A Note on Reimplementing the Castryck-Decru Attack and Lessons Learned for SageMath

Rémy Oudompheng
remyoudompheng@gmail.com

Giacomo Pope
giacomo.pope@nccgroup.com

September 27, 2022

Abstract. This note describes the implementation of the Castryck-Decru key recovery attack on SIDH using the computer algebra system, SageMath. We describe in detail alternate computation methods for the isogeny steps of the original attack ($(2,2)$-isogenies from a product of elliptic curves and from a Jacobian), using explicit formulas to compute values of these isogenies at given points, motivated by both performance considerations and working around SageMath limitations. A performance analysis is provided, with focus given to the various algorithmic and SageMath specific improvements made during development, which in total accumulated in approximately an eight-fold performance improvement compared with a naïve reimplementation of the proof of concept.

1 Introduction

Isogeny-based cryptography recently experienced a dramatic upheaval after Castryck and Decru presented a heuristic polynomial time key recovery attack against the Supersingular Isogeny Diffie-Hellman protocol (SIDH) [CD22a]. Accompanying the preliminary version of their paper, the authors additionally published an implementation of their attack [CD22b], which claimed to break the SIKE NIST level one parameter set in only an hour and the NIST level five parameters in less than one day running on a single core.

The publication of such a devastating attack[1] against SIDH brought with it a torrent of research interest and public attention. This was only amplified when days later, an independently discovered attack [MM22] described a subexponential algorithm to solve the same problem without assuming knowledge of the endomorphism ring of the starting curve. This paper was quickly followed by a paper [Rob22] describing an improvement which would allow a similar attack to be performed in proven polynomial time by considering isogenies in higher dimensions. An implementation of the Maino-Martindale attack has been written for SageMath and will be made publicly available soon. Robert is currently working on an implementation for the dimension eight attack.

The novel attack put forward by Castryck and Decru used (among other machinery) the computation of $(2,2)$-isogenies of abelian surfaces. Although this topic has a long history of research from mathematicians, it has only recently gained interest in the cryptographic community due to its constructive use in designing higher genus Diffie-Hellman protocols [FT19] and hash functions [CDS19, CD21].

[1]The SIKE parameter sets for the SIDH protocol had for a decade been left unscathed by cryptanalysis. A historical overview is presented in [Cos21].

Because of the sophistication of the mathematical tools used in the cryptanalysis of SIDH, for a large section of the cryptographic community, being able to run the attack on their own hardware was essential to verify the paper's claims. However, the proof of concept code has been written using the computer algebra system Magma [BCP97]. Magma is a very efficient and powerful piece of software, but it is difficult for many people to obtain access to. Despite public availability of code which could run the attack over a lunch break, the general public could not replicate the results themselves using only publicly available software components.

Motivated by the appearance of such a beautiful attack and the desire to promote open-source alternatives, a plan was made to reimplement the attack using SageMath [Sag22]; a free, open-source mathematics software system. This was not only a great opportunity to learn precisely how the attack was constructed, but more importantly, sharing an implementation which could be run by anyone would hopefully inspire other researchers to experiment with the novel cryptanalytic machinery.

The purpose of this note is to describe the process of reimplementing an algorithm from Magma to SageMath, with the hope that as a community we can work together to port additional Magma-only algorithms to SageMath in the future. We discuss problems we encountered along the way, along with our solutions. Hopefully, offering enough detail that our work can be applied to similar problems in the future.

The current version of our implementation deviates from the original description of the attack, and these deviations along with other SageMath specific optimisations have resulted in significantly faster running times, with the difference being more pronounced for the higher security parameter sets. Approximate running times are compared with those published in [CD22a] in Table 1.

Approximate Running Time	`$IKEp217`	`SIKEp434`	`SIKEp503`	`SIKEp610`	`SIKEp751`
Proof of Concept (Magma)	6 mins	1 hour	2h19m	8h15m	20h37m
This Note (SageMath)	2 mins	10 mins	15 mins	25 mins	1-2 hours

Table 1: Comparison between running times of the original proof of concept [CD22b] and the current version of our SageMath implementation [OPP$^+$22]. Magma times were recorded with a Intel Xeon CPU E5-2630v2 @ 2.60 GHz and SageMath times were achieved on an Intel Core i7-9750H CPU, both running on a single core.

Implementation. For those wanting to run the attack themselves, our implementation and supporting documentation is provided at

`https://github.com/jack4818/Castryck-Decru-SageMath`

Outline. The structure of this note is as follows. We first set notation and give a brief overview of the SIDH protocol sufficient to understand the attack. A high-level description of the Castryck-Decru attack is then offered, with focus given to the necessary conditions required for the attack to be successful. This is followed by a description of the translation process from Magma to SageMath, highlighting specifically when naïve usage of SageMath was slow and how this was addressed in our implementation. In particular, slow performance at one point in the algorithm inspired the derivation of compact explicit formulas for computing $(2,2)$-isogenies which are described in detail. The note finishes by giving an overview of performance analysis of our code and a detail of the improvements which led to an eight-fold improvement against our initial direct reimplementation. We include links to various enhancements which have been made for future SageMath versions after various slow algorithms were identified in the process of our work.

Acknowledgments. We would like to thank Lorenz Panny for both his assistance during the implementation and also for his insightful comments and suggestions offered during the writing of this note. Additionally, we thank Thomas Decru, Elena Bakos Lang, Chloe Martindale and Thomas Pornin for their comments on earlier drafts of this note. Finally, we would like to thank the wider CryptoHack community. Much of the development of this project was done in a chatroom with many people contributing ideas. In particular, we thank Robin Jadoul and Unblvr who helped with profiling and writing patches which have been included into the reimplementation.

2 Background

2.1 Supersingular Isogeny Diffie-Hellman

Supersingular Isogeny Diffie-Hellman (SIDH) was proposed in 2011 as a quantum-safe key exchange mechanism [JdF11]. Over the past decade, it has gained a lot of cryptanalyic attention, thanks in most part to its use in SIKE [ACC$^+$20], a key encapsulation mechanism which was submitted to the NIST Post-Quantum Cryptography Project. The inspiration for the protocol comes from the following problem:

Problem 2.1 (ℓ-isogeny path)**.** Given a prime p and two supersingular elliptic curves E_1, E_2 defined over the field $\mathbb{F}_{p^2}$, find a path between E_1 and E_2 in the ℓ-isogeny graph.

This problem is (still) assumed to be cryptographically hard for both classical and quantum algorithms for sufficiently large p, with the best-known-attacks having exponential complexity. Additionally, to allow for efficient protocols, ℓ is usually taken to be small.

Let us first describe the protocol at a high-level. The SIDH key exchange can be understood in direct analogy to the more familiar Elliptic Curve Diffie-Hellman (ECDH) key exchange which uses scalars as secrets and points on an elliptic curve as public keys. In SIDH, the secrets are isogenies, which are rational maps from one elliptic curve to another; in SIDH one considers isogenies between supersingular elliptic curves. Alice and Bob exchange tuples of data which contain the elliptic curve which the secret isogeny maps to, along with two special points on this new curve. We can visualise this mapping between curves as a walk along a secret path on a graph known as a *supersingular isogeny graph* which has isogenies as edges and (isomorphism classes of) supersingular elliptic curves as nodes.

The protocol is designed such that Alice and Bob take the protocol's public parameters and begin a walk through the graph on their respective secret paths. At the end of their walk, they share information about where they have arrived on the graph. Alice and Bob then set off through the graph on a path determined by their own secret and the other's shared data. When the protocol is completed successfully, Alice and Bob finish their walks and land on the same node in the graph. Data is derived from this shared node which can then be passed to a key derivation function for use in symmetric cryptography.

Public parameters. Let $p = 2^a 3^b - 1$ be a prime. Let E_0 be a fixed, starting supersingular elliptic curve defined over the field[2] $\mathbb{F}_{p^2}$ with order $\#E_0 = (p+1)^2$. Finally, let (P_A, Q_A) and (P_B, Q_B) be generators of the torsion groups $E_0[2^a]$ and $E_0[3^b]$ respectively.

