny((Q_c, -Q))
    print(f'\nimage of (Qc, Q):\t{imQcQ}')
    print(f'image of (Qc,-Q):\t{imQcQ_}')

    # Testing the splitting of points
    print('\nTesting the splitting of points\n')
    P_split, mus = SplitPoint(P, E(0), d)
    Q_split, mus_ = SplitPoint(Q, E(0), d)
    assert mus == mus_
    Pc_split, mus_ = SplitPoint(P_c, C(0), d)
    assert mus == mus_
    Qc_split, mus_ = SplitPoint(Q_c, C(0), d)
    assert mus == mus_
    imPcP_split = sum([mus[j] * isogeny((Pc_split[j], P_split[j])) for j in range(0,
    ↪  len(P_split), 1)])
    imQcQ_split = sum([mus[j] * isogeny((Qc_split[j], Q_split[j])) for j in range(0,
    ↪  len(Q_split), 1)])

    # Next asserts ensure we can split the kernels
    assert imPcP == imPcP_split
    assert imQcQ == imQcQ_split
    print(f'\nmu:\t{mus}')

    return h, imPcP[0], imPcP[1], imQcQ[0], imQcQ[1], isogeny

def test_FromProdToJac_():
    A = 304181303036879580815246611364368665235
    B = 46720814763454310373877236230100511563
    p = 4 * A * B - 1
    assert p.is_prime()
    k = GF(p**2, 'i', modulus=(x**2 + 1))
    i = k.gen()
    E = EllipticCurve(k, [0,
    (22628445289720396015652504674262311080845170988693192554195338479216713661679*i +
    3548400250796405259659630926596137085861982499350216491360605866873121722709),
    0,1,0])
    print(f'\nE is the {E}')
    # assert E.is_supersingular()
    # assert E.order() == (p+1)^2
    C = EllipticCurve(k, [0,
    (47566395385068891477156433209440162682985673554429053315995405204509950791733*i +
    45070910329265928958784248077907949974876514709136778422443975313711797347028),
    0,1,0])
    print(f'\nC is the {C}')
    # assert C.is_supersingular()
    # assert C.order() == (p+1)^2
    P = E(46601371436796787519179671500211852468586004922742506216989479543070271121088*i +
    9038122453348361741342678115173274557533008358808786629724886424505404536190,
    36814531135553385706829640816717226740793744297931897833596617270882568232614*i +
    44768257047520934886148492192261615939563313371996076894910174268391817250908)
    Q = E(44392969488800573703015711637855332109150565959789633865611796037839066460797*i +
    29012576981305969592091765764930647916848745556244732440926189488003402310599,
    18767939631305838744674601761655967177669568697891504735257671883811839665657*i +
    21245520863548722816721891422117698177863233432025179530912280495105600201764)
    Pc = C(25390509085786004688431775825614249130522865850652636936195699548741655305506*i +
    52098503968600138046171684526582969374851993795516383164559733448719087806608,
    56641888411200390057480982927512507277521736066804492830886545179434755247178*i +
    2499776690612137393769360465579262011208011783801802048594268738621458695254)
    Qc = C(437920780641478088634208588869491832654728840937148770770526731530139220949850*i +
    14575999843548358700519095323092735936411581482627508656629583497907057330375,
    34412852023221841304193069449735485979806109664113485980838987917120986159871*i +
    39265667720540940274276373415796873728671613105770613789631958516185507796258)
    return FromProdToJac_(C, E, Pc, Qc, P, Q, A)

test_FromProdToJac_()
```

C Code to generate a SIDH public key with field characteristic of a C-SIDH prime.

```
clear;
A := 304181303036879580815246611364368665235;
B := 46720814763454310373877236230100511563;
p := 4 * A * B - 1;
assert IsPrime(p);
fp := GF(p);
_<x> := PolynomialRing(fp);
fq<i> := ext<fp | x^2 + 1>;
assert i^2 + 1 eq 0;
_<x> := PolynomialRing(fq);

// +++++++++++++++++
coeff := function(A)
    return fq ! 2 * (2*A[1] - A[2]) / A[2];
end function;

// +++++++++++++++++++
xDBL := function(P, A)
    t_0 := P[1] - P[2];
    t_1 := P[1] + P[2];
    t_0 := t_0 ^ 2;
    t_1 := t_1 ^ 2;
    Z := A[2] * t_0;
    X := Z * t_1;
    t_1 := t_1 - t_0;
    t_0 := A[1] * t_1;
    Z := Z + t_0;
    Z := Z * t_1;
    return [X, Z];
end function;   // 4M + 2S + 4a

// +++++++++++++++++++++++
xADD := function(P, Q, PQ)

    a := P[1] + P[2];
    b := P[1] - P[2];
    c := Q[1] + Q[2];
    d := Q[1] - Q[2];
    a := a * d;
    b := b * c;
    c := a + b;
    d := a - b;
    c := c ^ 2;
    d := d ^ 2;
    X := PQ[2] * c;
    Z := PQ[1] * d;

    return [X, Z];
end function;   // 4M + 2S + 6a

// +++++++++++++++++++++++++++++++++++++++++++++++
CrissCross := function(alpha, beta, gamma, delta)
    t_1 := alpha * delta;
    t_2 := beta * gamma;
    return [t_1 + t_2, t_1 - t_2];
end function;   // 2M + 2 a

// ++++++++++++++++++++++++++++++
KernelPoints := function(P, A, l)
    d := (l - 1) div 2;
    K := [ [fq!0, fq!0] : j in [1 .. d] ];
    K[1] := P;
    K[2] := xDBL(P, A);                         // 4M + 2S + 4a
    for j:=3 to d do
        K[j] := xADD(K[j-1], K[1], K[j - 2]);   // 4M + 2S + 6a
    end for;
    return K;
end function;   // 4(d - 1)M + 2(d - 1)S + 2(3d - 4)a

// +++++++++++++++++++++++
xEVAL := function(P, K, l)
    d := (l - 1) div 2;
    Q := [P[1] + P[2], P[1] - P[2]];
    R := CrissCross(K[1][1], K[1][2], Q[1], Q[2]);
    for j:=2 to d do
        T := CrissCross(K[j][1], K[j][2], Q[1], Q[2]);
        R := [T[1] * R[1], T[2] * R[2]];
    end for;
    return [P[1] * (R[1]^2), P[2] * (R[2]^2)];
end function;   // (4d)M + 2S + 2(d + 1)a

// +++++++++++++++++++++++
xISOG := function(A, K, l)
    d := (l - 1) div 2;
    pi := [fq!1, fq!1];
    sg := [fq!0, fq!1];

    S := [];
    for j := 1 to d do
        t := (K[j][1] * K[j][2]);
        s_1 := (K[j][1] + K[j][2]);
        s_2 := (K[j][1] - K[j][2]);
        S[j] := [s_1, s_2];
        sg := [ s_1 * s_2 * sg[2] + sg[1] * t, sg[2] * t];
        pi := [pi[1] * K[j][1], pi[2] * K[j][2]];
    end for;

    A24 := 2*(2*A[1] - A[2]);

    A24 := (pi[1]^2) * ( (A24 * sg[2]) - (6 * sg[1] * A[2]) );
    C24 := (pi[2]^2) * A[2] * sg[2];

    return [A24 + 2*C24, 4*C24], S;
```

```
end function;   // (7d + 5)M + 2S + 3(d + 1)a

// ++++++++++++++++++++++++++
splits := function(P, seq)
    n := #seq;
    //print "...", n, seq;
    if n eq 1 then
        //print "--", seq;
        return [P];
    elif n gt 0 then
        h := n div 2;
        if h gt 0 then
            // ---
            LEFT_SUBSEQ := [];
            P_RIGHT := P;
            for j := 1 to h do
                P_RIGHT := seq[j] * P_RIGHT;
                LEFT_SUBSEQ[j] := seq[j];
            end for;
            // ---
            RIGHT_SUBSEQ := [];
            P_LEFT := P;
            for j := (h+1) to n do
                P_LEFT := seq[j] * P_LEFT;
                RIGHT_SUBSEQ[j-h] := seq[j];
            end for;
            // -
            return $$(P_LEFT, LEFT_SUBSEQ) cat $$(P_RIGHT, RIGHT_SUBSEQ);
        end if;
        return [];
    end if;
end function;