Key pair generation. Alice picks a random integer $0 \leq x_A < 2^a$ and derives a degree 2^a isogeny ϕ_A whose kernel is the subgroup $\langle P_A + [x_A]Q_A \rangle$. The codomain of this isogeny is denoted $E_A = \phi_A(E_0)$. Alice additionally computes the action of her isogeny on the generators of $E_0[3^b]$. The key pair is comprised of the private key: x_A and the public key $(E_A, \phi_A(P_B), \phi_A(Q_B))$, which is sent to Bob. Bob performs a similar process.

Secret derivation. Alice receives Bob's public key: $(E_B, \phi_B(P_A), \phi_B(Q_A))$. Using her secret key, Alice computes a new degree 2^a isogeny ϕ'_A with kernel $\phi_A(\ker(\phi_B)) = \langle \phi_B(P_A) + [x_A]\phi_B(P_A) \rangle$. The shared secret is the *j-invariant* of the codomain of this isogeny $E_{AB} = \phi'_A(E_B)$. Bob performs a similar process to derive a secret from $E_{BA} = \phi'_B(E_A)$. As Alice and Bob are on the same node, their curves are isomorphic and so the j-invariant is the same: $j(E_{AB}) = j(E_{BA})$.

$$\begin{tikzcd}
E_0 \arrow[r, "\phi_A"] \arrow[d, "\phi_B"] & E_A \arrow[d, "\phi'_B"] \\
E_B \arrow[r, "\phi'_A"] & E_{AB}
\end{tikzcd}$$

Figure 1: The SIDH protocol

A keen reader will have noticed that Alice and Bob not only share where they are within the graph, but in order for both parties to end on the same node, they additionally share the action of their isogeny on the generators of the torsion groups. This "leakage" of additional information has worried researchers since SIDH was proposed. The concern was that even if

[2]Whenever $a > 1$, we have $p \equiv 3 \pmod 4$, allowing one to define the extension field $\mathbb{F}_{p^2}$ with the irreducible polynomial $x^2 + 1 = 0$.

ℓ-isogeny path problem 2.1 was cryptographically hard, the SIDH protocol would be totally broken if an efficient solution could be found to the following problem:[3]

Problem 2.2 (Supersingular isogeny with torsion). Let $p = f \cdot N_A \cdot N_B - 1$ be prime, and N_A and N_B be smooth, coprime integers. Additionally, let E_0, E_A be supersingular elliptic curves defined over the field $\mathbb{F}_{p^2}$, connected by some secret degree-N_A isogeny $\phi_A : E_0 \to E_A$. Given the curves E_0 and E_A, together with the restriction of ϕ_A on the N_B-torsion of E_0, recover an isogeny ϕ which matches these constraints. [KMP+20, Problem 1]

Indeed, by using the information leaked by the auxiliary points, "torsion-point" attacks were found, which showed that problem 2.2 could be considered easy for certain classes of parameter sets [Pet17, dQKL+20, KP22]. However, it wasn't until the more recent work of Castryck-Decru, Maino-Martindale and Robert that these attacks could be extended to the parameter sets used by SIDH/SIKE [CD22a, MM22, Rob22]. Before jumping into the mathematics of the attack, we offer a high-level overview and look at how secret data is extracted from the image of the auxiliary points included in the data exchanged between Alice and Bob.

2.2 Castryck-Decru attack

For this section, we focus on the Castryck-Decru attack as it is described in [CD22a] and implemented in the Magma code [CD22b] and leave a discussion of the generalisations of the attacks for the conclusions of the paper.

To recover Bob's private key, the attack uses several properties of the SIDH protocol and SIKE parameters. There are three key pieces of information used in the current implementation:

1. The exchanged data contains the image of the auxiliary points.
2. The secret isogeny ϕ_B has a fixed, known degree.
3. The starting curve E_0 has a known endomorphism ring.

High-level description. The Castryck-Decru attack recovers Bob's secret integer x_B ternary digit by digit; it does not directly compute the secret isogeny itself. After x_B has been recovered, Bob's isogeny ϕ_B and the shared secret $j(E_{BA})$ can be recovered by following the SIDH protocol.

The algorithm works by guessing a step along Bob's unknown path and asking an oracle whether the step was correct. If the oracle returns `false`, a different step can be guessed, otherwise it continues down the chosen path and attempts to recover the next secret digit. This strategy of guessing an isogeny step and consulting an oracle was known prior to this attack. For example, this method is used in the active GPST attack [GPST16] where errors which prevent the SIDH protocol completing successfully serve as the oracle.

[3] Problem 2.2 was first referred to as the **Computational Supersingular Isogeny (CSSI)** problem [JdF11, Problem 5.2], where the casting of the problem is much closer to the computational Diffie-Hellman problem. We follow [KMP+20] to emphasise the knowledge of the auxiliary points along with the codomain of the secret isogeny.

Walking down Bob's secret path, there are only three possible steps to choose from (each corresponding to one of the three ternary digits $\{0, 1, 2\}$). This means that for each taken step, at most two calls to the oracle are needed. It is this fact which makes the attack so efficient.

The genius of the Castryck-Decru attack was creating a suitable oracle to validate whether the step taken is on the correct path using only public data. The oracle was named the *glue-and-split* oracle, and before a slightly more formal discussion is given, we give a moderately high-level overview.

The oracle is given the collected public data together with a cleverly constructed auxiliary isogeny $\gamma : E_0 \to X_0$. From the supersingular elliptic curves (X_0, E_B) a new object[4] is constructed from their product: $X_0 \times E_B$. The images of the auxiliary points P_A, Q_A on both $E_B[2^a]$ and $X_0[2^a]$ are needed. The former can be taken from Bob's public key, while the latter is computed directly from the action of γ. These points are lifted and represented as a pair of points on the Jacobian of a hyperelliptic curve H, obtained by an isogeny (the *gluing map*) $X_0 \times E_B \to \mathrm{Jac}(H)$

This hyperelliptic curve and pair of new points are mapped through a chain of genus two isogenies (known as Richelot isogenies). At the end of this chain, if the resulting hyperelliptic curve can be decomposed back into a product of supersingular elliptic curves, then the curve is said to split, and the oracle returns `true` and `false` otherwise. The reason that the oracle is reliable is that only $\mathcal{O}(p^2)$ of the $\mathcal{O}(p^3)$ total hyperelliptic curves which can be reached from this chain decompose into the product of elliptic curves. For cryptographically large p, when a split is detected we can be confident that it is because our path is correct, and not because we got (un)lucky and stumbled upon some other, random product of elliptic curves.

One can then see that the whole attack hinges on the miraculous splitting of an abelian surface after computing a chain of Richelot isogenies, but only when the correct step has been taken. To understand why this is the case, we must understand the special auxiliary isogeny γ, whose construction is guided by a theorem published by Ernst Kani in 1997.

Proposed countermeasures. At the time of writing, since the publication of [CD22a], two proposals have been put on eprint which aim to circumvent the attack by obscuring the necessary data. In [Mor22], the author suggests masking $\mathrm{End}(E_0)$ and the degree of the secret isogeny and in [Fou22] it is suggested to mask $\mathrm{End}(E_0)$ and the auxiliary points. Aiming to mask $\mathrm{End}(E_0)$ alone is hopeless, thanks to the generalisations [MM22, Rob22] which describe subexponential and polynomial time attacks which do not require knowledge of the starting curve's endomorphism ring.

2.3 Kani's theorem

Kani's theorem [Kan97, Theorem 2.6] is a statement about the existence of a unique *anti-isometry* between the torsion groups: $\psi : X[N] \to E[N]$ for elliptic curves X and E when

[4]Precisely, this is a superspecial principally polarised abelian surface over $\mathbb{F}_{p^2}$.

considering a special configuration of elliptic curves and isogenies. The crux of the Castryck-Decru attack relies on the existence of ψ, together with another genus two isogeny which maps between two products of supersingular elliptic curves.

Let us first describe Kani's theorem borrowing from the notation used previously. We assume the existence of two isogenies $\phi_A : E_0 \to E_A$ and $\phi_B : E_0 \to E_B$ of (coprime) degree N_A and N_B respectively. Furthermore, we consider two additional isogenies $\phi'_A = \phi_B(\ker(\phi_A))$ and $\phi'_B = \phi_A(\ker(\phi_B))$. This is precisely the scenario in SIDH when $N_A = 2^a$ and $N_B = 2^b$. In Kani's language, the SIDH square drawn in Figure 1, is an *isogeny diamond configuration*.

Let us denote $N = N_A + N_B$. Kani's theorem tells us that it is possible to build an explicit (N, N)-isogeny in two dimensions:

$$\Phi : E_B \times E_A \to E_0 \times E_{AB}, \qquad \ker(\Phi) = \langle (\phi_B(P_A), \phi_A(P_A)), (\phi_B(Q_A), \phi_A(Q_A)) \rangle ,$$

where $\psi : E_B[N] \to E_A[N]$ is the unique anti-isometry such that $\psi(\phi_B(x)) = \phi_A(x)$. We note here that Φ can be constructed from its kernel given the action of ϕ_A and ϕ_B on the auxiliary points of E_0. For more precise details on the proof of existence of Φ and the uniqueness of ψ we refer to the original paper [Kan97].

Now let us repeat the above discussion, but changing the perspective to that which suits the attack, with the additional condition that $N_A > N_B$.[5] We consider a starting curve E_0 and Bob's curve E_B which is the codomain of a secret degree N_B-isogeny ϕ_B which we wish to recover. Now consider an arbitrary auxiliary isogeny $\gamma : E_0 \to X_0$ of degree $N_A - N_B$. As $\gcd(N_A, N_B) = 1$, we additionally have $\gcd(N_A - N_B, N_B) = 1$.

We can then construct a new isogeny diamond configuration:

$$\begin{tikzcd} E_0 \arrow[r, "\phi_B"] \arrow[d, "\gamma"] & E_B \arrow[d, "\gamma'"] \\ X_0 \arrow[r, "\phi'_B"] & X_B \end{tikzcd}$$

and Kani's theorem tells us that there is a unique anti-isometry $\psi : X_0[N_A] \to E_B[N_A]$ and a genus two (N_A, N_A)-isogeny which maps between two products of supersingular elliptic curves

$$\Phi : X_0 \times E_B \to E_0 \times X_B,$$

with $\ker(\Phi) = \langle (\gamma(P_A), \phi_B(P_A)), (\gamma(Q_A), \phi_B(Q_A)) \rangle$. For our computations, we will use that $N_A = 2^n$, and so we can efficiently compute Φ as a chain of $(2, 2)$-isogenies of length n, using its kernel determined by the auxiliary point information of the SIDH exchange.

The attack works by making a guess about the next digit of x_B and using this in the construction of γ. When the correct step is taken, the data forms an isogeny diamond configuration and we can test for this by checking whether the codomain of Φ splits. Depending on

[5]When this is not the case, we could switch our targets and recover Alice's secret isogeny instead. However, this is not preferred, as it requires computing a chain of $(3, 3)$-isogenies, which have significantly longer formulas. Instead, we perform an initial guess and consider $N_B = 3^{b-\beta_i}$ such that the inequality $N_A > N_B$ holds.

the digit of x_B that is being guessed, we have $N_A = 2^{a-\alpha_1}$ and $N_B = 3^{b-\beta_1}$ for integers α_1, β_1, meaning the $(2,2)$-isogeny chain will be of length $n = a - \alpha_1$.

2.4 Auxiliary isogenies

The last big hurdle in implementing the attack is to find an efficient way to compute the auxiliary isogeny of degree $x = N_A - N_B$. Without any additional information, we can only compute large degree isogenies using two methods:

1. When $x = y^2$ is a perfect square, we can compute an isogeny of degree x by using the scalar multiplication map $[y]$.
2. When x is smooth, we can factor x and compute γ by composing a chain of isogenies with degree equal to the factors of x.

However, it is unlikely that x will be smooth, and even less likely that x will be a perfect square!

The insight in the Castryck-Decru attack is that the SIKE parameters pick a special starting curve $E_0 : y^2 = x^3 + 6x^2 + x$, for which the endomorphism ring $\mathrm{End}(E_0)$ is known. In particular, we have the endomorphism $2\mathbf{i}$ which satisfies $(2\mathbf{i})^2 = [-4]$.

We can then construct an isogeny of degree x whenever we can represent the degree in the form $x = u^2 + 4v^2$, which is possible if all prime factors of the form $q = 4k + 3$ have even exponent in x. When this condition is satisfied, we can construct the x-isogeny $\gamma = [u] + [v] \cdot 2\mathbf{i}$. This step requires factoring integers of size $\mathcal{O}(2^a)$, but can be performed as a precomputation for the parameter sets considered.

Since the publication of the attack, there has already been work generalising the computation of the auxiliary isogeny. In a note [Wes22], it is shown that when $\mathrm{End}(E_0)$ is known, a degree x isogeny can *always* be computed in polynomial time removing a heuristic property of the attack.[6] In [Rob22] a provable polynomial time algorithm can compute γ even without the knowledge of $\mathrm{End}(E_0)$ by working in higher dimension and using the fact that every integer x can be represented by the sum of four squares.

3 Translating the proof of concept from Magma to SageMath

Cryptanalytic attacks which rely on advanced mathematics are often written using specialised mathematical software known as computer algebra systems (CAS). For cases when the code needs to be optimised, the code is then usually translated to a more performant language after a proof of concept is developed. However, more often than not, these pieces of software are advanced enough to do what is needed. Two of the most used CAS in cryptanalysis are:

[6]The second heuristic property from the $1/p$ failure when studying whether the hyperelliptic curve splits after the (N_A, N_A)-isogeny chain can also be removed as the secret isogeny ϕ_B can be directly recovered from Φ as noticed in [MM22, Oud22, Wes22], removing the risk of proceeding with incorrect guesses.

- **Magma** [BCP97]: a computer algebra software package maintained by the University of Sydney. It is known for having expansive coverage, with efficient implementations of computational algorithms from algebra, number theory and algebraic geometry.
- **SageMath** [Sag22]: a free, open source mathematics software system licensed under the GPL. Its mission is to "Create a viable free open source alternative to Magma, Maple, Mathematica and Matlab." SageMath is built on top of Python, but many algorithms come from other open-source packages and are accessed through wrappers and interfaces, allowing high performance computation.

As Magma is proprietary software, many people active in the cryptographic research community do not have the ability to run Magma files. In contrast, SageMath is accessible to anyone with a suitable computer[7] and internet access (for download/updating only). Because of this, it felt important to take the incredible work of Castryck and Decru and translate it from Magma to something more accessible.

3.1 Overview of code structure

Before detailing the translation process, we offer an overview of the proof of concept code itself. The attack is split between four files:

- `uvtable.m`: a file containing a precomputed array of x, u, v values such that $x = u^2 + 4v^2$. Used when generating the auxiliary isogenies of degree x.
- `richelot_aux.m`: a file with helper functions which compute isogeny chains, and the glue-and-split oracle computations. The bulk of this work is done by three functions:
  - `FromProdToJac()`: Given two elliptic curves and two pairs of points, glues the curves and lifts the points to the Jacobian of the hyperelliptic curve.
  - `FromJacToJac()`: Computes one step in the chain of $(2, 2)$-isogenies using the Richelot correspondance between hyperelliptic curves.
  - `Does22ChainSplit()`: Via `FromProdToJac()` and `FromJacToJac()` computes the codomain of a $(2^a, 2^a)$–isogeny. Returns `true` if the codomain splits to a product of elliptic curves and `false` otherwise.
- `SIKE_challenge.m`: loads the public data from the Microsoft `$IKEp217` challenge [Mic21] and runs the attack to recover Bob's secret integer x_B.
- `SIKEp434.m` generates random auxiliary points and public data using the SIKE NIST level 1 parameter set [ACC+20], runs the attack and recovers Bob's secret integer x_B.

Translating `uvtable.m` was a simple syntax modification. Both `SIKEp434.m` and `SIKE_challenge.m` contained some more sophisticated code, but nothing which would

[7] SageMath is fairly large (the current recommendation for installation is a minimum of 6GB of free disk space), with a recommended minimum of 2GB of RAM. Installation is straightforward for Linux and MacOS. For Windows users, the recommendation is either to run SageMath within WSL2 or within the SageMath Docker.

seem out of place in standard isogeny-based cryptography. Aside from syntax translation of these files, we would need to compute Weil pairings, chains of elliptic curve isogenies, various bases for torsion groups and the j-invariant for elliptic curves. From previous experience, we knew SageMath had efficient implementations for all these pieces.