// ++++++++++++++++++++++++++++
isfull_order := function(split)
    return &and[s[2] ne fp!0 : s in split];
end function;

// ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
function OddCyclicSumOfSquares(n, factexpl, provide_own_fac)
  if provide_own_fac then
    fac := factexpl;
  else
    fac := Factorization(n);
  end if;
  if {f[1] mod 4 : f in fac} ne {1} then
    return false, 0, 0;
  else
    Z<I> := GaussianIntegers();
    prod := 1;
    for f in fac do
      p := f[1];
      repeat
        z := Random(GF(p));
      until z^((p-1) div 2) eq -1;
      z := Integers() ! (z^((p-1) div 4)); // square root of -1
      prod *:= GCD(Z ! p, Z ! z + I)^f[2];
    end for;
    u := [Integers() ! e : e in Eltseq(prod)];
    if IsOdd(u[1]) then u1 := u[1]; u2 := u[2] div 2; else u1 := u[2]; u2 := u[1] div 2; end
    ↪  if;
    return true, Abs(u1), Abs(u2);
  end if;
end function;

// ++++++++++++++++++++++++++++++++++++++++++++
function FromProdToJac(C, E, P_c, Q_c, P, Q, d)
```

```
Fp2 := BaseField(C);
R<x> := PolynomialRing(Fp2);

Pc2_and_Qc2 := Roots(x^2 + (Coefficients(C)[2]) * x + 1);
P_c2 := C![Pc2_and_Qc2[1,1], 0];
Q_c2 := C![0, 0];
P2_and_Q2 := Roots(x^2 + (Coefficients(E)[2]) * x + 1);
P2 := E![P2_and_Q2[1,1], 0];
Q2 := E![0, 0];

alp1 := P_c2[1];
alp2 := Q_c2[1];
alp3 := (P_c2 + Q_c2)[1];
bet1 := P2[1];
bet2 := Q2[1];
bet3 := (P2 + Q2)[1];
a1 := (alp3 - alp2)^2/(bet3 - bet2) + (alp2 - alp1)^2/(bet2 - bet1) + (alp1 - alp3)^2/(bet1
↪  - bet3);
b1 := (bet3 - bet2)^2/(alp3 - alp2) + (bet2 - bet1)^2/(alp2 - alp1) + (bet1 - bet3)^2/(alp1
↪  - alp3);
a2 := alp1*(bet3 - bet2) + alp2*(bet1 - bet3) + alp3*(bet2 - bet1);
b2 := bet1*(alp3 - alp2) + bet2*(alp1 - alp3) + bet3*(alp2 - alp1);
Deltalp := (alp1 - alp2)^2*(alp1 - alp3)^2*(alp2 - alp3)^2;
Deltbet := (bet1 - bet2)^2*(bet1 - bet3)^2*(bet2 - bet3)^2;
A := Deltbet*a1/a2;
B := Deltalp*b1/b2;

h := - (A*(alp2 - alp1)*(alp1 - alp3)*x^2 + B*(bet2 - bet1)*(bet1 - bet3)) *
       (A*(alp3 - alp2)*(alp2 - alp1)*x^2 + B*(bet3 - bet2)*(bet2 - bet1)) *
       (A*(alp1 - alp3)*(alp3 - alp2)*x^2 + B*(bet1 - bet3)*(bet3 - bet2));

t1 := -(A/B)*b2/b1;
t2 := (bet1*(bet3 - bet2)^2/(alp3 - alp2) + bet2*(bet1 - bet3)^2/(alp1 - alp3) + bet3*(bet2
↪  - bet1)^2/(alp2 - alp1))/b1;
s1 := -(B/A)*a2/a1;
s2 := (alp1*(alp3 - alp2)^2/(bet3 - bet2) + alp2*(alp1 - alp3)^2/(bet1 - bet3) + alp3*(alp2
↪  - alp1)^2/(bet2 - bet1))/a1;