The file which would take the most work was `richelot_aux.m`. In particular, the functions `FromProdToJac()` and `FromJacToJac()` make up the bulk of the file and require solving multivariate polynomials, computing the image of elliptic curve points on the Jacobian of the glued hyperelliptic curve and the explicit computation of $(2, 2)$-isogenies using the Richelot correspondence.

3.2 Syntax translation

The busy work of translating from Magma to SageMath was simply understanding and translating the syntax. Luckily, there is so much overlap between the two languages, files can be copied verbatim and resulting syntax errors corrected.

Some changes, like variable declaration from `a := 110;` to `a = 110` were easy to fix up. Additionally, many of the higher-level mathematical objects such as `EllipticCurve()` and `PolynomialRing()` had almost identical representations. In Figure 2, the first few lines of `SIKE_challenge.m` are given, along with the corresponding SageMath translation, which hopefully shows how similar the code is between the two systems.

```
// SIKE_challenge.m
a := 110;
b := 67;
p := 2^a*3^b - 1;
Fp2<I> := GF(p, 2);
assert I^2 eq -1;
R<x> := PolynomialRing(Fp2);
E := EllipticCurve(x^3 + 6*x^2 + x);
```

```
# SageMath Translation
a = 110
b = 67
p = 2^a*3^b - 1
Fp2.<i> = GF(p^2, modulus=x^2+1)
assert i^2 == -1
R.<x> = PolynomialRing(Fp2)
E = EllipticCurve(Fp2, [0,6,0,1,0])
```

Figure 2: Comparison of Magma and SageMath code showing the close similarity between how algebraic objects are constructed.

Occasionally, Magma code could look quite different, but usually from context it was easy to see what the intention was. Two differences we encountered which may not be totally obvious are:

- In Magma, the length of an array is denoted `#my_array`. In SageMath we have the pythonic `len(my_array)`.
- For two elements of a field `Px,Py` and an elliptic curve `E`, we represent the point $P = (P_x, P_y)$ by `P := E ! [Px, Py];` in Magma, and `P = E(Px,Py)` in SageMath. The same notation is used for setting elements of a polynomial ring.

There were also cases when the Magma code would use some algebraic object which wasn't immediately available in SageMath. One example of this was that Magma can work with multivariate function fields by calling `FF<u, v, x, y> := FunctionField(Fp2, 4);`. SageMath currently only supports univariate function fields. However, the solution to this problem is to first define a multivariate polynomial ring and create the fraction field from this:

```
# SageMath
FF_poly.<u, v, x, y> = PolynomialRing(Fp2, 4)
FF = FF_poly.fraction_field()
```

Mathematics aside, the difference which caused the most bugs during conversion was very simple. Magma accesses elements in arrays using 1-index, and when looping through a range, it is inclusive of the upper bound. In contrast, SageMath is 0-indexed and does not include the upper bound. In this sense, Magma behaves in the "old-style" similar to Fortran or Pascal, whereas (via Python) SageMath follows the 0-index convention started with C (or rather its predecessor B). An example of this difference is given in Figure 3.

```
// Magma
my_array := [2,3,5,7,11];

for i in [1..5] do
  print my_array[i];
end for;
// output: 2,3,5,7,11
```

```
# SageMath
my_array = [2,3,5,7,11]

for i in range(0,5):
    print(my_array[i])
# output: 2,3,5,7,11
```

Figure 3: Example of the difference in indexing and upper-bounds in a range.

As a result of this difference, careless copy-pasting and tidying could easily introduce off-by-one errors throughout the code. This is exactly what happened and correcting these syntax typos was being performed all the way up to the code working!

3.3 Warning: falling back to very slow toy implementation

The first major difficulty came while reimplementing `FromProdToJac()`. At a high-level, this function takes pairs of points (P_X, Q_X) and (P, Q) on two elliptic curves X and E and lifts them to (P_X, P) and (Q_X, Q), which are points on the Jacobian of a hyperelliptic curve obtained by applying the gluing map: $X \times E \to \mathrm{Jac}(H)$.

To perform the lifting of points, the Castryck-Decru attack derives five multivariate equations in four variables. Although the lines which define these polynomials appear dense, the similarity between Magma and SageMath meant it was easy to translate the first half of the function. A simultaneous solution to these polynomials is computed, which is then used to build a point on the Jacobian of the hyperelliptic curve. Details of this process are described in [CD22a, Section 6.1].

The standard method to solve systems of equations like this is to first build an ideal from the set of polynomials and then compute the ideal's Gröbner basis. Magma comes with

GrobnerBasis() and it is very efficient and works with a wide range of polynomial rings.

In the proof of concept, the function GrobnerBasis() is not directly called. Instead, a scheme is created from an affine space and the set of equations. Calling the Magma function Points(V) on the scheme finds the set of points, which are equivalently the set of solutions to the polynomials. Points(V) computes this by (in part) calling GrobnerBasis(). A snippet showing this code, together with a translation into SageMath is given in Figure 4.

```
// Magma snippet
A4<U0, U1, V0, V1> := AffineSpace(Fp2, 4);
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;
```

```
# SageMath translation
A4.<U0, U1, V0, V1> = AffineSpace(Fp2, 4)
V = A4.subscheme([eq1, eq2, eq3, eq4, eq5])

realsols = []
for D in V.rational_points():
    Dseq = list(D)
    if not 0 in Dseq:
        realsols.append(Dseq)
```

Figure 4: A snippet of Magma code and its SageMath direct translation which is used to derive solutions to the set of five multivariate polynomials over the field $\mathbb{F}_{p^2}$.

Despite the SageMath translation having an almost identical form, running the code we are given the following warning:

```
verbose 0 (3848: multi_polynomial_ideal.py, groebner_basis) Warning:
    falling back to very slow toy implementation.
```

The problem is that SageMath does not natively have a fast implementation to compute the Gröbner basis of ideals in a multivariate polynomial ring. When the multivariate ring is defined over finite fields, the Gröbner basis is usually computed using the external library Singular [DGPS22]. Singular has implemented the Faugère's F4 and F5 algorithms, which are very fast, and the same as what Magma is said to use. Singular has been designed to work well with prime order finite fields of large characteristic. However, when working with fields of prime-power order, Singular can only handle fields with characteristic less than

2^{29}. Msolve, another library available in SageMath for working multivariate polynomials [BED21] also has these fast algorithms, but is restricted to only work with finite fields of characteristic less than 2^{31}.

When the polynomial ring or base field are incompatible with external libraries, SageMath falls back to a toy implementation of Buchberger's algorithm. This was the first algorithm proposed for computing Gröbner bases, but it is known to often require a large amount of memory or generate polynomials much larger than the input, making it impossible to terminate in a reasonable amount of time. The question then is how slow is *very slow* for our configuration. When running the attack, `FromProdToJac()` would be called for each oracle request and each call needed to lift two sets of points. This meant it would be called a few hundred times for the easiest `$IKEp217` challenge and a magnitude more for the hardest parameter set.

After the code whirred for long enough, we stopped the execution and decided to run the same attack but on a reduced, 64-bit prime $p = 2^{33} \cdot 3^{19} - 1$ which we'll refer to as the `SIKEp64` parameter set. Even with this reduced prime, the code could not find a solution in the 30 minutes we let the code run for. It was obvious that this method was the wrong avenue for our SageMath implementation.