Uff<u0, u1, v0, v1> := FunctionField(Fp2, 4);
A4<U0, U1, V0, V1> := AffineSpace(Fp2, 4); U := Parent(U0);

u0tilde := 1/u0;
u1tilde := u1/u0;
v0tilde := (u1*v0 - u0*v1)/u0^2;
v1tilde := (u1^2*v0 - u0*v0 - u0*u1*v1)/u0^2;

lamb1 := - (Deltbet/A^3)*v1tilde/(s1*u1tilde);
lamb2 := - (Deltalp/B^3)*v1/(t1*u1);

x1 := lamb1^2 + alp1 + alp2 + alp3 - s1*(u1tilde^2 - 2*u0tilde) - 2*s2;
y1 := -lamb1*(x1 - s2 + (u0tilde*v1tilde - u1tilde*v0tilde)*s1/v1tilde);

x2 := lamb2^2 + bet1 + bet2 + bet3 - t1*(u1^2 - 2*u0) - 2*t2;
y2 := -lamb2*(x2 - t2 + (u0*v1 - u1*v0)*t1/v1);

eq1 := U ! Numerator(x1 - P_c[1]);
eq2 := U ! Numerator(y1 - P_c[2]);
eq3 := U ! Numerator(x2 - P[1]);
eq4 := U ! Numerator(y2 - P[2]);
eq5 := 2*V0^2 - 2*V0*V1*U1 + V1^2*(U1^2 - 2*U0)
       - 2*Coefficient(h, 0)
       - (-U1)*Coefficient(h, 1)
       - (U1^2 - 2*U0)*Coefficient(h, 2)
       - (-U1^3 + 3*U0*U1)*Coefficient(h, 3)
       - (U1^4 - 4*U1^2*U0 + 2*U0^2)*Coefficient(h, 4)
       - (-U1^5 + 5*U1^3*U0 - 5*U1*U0^2)*Coefficient(h, 5)
       - (U1^6 - 6*U1^4*U0 + 9*U1^2*U0^2 - 2*U0^3)*Coefficient(h, 6);
```

```
  V := Scheme(A4, [eq1, eq2, eq3, eq4, eq5]);

  // point with zero coordinates probably correspond to "extra" solutions, we should be left
  ↪  with 4 sols
  // (code may fail over small fields)

  realsols := [];
  for D in Points(V) do
    Dseq := Eltseq(D);
    if not 0 in Dseq then
      realsols cat:= [Dseq];
    end if;
  end for;

  // print "Number of inverse images found:", #realsols, "(hopefully 4)";

  J := Jacobian(HyperellipticCurve(h));
  sol := Random(realsols);
  D := elt<J | x^2 + sol[2]*x + sol[1], sol[4]*x + sol[3]>;
  imPcP := 2*D;

  // now for (Q_c, Q)

  eq1 := U ! Numerator(x1 - Q_c[1]);
  eq2 := U ! Numerator(y1 - Q_c[2]);
  eq3 := U ! Numerator(x2 - Q[1]);
  eq4 := U ! Numerator(y2 - Q[2]);
  V := Scheme(A4, [eq1, eq2, eq3, eq4, eq5]);
  realsols := [];
  for D in Points(V) do
    Dseq := Eltseq(D);
    if not 0 in Dseq then
      realsols cat:= [Dseq];
    end if;
  end for;
  // print "Number of inverse images found:", #realsols, "(hopefully 4)";
  sol := Random(realsols);
  D := elt<J | x^2 + sol[2]*x + sol[1], sol[4]*x + sol[3]>;
  imQcQ := 2*D;

  assert imPcP * d eq J!0;
  assert imQcQ * d eq J!0;
  printf "[ ] First (2,2)-isogeny with codomain: %o\n", HyperellipticCurve(h);
  printf "[ ] > D_1:\t%o\n", imPcP;
  printf "[ ] > D_2:\t%o\n", imQcQ;

  return HyperellipticCurve(h), imPcP, imQcQ;
end function;