3.4 Searching for efficient alternatives

SageMath is used by a lot of people in the cryptography community, and often people find clever tricks to allow SageMath to be more performant. The hope was that if we reached out to some friends with our current problem, someone may have solved something similar before. Finding solutions to multivariate polynomials isn't a particularly obscure problem! Below are some suggestions which seemed hopeful but didn't lead immediately to a solution:

- Lorenz Panny suggested to Weil restrict the equations by introducing $2 \cdot 5 + 1$ variables allowing us to work in $\mathbb{F}_p$ rather than $\mathbb{F}_{p^2}$.
    - This trick would mean that the toy implementation of Gröbner would not be needed anymore as the polynomial ring would be defined over a prime order field, but now the system of equations was so complicated, the code was still too slow.
- Tony Shaska suggested that instead of computing the Gröbner bases, a faster method could be to instead use resultants to remove variables from the equation one at a time and then solve the equation in a univariate polynomial ring and work backwards.
    - This is a clever idea, but SageMath was still too inefficient. The only method available to compute resultants for the polynomial ring over an extension field was to compute the determinant of the Sylvester matrix, which was very slow for our problem.
- Bryan Gillespie suggested to try and use the `Macaulay2` [GS] interface to compute

the Gröbner basis. This doesn't come with SageMath by default, but is free and open-source and can be included pretty easily. It's also known for being pretty fast.

- This solution may work, but the current SageMath interface doesn't allow for sending extension fields to `Macaulay2` (it's not even certain from the documentation that `Macaulay2` can do this, but the SageMath interface certainly can't). For this to have a chance at working, one would first have to rewrite the interface.[8]

The solution came from one of the authors of this note, who saw a way to avoid the problem of the slow Gröbner basis altogether with the following suggestion:

> Are you trying to lift a pair (P_X, P) to the Jacobian? I wonder if it's easier to lift $(P_X, 0)$ to a divisor on H, lift $(0, P)$ to a divisor and add them? I may be confused but it feels like it gives the answer without solving any equations.

This indeed worked, and the result is an efficient and clean piece of code which feels natural to the context of the gluing and lifting. Avoiding solving complicated polynomials simplified `FromProdToJac()` and the work carried over in a similar way in the computations in `FromJacToJac()`. Due to significant differences in the computations of these algorithms (with respect to the Castryck-Decru attack), we describe them in detail in Section 4.

3.5 Baby steps towards a working implementation

With the novel work which allowed efficient lifting and higher-genus isogenies, the Gröbner obstacle was removed and we finished off the rest of the translation. We ran the attack against `SIKEp64` and got the following output:

```
Bridging last gap took: 0.1307520866394043
Bob's secret key revealed as: 15002860
In ternary, this is: [1, 1, 1, 1, 0, 0, 0, 2, 0, 0, 2, 0, 1, 0, 0, 1]
Altogether this took 43.73990249633789 seconds.
```

It worked! The attack successfully recovered Bob's private key in less than a minute. It was so exciting to see Wouter Castryck and Thomas Decru's attack run in real time on our own laptops.

However, a 64-bit prime wasn't close to being secure from previously known attacks. So, with confidence that the code was correct, the next test was to see whether the implementation was efficient enough to recover private keys on serious SIDH instances. However, before detailing performance analysis and the improvements we made to speed up the attack, we describe the compact, explicit formulas derived for our implementation.

[8] If `Macaulay2` can indeed work with extension fields, this could be a nice project for a reader looking to improve SageMath.

4 Compact explicit formulas for dimension two isogenies

As described above, two types of isogenies need to be computed to perform the attack. In this section, we first give a detailed discussion of the gluing of elliptic curves to recover the Jacobian of a genus two hyperelliptic curve. This is followed by a description of efficient formulas to compute Richelot isogenies between genus two Jacobians by quotienting out an explicit $(2,2)$-torsion subgroup.

4.1 Gluing elliptic curves

The first step in the computation is a quotient of $E_1 \times E_2$ by a subgroup $G = \langle (P_1, P_2), (Q_1, Q_2) \rangle$ generated by points of order two (corresponding to roots of the Weierstrass equations of E_1 and E_2).

Assuming reduced Weierstrass equations for E_1 and E_2:

$$\begin{aligned} y^2 &= (x-\alpha_1)(x-\alpha_2)(x-\alpha_3), \\ y^2 &= (x-\beta_1)(x-\beta_2)(x-\beta_3), \end{aligned}$$

the chosen subgroup is equivalent to the choice of a bijection between sets $\{\alpha_i\}$ and $\{\beta_i\}$, and we assume the numbering has been chosen to match this bijection.

The gluing construction is the definition and computation of an isogeny:

$$\gamma : E_1 \times E_2 \to S \simeq \mathrm{Jac}(H),$$

where $\ker \gamma = G$ and H is a genus two (hyperelliptic) curve defined explicitly. This construction uses formulas from [HLP00] for hyperelliptic (genus two) double covers of elliptic curves but abundant literature can be found in recent research.

We will take some time to explain how they can be derived from hopefully simple considerations. Then, the modified way of computing the isogeny itself for given points of $E_1 \times E_2$ will be described.

4.1.1 *Simple case*

The construction is easiest to understand in the following *symmetric* case where $\alpha_i \beta_i = \kappa$. Define curves:

$$\begin{aligned} E_1 &: y^2 = (x-\alpha_1)(x-\alpha_2)(x-\alpha_3), \\ E_2 &: y^2 = (x-\beta_1)(x-\beta_2)(x-\beta_3), \\ H &: y^2 = (x^2-\alpha_1)(x^2-\alpha_2)(x^2-\alpha_3). \end{aligned}$$

Then there is a degree two map:

$$H \to E_1 : (x, y) \to (x^2, y),$$

and the following identity holds on H:

$$\left(\frac{y}{x^3}\right)^2 = -\kappa^{-3}\alpha_1\alpha_2\alpha_3\left(\frac{\kappa}{x^2}-\beta_1\right)\left(\frac{\kappa}{x^2}-\beta_2\right)\left(\frac{\kappa}{x^2}-\beta_3\right),$$

meaning that we can define the second projection:

$$H \to E_2 : (x,y) \to (\kappa/x^2, y/(Ax^3)),$$

where A is a square root of $-\alpha_1\alpha_2\alpha_3/\kappa^3$ (assumed to exist).

4.1.2 General case

In the general case, we want to determine two affine transformations:

$$\begin{aligned} E_1 \to E_1' &: x \mapsto u_1 x + v_1, \\ E_2 \to E_2' &: x \mapsto u_2 x + v_2, \end{aligned}$$

such that the roots α_i and β_i map to α_i' and β_i' such that $\alpha_i'\beta_i' = 1$, where the previous construction applies.

This can be obtained through a linear equation with matrix:

$$M = \begin{pmatrix} \alpha_1\beta_1 & \alpha_1 & \beta_1 \\ \alpha_2\beta_2 & \alpha_2 & \beta_2 \\ \alpha_3\beta_3 & \alpha_3 & \beta_3 \end{pmatrix}, \qquad M\begin{pmatrix} u_1u_2 \\ u_1v_2 \\ u_2v_1 \end{pmatrix} = \begin{pmatrix} 1-v_1v_2 \\ 1-v_1v_2 \\ 1-v_1v_2 \end{pmatrix}.$$

A slightly different solution can be obtained by defining:

$$\begin{pmatrix} R \\ S \\ T \end{pmatrix} = \begin{pmatrix} \alpha_1\beta_1 & \alpha_1 & \beta_1 \\ \alpha_2\beta_2 & \alpha_2 & \beta_2 \\ \alpha_3\beta_3 & \alpha_3 & \beta_3 \end{pmatrix}^{-1} \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}, \quad \text{and } D = \det M.$$

Then for $i = 1,2,3$, $(R\alpha_i + T)(R\beta_i + S) = R + ST$. The interchange of α_i and β_i has effect $S \leftrightarrow T$.