// ==================================
E_0 := EllipticCurve(x^3 + 6*x^2 + x);
printf "\n%o\n", E_0;

// NAIVE GENERATION OF AUTOMORPHISM 2i
E1728, phi := IsogenyFromKernel(E_0, x);
for iota in Automorphisms(E1728) do
  P := Random(E1728);
  if iota(iota(P)) eq -P then
    two_i := phi*iota*DualIsogeny(phi);
    break;
  end if;
end for;

infty := E_0 ! 0;

// Generators of the torsion-A and torsion-B subgroups
Afactors := [a[1] : a in Factorization(A)];
Bfactors := [b[1] : b in Factorization(B)];
```

```
repeat
  PA := (4*B)*Random(E_0);
until isfull_order(splits(PA, Afactors));
repeat
  QA := (4*B)*Random(E_0);
  wPQA := WeilPairing(PA, QA, A);
until &and[wPQA^(A div a) ne 1 : a in Afactors];
assert isfull_order(splits(PA, Afactors));
assert A * PA eq infty;
assert isfull_order(splits(QA, Afactors));
assert A * QA eq infty;
print "PA := E_0!", Eltseq(PA)[1 .. 2], ";";
print "QA := E_0!", Eltseq(QA)[1 .. 2], ";";
repeat
  PB := (4*A)*Random(E_0);
until isfull_order(splits(PB, Bfactors));
repeat
  QB := (4*A)*Random(E_0);
  wPQB := WeilPairing(PB, QB, B);
until &and[wPQB^(B div b) ne 1 : b in Bfactors];
assert isfull_order(splits(PB, Bfactors));
assert B * PB eq infty;
assert isfull_order(splits(QB, Afactors));
assert B * QB eq infty;
print "PB := E_0!", Eltseq(PB)[1 .. 2], ";";
print "QB := E_0!", Eltseq(QB)[1 .. 2], ";";

// Computing Bob's public key : (EB, P, Q)
EB := [fq!6+2, fq!4]; // Corresponding with A=6
E := EllipticCurve(x^3 + coeff(EB)*x^2 + x);
assert  E eq E_0;
Bobskey := Integers()!(IntegerRing(B)!987 / IntegerRing(B)!610);    // We use as secret an approximation to the golden ration;

K := PB + Bobskey*QB;
assert Order(K) eq B;

print "\n";
scalar := B;
K := [K[1], fq!1];
P := [PA[1], fq!1];
Q := [QA[1], fq!1];
PQ := [(PA-QA)[1], fq!1];
for n:=1 to (#Bfactors - 1) do
    printf ">> Computing %o-isogeny codomain (public key computation)\n", Bfactors[n];
    good, Kt := IsPoint(E, K[1]/K[2]);
    assert good;
    assert isfull_order(splits(Kt, Bfactors[n .. #Bfactors]));
    Kt := (scalar div Bfactors[n]) * Kt;
    Kps := KernelPoints([Kt[1], fq!1], EB, Bfactors[n]);
    EB, Kps := xISOG(EB, Kps, Bfactors[n]);
    K := xEVAL(K, Kps, Bfactors[n]);
    P := xEVAL(P, Kps, Bfactors[n]);
    Q := xEVAL(Q, Kps, Bfactors[n]);
    PQ := xEVAL(PQ, Kps, Bfactors[n]);
    E := EllipticCurve(x^3 + coeff(EB)*x^2 + x);
    assert Random(E) * (p + 1) eq infty;
    good, Pt := IsPoint(E, P[1]/P[2]);
    assert good;
    assert isfull_order(splits(Pt, Afactors));
    assert Pt * A eq infty;
    good, Qt := IsPoint(E, Q[1]/Q[2]);
    assert good;
    assert isfull_order(splits(Qt, Afactors));
    assert Qt * A eq infty;
    good, PQt := IsPoint(E, PQ[1]/PQ[2]);
    assert good;
    assert isfull_order(splits(PQt, Afactors));
    assert PQt * A eq infty;
end for;