A computer algebra system can be used to factor and prove that:

$$R + ST = -\frac{(\alpha_1-\alpha_2)(\alpha_2-\alpha_3)(\alpha_3-\alpha_1)(\beta_1-\beta_2)(\beta_2-\beta_3)(\beta_3-\beta_1)}{D^2}.$$

If $\delta_\alpha = (\alpha_1-\alpha_2)(\alpha_2-\alpha_3)(\alpha_3-\alpha_1)$ (resp. δ_β) are square roots of the discriminants, then we have

$$R + ST = -\frac{\delta_\alpha\delta_\beta}{D^2}.$$

This means that through simple translations $\alpha_i' = \alpha_i + T/R$ and $\beta_i' = \beta_i + S/R$ we obtain

$\alpha'_i\beta'_i = \kappa$. The factorisation of $R + ST$ gives a more precise identity:

$$-\frac{RD}{\delta_a}(\alpha_i + T/R)\frac{RD}{\delta_\beta}(\beta_i + S/R) = 1.$$

In the following part, we assume that the additive transform has been performed and that $S = T = 0$. Note that the determinant D is not invariant under additive transforms. However, the quantities RD, δ_α and δ_β are invariant under additive transforms. This is because:

$$RD = \det \begin{vmatrix} 1 & \alpha_1 & \beta_1 \\ 1 & \alpha_2 & \beta_2 \\ 1 & \alpha_3 & \beta_3 \end{vmatrix}.$$

We now assume S = T = 0. Define $s_1 = -\delta_\alpha/(RD)$ (we will see that it matches the definition of [HLP00]), let $\alpha'_i = -\alpha_i/s_1$ and define curves:

$$\begin{aligned} E_1 &: y^2 = (x-\alpha_1)(x-\alpha_2)(x-\alpha_3), \\ E_2 &: y^2 = (x-\beta_1)(x-\beta_2)(x-\beta_3), \\ H &: y^2 = d(x^2-\alpha'_1)(x^2-\alpha'_2)(x^2-\alpha'_3), \end{aligned}$$

where d is a suitable twist to be determined. This is because modifying roots by adding a constant keeps elliptic curves isomorphic, but multiplying them by a non-square constant transforms elliptic (or hyperelliptic) curves by a quadratic twist.

Then there is a degree two map:

$$H \to E_1 : (x, y) \to (s_1x^2, y\sqrt{s_1^3/d}),$$

because $s_1x^2 - \alpha_1 = s_1(x^2 - \alpha'_1)$. We can choose $d = s_1$ and the map becomes:

$$(x, y) \mapsto (s_1x^2, s_1y),$$

Similarly there is a map, defining $t_1 = \delta_\beta/(RD)$ so that $\beta_i = t_1/\alpha'_i$:

$$H \to E_2 : (x, y) \to \left(\frac{t_1}{x^2}, \frac{y}{x^3}\sqrt{\frac{-\beta_1\beta_2\beta_3}{s_1}}\right),$$

because $t_1/x^2 - \beta_i = -\beta_i/x^2(x^2 - \alpha'_i)$.

We now need to determine an explicit square root of $-\beta_1\beta_2\beta_3/s_1$ to make formulas usable. Again using the help of a computer algebra system we observe that:

$$\frac{-\beta_1\beta_2\beta_3}{s_1} = \left(\frac{\delta_b}{RD}\right)^2 = t_1^2.$$

This can be proved easily in the specific case where we assumed $S = T = 0$, $\alpha_i\beta_i = s_1t_1$ so the matrix M is almost a Vandermonde matrix with rows $(s_1 t_1, s_1 t_1/\beta_i, \beta_i)$. In particular:

$$D = s_1^2 t_1^2 \delta_\beta / (\beta_1 \beta_2 \beta_3),$$

yielding the following formula:

$$\frac{s_1 t_1^2}{\beta_1 \beta_2 \beta_3} = \frac{D}{s_1 \delta_\beta} = \frac{D}{\delta_\beta(-\delta_\alpha/(RD))} = -\frac{RD^2}{\delta_\alpha \delta_\beta} = -1.$$

Now dropping the assumption that $S = T = 0$ we can recover the full formulas:

$$\begin{aligned}
\alpha'_i &= \frac{\alpha_i + T/R}{s_1}, \\
H : y^2 &= s_1(x^2 - \alpha'_1)(x^2 - \alpha'_2)(x^2 - \alpha'_3), \\
H \to E_1 : (x, y) &\mapsto (s_1 x^2 - T/R, s_1 y), \\
H \to E_2 : (x, y) &\mapsto (t_1/x^2 - S/R, t_1 y/x^3).
\end{aligned}$$

The formulas here are slightly different from [HLP00] because the choice of d creates apparent symmetry breaking between α and β. However, the values of s_1 and t_1 in this section were chosen to match [HLP00], and the full formulas are still very closely related, only differing by a factor on the y coordinate, as shown below.

4.1.3 Comparison with Howe-Leprévost-Poonen formulas

If s_1 and t_1 are the factors defined in [HLP00] the following identities can be verified through using a computer algebra system (notably by computing multivariate factorisation):

$$R + ST = R^2 s_1 t_1.$$

And the other constants can be described compactly as:

$$\begin{aligned}
A_{HLP} &= -\delta_\beta RD, & B_{HLP} &= \delta_\alpha RD, \\
s_1 &= -\frac{\delta_\alpha}{RD}, & t_1 &= \frac{\delta_\beta}{RD}, \\
s_2 &= -T/R, & t_2 &= -S/R.
\end{aligned}$$

4.1.4 Lifting divisors

The above construction defines a morphism $H \to E_1 \times E_2$ which defines an isogeny γ : $\mathrm{Jac}(E_1) \oplus \mathrm{Jac}(E_2) \to \mathrm{Jac}(H)$ whose kernel is exactly the group G chosen at the beginning.

Castryck and Decru compute this isogeny by looking for a divisor $\eta \in \mathrm{Jac}(H)$ such that $\hat{\gamma}(\eta) = (P_1, P_2)$ for chosen points on $E_1 \times E_2$ where $\hat{\gamma}$ is the dual isogeny. Then $\gamma(P_1, P_2) = \gamma \circ \hat{\gamma}(\eta) = 2\eta$.

The corresponding system of equations is non-trivial and the isogeny can actually be computed directly. Since γ is a morphism of groups, it is determined by $\gamma_1 : \mathrm{Jac}(E_1) \to \mathrm{Jac}(H)$ and $\gamma_2 : \mathrm{Jac}(E_2) \to \mathrm{Jac}(H)$ which are easily seen to be the canonical morphism pulling back divisors on E_i to H through the projections defined above.

Using the formalism of Mumford coordinates, a point of $\mathrm{Jac}(H)$ is expected to be represented as a pair of points of H through two polynomials $(a(x), y = b(x))$ where a has degree two and b has degree one. Using the definition of the morphisms $H \to E_1$ and $H \to E_2$:

$$\begin{aligned} x_{E_1} &= s_1 x_H^2 + s_2, & x_{E_2} &= t_1/x_H^2 + t_2, \\ y_{E_1} &= s_1 y_H, & y_{E_2} &= t_1 y_H / x_H^3, \end{aligned}$$

we can see which equations (on H) are determined by points (x_1, y_1) on E_1 and (x_2, y_2) on E_2:

$$(s_1 X^2 + s_2 - x_1 = 0, s_1 Y - y_1 = 0),$$
$$(t_1 - X^2(x_2 - t_2) = 0, t_1 Y - y_2 X^3 = 0).$$

These equations are almost exactly reduced Mumford coordinates for the images of P_1 and P_2 in the Jacobian variety of H, the only required step being computing the remainder of $-y_2 X^3/t_1$ modulo $t_1 - X^2(x_2 - t_2)$ to obtain a degree one polynomial.

Then the image of point (P_1, P_2) can be computed using the classical group law on the Jacobian of H. In addition to avoiding computation of Gröbner bases, this method does not involve extracting roots of polynomials, so it involves only a constant amount of basic field operations.

4.2 Richelot isogenies

The following steps are Richelot isogenies where each kernel is a subgroup generated by a pair of order two elements of the Jacobian. Formulas for these isogenies can be found (notably) in [Smi05] and are used in [CD22a]. We note here that shortly after the announcement of the Castryck-Decru attack, Kunzweiler published an efficient implementation of $(2^n, 2^n)$-isogenies [Kun22]. Although originally designed to aid the performance of higher-dimensional isogeny-based protocols, these algorithms also allow better performance for the key recovery attack on SIDH. Towards the end this note, we comment on the inclusion of Kunzweiler's algorithm into our implementation.

The isogenies which we consider in this section have type:

$$\Phi : S \simeq \mathrm{Jac}(H) \to S' \simeq \mathrm{Jac}(H'),$$

and the kernel of Φ can be represented as a group $\ker \Phi = \{0, P, Q, P + Q\}$.

The kernel must also be isotropic for the Weil pairing. Without entering into details, we will only use a consequence which is that the domain curve H can be represented by an equation:

$$H : y^2 = G_1(x) G_2(x) G_3(x),$$

where G_i are factors of the defining polynomials of the hyperelliptic curves, but also Mumford coordinates for P, Q and $P + Q$, defined over the base field.