// Last l-isogeny
printf ">> Computing the last %o-isogeny codomain (public key computation)\n",
↪  Bfactors[#Bfactors];
Kps := KernelPoints(K, EB, Bfactors[#Bfactors]);
EB, Kps := xISOG(EB, Kps, Bfactors[#Bfactors]);
P := xEVAL(P, Kps, Bfactors[#Bfactors]);
Q := xEVAL(Q, Kps, Bfactors[#Bfactors]);
PQ := xEVAL(PQ, Kps, Bfactors[#Bfactors]);
E := EllipticCurve(x^3 + coeff(EB)*x^2 + x);
assert Random(E) * (p + 1) eq infty;
printf "\n%o\n", E;
good, Pt := IsPoint(E, P[1]/P[2]);
assert good;
assert isfull_order(splits(Pt, Afactors));
assert Pt * A eq infty;
good, Qt := IsPoint(E, Q[1]/Q[2]);
assert good;
assert isfull_order(splits(Qt, Afactors));
assert Qt * A eq infty;
good, PQt := IsPoint(E, PQ[1]/PQ[2]);
assert good;
assert isfull_order(splits(PQt, Afactors));
assert PQt * A eq infty;
if Pt + Qt eq PQt then Qt := -Qt; end if;
if Pt + Qt eq -PQt then Pt := -Pt; end if;
if Pt - Qt eq -PQt then
    Pt := -Pt;
    Qt := -Qt;
end if;
assert Pt - Qt eq PQt;
P := Pt;
Q := Qt;
assert isfull_order(splits(P, Afactors));
assert A * P eq infty;
assert isfull_order(splits(Q, Afactors));
assert A * Q eq infty;
print "P := E!", Eltseq(P)[1 .. 2], ";";
print "Q := E!", Eltseq(Q)[1 .. 2], ";";

// Computing the endomorphism \gamma : E_0 --> E_0
alpha := 61;
assert A mod alpha eq 0;
beta := 191;
assert B mod beta eq 0;
c := (2 * (A div alpha)) - (B div beta);
printf "\nc = %o\n", c;
good, u, v := OddCyclicSumOfSquares(c, [], false);
assert good;
printf "gamma = [%o] + [%o] * (2i)\n", u, v;

// SetVerbose("AVIsogenies",4);
// import "/Users/chi-dominguez/GitLab/avisogenies/src/Arithmetic.m" : precompute_D;
// import "/Users/chi-dominguez/GitLab/avisogenies/src/Isogenies.m" : precompute_isogenies;
// //warning: the import lines assume that we are in the avisogenies repository. Adapt the
↪  filenames otherwise

guess := 7;
printf "+++++++\t guess: %o\n", guess;
bi := B div beta;
tauhatkernel := bi*PB + guess*bi*QB;
assert tauhatkernel ne infty;
assert tauhatkernel * beta eq infty;
tauhatkernel_distort := u*tauhatkernel + v*two_i(tauhatkernel);
assert tauhatkernel_distort ne infty;
assert tauhatkernel_distort * beta eq infty;
K := [tauhatkernel_distort[1], fq!1];
C := [fq!6+2, fq!4]; // Corresponding with A=6
Kps := KernelPoints(K, C, beta);
C, Kps := xISOG(C, Kps, beta);
P_c := xEVAL([PA[1], fq!1], Kps, beta);
Q_c := xEVAL([QA[1], fq!1], Kps, beta);
PQ_c := xEVAL([(PA-QA)[1], fq!1], Kps, beta);
C := EllipticCurve(x^3 + coeff(C)*x^2 + x);
assert Random(C) * (p + 1) eq infty;
printf "\n%o\n", C;
good, Pt_c := IsPoint(C, P_c[1]/P_c[2]);
assert good;
assert isfull_order(splits(Pt_c, Afactors));
assert Pt_c * A eq infty;
good, Qt_c := IsPoint(C, Q_c[1]/Q_c[2]);
assert good;
assert isfull_order(splits(Qt_c, Afactors));
assert Qt_c * A eq infty;
good, PQt_c := IsPoint(C, PQ_c[1]/PQ_c[2]);
assert good;
assert isfull_order(splits(PQt_c, Afactors));
assert PQt_c * A eq infty;
if Pt_c + Qt_c eq PQt_c then Qt_c := -Qt_c; end if;
if Pt_c + Qt_c eq -PQt_c then Pt_c := -Pt_c; end if;
if Pt_c - Qt_c eq -PQt_c then
    Pt_c := -Pt_c;
    Qt_c := -Qt_c;
end if;
assert Pt_c - Qt_c eq PQt_c;
print "P_c := C!", Eltseq(Pt_c)[1 .. 2], ";";
print "Q_c := C!", Eltseq(Qt_c)[1 .. 2], ";\n";