Then Richelot formulas define new polynomials:

$$H_i = \frac{1}{\det G}(G_j' G_k - G_k G_j'),$$

where G is the matrix of coefficients:

$$\begin{pmatrix} G_{1,0} & G_{1,1} & G_{1,2} \\ G_{2,0} & G_{2,1} & G_{2,2} \\ G_{3,0} & G_{3,1} & G_{3,2} \end{pmatrix}.$$

Since we know that in a 3×3 matrix M, the coefficients of M^{-1} are obtained by 2×2 minors $\pm \det M_{(i,j),(k,l)} / \det M$ we can actually derive a compact representation of the coefficients of H_i which avoids redundant computations:

$$\begin{pmatrix} H_{1,2} & H_{2,2} & H_{3,2} \\ H_{1,1} & H_{2,1} & H_{3,1} \\ H_{1,0} & H_{2,0} & H_{3,0} \end{pmatrix} = \mathrm{diag}(-1, 2, -1) \times \begin{pmatrix} G_{1,0} & G_{1,1} & G_{1,2} \\ G_{2,0} & G_{2,1} & G_{2,2} \\ G_{3,0} & G_{3,1} & G_{3,2} \end{pmatrix}^{-1}.$$

The Richelot isogeny is defined by the following correspondence:

$$G_1(x)H_1(x') + G_2(x)H_2(x') = 0,$$

$$yy' = G_1(x)H_1(x')(x - x')$$

with target curve:

$$H' : y'^2 = H_1(x')H_2(x')H_3(x').$$

To compute the image of a point defined by Mumford coordinates $(a(x), y = b(x))$ (representing a pair of points of H) we are expected to compute the image of these equations, through the correspondence, as equations on H' (defining a 4-tuple of points), and apply standard Cantor reduction to obtain reduced Mumford coordinates in $\mathrm{Jac}(H')$.

This can be done by elimination of variables, but direct computation is also possible as explained in the following sections.

4.2.1 *Computation through a quadratic extension*

Similarly to the case of gluing elliptic curves, we can easily compute Mumford coordinates for the image of a *point* of H through the Richelot correspondence because a point (x, y) defines a degree two equation for x' and a degree three equation $y' = g(x)$ giving non-reduced Mumford coordinates.

However, a k-point of the Jacobian $\mathrm{Jac}(H)$ is not necessarily represented by a pair of k-points of H. Let D be a k-point of $\mathrm{Jac}(H)$ defined by Mumford coordinates $(a(x), b(x))$ whose coefficients are in k. Using a quadratic extension K/k we can compute the x coordinates of a representative divisor for a point of $\mathrm{Jac}(H)$ (roots of a), map them to $\mathrm{Jac}(H')$

and compute their sum through the addition group law. This is, however, computationally suboptimal due to computation of square roots and polynomials in $\mathbb{F}_{p^4}$.

4.2.2 *Symbolic computation and explicit elimination*

We can do *the same* computation using symbolic variables x_1 and x_2 for the roots of polynomial a. Then the coefficients of a are related to symmetric functions $S = x_1 + x_2$ and $P = x_1x_2$.

By mapping the symbolic roots of a through the Richelot correspondence, we can obtain a (non-reduced) representative of $\phi(D)$ whose equations are symmetric functions of x_1 and x_2, thus can be expressed in terms of S and P.

For example, the first Mumford coordinate of $D' = \phi(D) = \phi(D_1) + \phi(D_2)$ (where D_1 and D_2 are the symbolic elements of D) is defined by the union of two equations:

$$G_1(x_1)H_1(x') + G_2(x_1)H_2(x') = 0,$$

$$G_1(x_2)H_1(x') + G_2(x_2)H_2(x') = 0.$$

Note that since x_1 and x_2 are roots of $a(x)$ which is of degree two, we can replace everywhere G_1 and G_2 by their degree one remainder modulo a which greatly simplifies the computation.

This means that the first Mumford coordinate of the divisor D' can be obtained as the product of these equations:

$$G_1(x_1)G_1(x_2)H_1^2 + (G_1(x_1)G_2(x_2) + G_2(x_2)G_2(x_1))H_1H_2 + G_2(x_1)G_2(x_2)H_2^2.$$

It is possible to precompute H_1^2, H_2^2 and H_1H_2 so that the computation reduces to a linear combination of these three polynomials, with coefficients computed from the first Mumford coordinate $a(x)$ of D.

The second Mumford coordinate can be computed by explicit elimination. Again, D' is defined by a union of two equations:

$$\begin{aligned} y_1y' &= G_1(x_1)H_1(x')(x_1 - x'), \\ y_2y' &= G_1(x_2)H_1(x')(x_2 - x'), \end{aligned}$$

and the product reduces to a polynomial of degree one in y' through equation $y'^2 = h'(x')$. Moreover $y_i = b(x_i)$ and $(x_1 - x')(x_2 - x') = a(x)$.

The product equation is:

$$\begin{aligned} 0 = b(x_1)b(x_2)h'(x') \\ - y'H_1'(x')(b(x_1)G_1(x_2)(x_2 - x') + b(x_2)G_1(x_1)(x_1 - x')) \\ + G_1(x_1)G_1(x_2)H_1(x')^2a(x'). \end{aligned}$$

Again, we can use symmetric functions to eliminate x_1 and x_2 and replace them by coefficients of $a(x)$, resulting in an explicit formula. The implementation can either compute this as a multivariate polynomial in (x', y', x_1, x_2) and then select the coefficients of $x_1^i x_2^j$ to be replaced by an expression $x_1^i x_2^j + x_1^j x_2^i = f_{ij}(S, P)$, or we can expand the computation in advance to leave only the explicit formula. Complete formulas can be found in Appendix A as an actual implementation. Cantor's reduction formulas can then be applied to obtain normalised Mumford coordinates.

5 Performance analysis

In this section, we discuss the various performance improvements which were made during the reimplementation of the Castryck-Decru attack. As well as algorithmic and mathematical improvements, we additionally mention parts of the code which were slow specifically due to SageMath and how we were able to optimise around these limitations. We hope that these points in particular will help with future SageMath implementations.

By far the largest performance issue was SageMath's lack of a fast Gröbner basis algorithm for large characteristic extension fields, previously discussed in Section 3. As we were unable to have the algorithm terminate (even for our fabricated 64-bit parameter set) we can consider this as an "infinite bottleneck".

Although we were able to optimise our code by simply avoiding calls to the problematic functions, having a fast implementation for solving multivariate polynomials over extension fields would still be a great addition to SageMath. We encourage anyone interested in implementing this to have a go and to reach out if you wish to collaborate.

5.1 Breaking the Microsoft's `$IKEp217` challenge

Picking up where we left off, we had a working implementation of the attack with a running time of less than one minute against the baby parameter set `SIKEp64`. The first true test of our implementation was to break the Microsoft `$IKE` challenge [Mic21].

The proof of concept implementation stated that their attack took six minutes to recover Bob's private key. Running our implementation against the `$IKEp217` parameter set, a single call to the oracle was taking approximately 30 seconds, which suggested a total running time in the order of hours.

This was significantly slower than we expected it should be. To try and track down the issue, we ran the `SIKEp64` attack again with standard Python profiling tools. As expected, the majority of work done by the algorithm was being performed by `FromJacToJac()`. However, the profiling tool showed an incredible number of constructions of `FiniteField`, which for our code would be elements of $\mathbb{F}_{p^2}$.

Studying the Jacobian arithmetic, a dramatic performance issue was discovered with how SageMath was working with points in the Jacobian of a hyperelliptic curve. When performing arithmetic operations with points, the code would first invoke `GF(p^k)(...)` for every coefficient of each of the point's coordinates.

Aside from the performance issue of constructing thousands of objects, the constructor of the `FiniteField` includes a primality test for every call. As the finite field $\mathbb{F}_{p^2}$ is constructed for every coefficient of every point for every arithmetic operation, we are performing a primality test thousands of times. The larger the characteristic of the field, the more expensive this construction becomes.