exit;
```

References

1. Azarderakhsh, R., Campagna, M., Costello, C., De Feo, L., Hess, B., Jalali, A., Jao, D., Koziel, B., LaMacchia, B., Longa, P., Naehrig, M., Pereira, G., Renes, J., Soukharev, V., Urbanik, D.: Supersingular Isogeny Key Encapsulation. Third Round Candidate of the NIST's post-quantum cryptography standardization process (2020), available at: https://sike.org/
2. Castryck, W., Decru, T.: An efficient key recovery attack on SIDH (preliminary version). IACR Cryptol. ePrint Arch. p. 975 (2022), https://eprint.iacr.org/2022/975
3. De Feo, L., Jao, D., Plût, J.: Towards quantum-resistant cryptosystems from supersingular elliptic curve isogenies. Journal of Mathematical Cryptology **8**(3), 209–247 (2014). https://doi.org/10.1515/jmc-2012-0015, https://doi.org/10.1515/jmc-2012-0015
4. Fouotsa, T.B.: SIDH with masked torsion point images (2022), https://eprint.iacr.org/2022/1054
5. Jao, D., De Feo, L.: Towards Quantum-Resistant Cryptosystems from Supersingular Elliptic Curve Isogenies. In: Yang, B. (ed.) Post-Quantum Cryptography - 4th International Workshop, PQCrypto 2011, Taipei, Taiwan, November 29 - December 2, 2011. Proceedings. Lecture Notes in Computer Science, vol. 7071, pp. 19–34. Springer (2011). https://doi.org/10.1007/978-3-642-25405-5_2, https://doi.org/10.1007/978-3-642-25405-5_2

6. Longa, P., Wang, W., Szefer, J.: The Cost to Break SIKE: A Comparative Hardware-Based Analysis with AES and SHA-3. In: Malkin, T., Peikert, C. (eds.) Advances in Cryptology - CRYPTO 2021 - 41st Annual International Cryptology Conference, CRYPTO 2021, Virtual Event, August 16-20, 2021, Proceedings, Part III. Lecture Notes in Computer Science, vol. 12827, pp. 402–431. Springer (2021). https://doi.org/10.1007/978-3-030-84252-9_14, https://doi.org/10.1007/978-3-030-84252-9_14
7. Maino, L., Martindale, C.: An attack on SIDH with arbitrary starting curve. IACR Cryptol. ePrint Arch. p. 1026 (2022), https://eprint.iacr.org/2022/1026
8. Moriya, T.: Masked-degree SIDH (2022), https://eprint.iacr.org/2022/1019
9. Oudompheng, R., Pope, G.: A Note on Reimplementing the Castryck-Decru Attack and Lessons Learned for SageMath (2022), https://eprint.iacr.org/2022/1283
10. Robert, D.: Breaking SIDH in polynomial time. IACR Cryptol. ePrint Arch. p. 1038 (2022), https://eprint.iacr.org/2022/1038