One option to fix this issue is to directly patch the SageMath source, ensuring that the objects are cached after construction:

```
from sage.misc.cachefunc import cached_method

@cached_method
def vector_space(self, *args, **kwds):
    # SNIP
```

Although this fixes the problem, one of the motivations for this project was to allow researchers to easily run the attack themselves. Writing an attack which first asked the user to patch the source code wasn't attractive.

A gentler fix was to set the flag `proof.arithmetic(False)` in our code. This globally tells SageMath to use (among many things) a much faster, probabilistic primality test. We're not worried about false positives this could (very rarely) introduce, as we are working with a known, fixed prime. As an example of how dramatic this speed-up is, a primality test of a 1024 bit integer is more than 1000 times as fast with the flag set to `false`:

```
sage: p = random_prime(2^1024)
sage: time assert is_prime(p)
Wall time: 2.86 s
sage: proof.arithmetic(False)
sage: time assert is_prime(p)
Wall time: 2.11 ms
```

Setting this flag to `false` in our attack, we ran the code and found that we were able to recover Bob's private key from the `$IKEp217` challenge in only 30 minutes. Although this fix doesn't address the redundant field constructions, it made the attack fast enough to finish in a reasonable amount of time.

However, the best fix was found by Robin Jadoul in a conversation in the CryptoHack chat [Cry22]. Robin found that we could implement a monkey patch[9] which had the same effect as the hard patch given above:

```
Fp2.<i> = GF(p^2, modulus=x^2+1)
type(Fp2).vector_space =
    sage.misc.cachefunc.cached_method(type(Fp2).vector_space)
```

With the redundant constructions removed thanks to caching, we found that our attack on the `$IKEp217` challenge finished in only 15 minutes (approximately three times slower

[9]A monkey patch is a way to extend or modify the runtime code of dynamic languages such as Python.

than the proof of concept) with an estimated performance speed up of 400-800% against the unpatched code. Following the discussion of this patch, Lorenz Panny has submitted an enhancement[10] to SageMath to defer primality and irreducibility testing before caching, which has been accepted and will be available from SageMath version `9.7`.

Additionally, we ran the attack against the `SIKEp434` parameter set, which finished in only 90 minutes (approximately 1.5 times slower than the proof of concept). A summary of these results are shown in Table 2, labelled as the "first draft". Timings of attacks against the higher parameter sets were delayed, as our profiling had shown other areas of improvements which could be made before running more benchmarks.

5.2 Breaking SIKE over lunch with a laptop

We finish this section with a sparse covering of some additional improvements we made to our implementation. When considered together, our modifications resulted in approximately an eight-fold performance improvement, with the `$IKEp217` challenge being broken in two minutes, the `SIKEp434` parameters broken in ten minutes and the hardest parameter set, `SIKEp751`, taking on average an hour and a half to crack (compared to the Magma proof of concept finishing in 20 hours). A summary of these results are shown in Table 2.

Before continuing, we note that a lot of the performance benefits were gained in improving the implementation of the genus two calculations. The code given in Appendix A represents the code in a clean and clear form, but the code as implemented in the repository has some additional performance tweaks. For a reader who is eager to see the whole process of tweaking and improving the $(2,2)$-isogenies, we suggest reading through the git history of our project.

5.2.1 Improving SageMath performance

Beyond the performance improvements allowed by caching the `vector_space` and avoiding primality checks, other patches and modifications allowed a performance boost by simply avoiding slow python constructions and redundant computations.

Due to Tate's isogeny theorem, we know curves related by isogenies will have the same number of $\mathbb{F}_{p^2}$-rational points. SageMath has the method `E.set_order()` for an elliptic curve `E`. We explicitly set the number of points to $(p+1)^2$ when creating a new curve from the codomain of an isogeny. This saves any time which would need to be spent computing the curve's order (for example when generating a torsion basis). Additionally, we set the optional parameter `num_checks=0` to avoid any trial scalar multiplications which are performed as a check against the order.

When constructing the Jacobian of the hyperelliptic curve, a call is made to `dimension()` which computes the dimension of the curve (in doing this it makes a call to the same slow toy algorithms discussed previously). As the dimension of the curve is always equal to one, this check is redundant and can be removed in a monkey patch:

[10] https://trac.sagemath.org/ticket/34281

```
from sage.schemes.projective.projective_subscheme import
    AlgebraicScheme_subscheme_projective
AlgebraicScheme_subscheme_projective.dimension = lambda self: 1
```

Introducing this patch, we saw approximately a 20% improvement when first included. As with the other patch, this has been submitted as an enhancement[11] to SageMath by Lorenz Panny and will be made available in SageMath version `9.7`. Note that in the final version of our code, we no longer directly call the Jacobian, so this patch was made redundant with the speedier explicit formulas explained in Section 4.

In the computation of Richelot isogenies, many calls to `.monic()` are made, which are generic and slow. By simply not calling this and working with non-monic Mumford coordinates, we saw a 10% improvement.

5.2.2 *Algorithmic improvements*

In the proof of concept code, elliptic curve isogenies are built from the kernel polynomial using Kohel's formulas. By switching to using points in the kernel instead, we can use Vélu's formula instead, for a 10-15% speed-up.

As coded in Magma files, the chain of 3-isogenies has quadratic complexity. Following the description in [JdF11], we used a quasilinear sparse strategy for computing the isogeny chain implemented by Lorenz Panny. This code was already available and awaiting review as an enhancement[12] to be used by default for chains of isogenies where the degree is factored.

Additionally, each degree three isogeny computation can be made faster by passing hints to the function `EllipticCurveIsogeny()` to set the optional parameters: `degree=3` and `check=False`. These modifications of the elliptic curve isogeny computations offer an approximate 20% improvement.

In a similar way, the computation of the chain of Richelot isogenies can be made subquadratic by precomputing certain powers. Implementing this yielded approximately a 10-15% speedup.

5.2.3 *Parallelisation*

Allowing the code to run in parallel (where possible) was not an initial focus, as the goal was to reimplement the attack as Castryck and Decru had described, which was designed to run on a single core. However, there are parts in the algorithm which lend themselves to parallel computations, and it was an interesting addition to the attack. Parallel compute is available in our implementation and is accessed by including the `--parallel` argument.

One bottleneck of the algorithm is in guessing the first β_1 digits all at once, a limitation which comes from finding $x = N_A - N_B$ such that the degree of the auxiliary isogeny can be expressed as the sum of two squares. For the smaller parameter sets we have $\beta_1 = 2$, which

[11] https://trac.sagemath.org/ticket/34284
[12] https://trac.sagemath.org/ticket/34239

Approximate Running Time	`$IKEp217`	`SIKEp434`	`SIKEp503`	`SIKEp610`	`SIKEp751`
Proof of Concept (Magma)	6 mins	1 hour	2h19m	8h15m	20h37m
First Draft (SageMath)	15 mins	1.5 hours	–	–	–
This Note (SageMath)	2 mins	10 mins	15 mins	25 mins	1-2 hours
Direct Calculation [Oud22]	9 secs	22 secs	2 mins	15 mins	1 hour

Table 2: Comparison between running times of the original proof of concept [CD22b] and our SageMath implementation [OPP+22]. Also included are the approximate running times for the direct method [Oud22] which recovers the secret isogeny after only the first digits are recovered. Magma times were recorded with a Intel Xeon CPU E5-2630v2 @ 2.60 GHz and SageMath times were achieved on an Intel Core i7-9750H CPU, both running on a single core.

only requires $3^2 = 9$ guesses, but for the hardest parameter set we have $\beta_1 = 6$, which means in the worst-case scenario, more than half of the computation time is recovering six of the 249 unknown ternary digits.

Each guess for the first digits can be made in parallel, and so this part of the code can be made approximately n-times faster when computed using n-cores. For `SIKEp751`, where $3^6 = 729$ guesses are made in the worst case, parallel compute makes a significant difference.

For the remaining digits, the best we can hope for is a speed-up by computing both oracle calls simultaneously. This means we halve the computation time for approximately two-thirds of the digits, but doubles the amount of computations used for the remaining third. Regardless of the number of cores available, only two are needed and we can hope for a 66% increase in performance.

6 Conclusions and future work

In this note we have discussed the process of reimplementing the Castryck-Decru attack into a form compatible with SageMath. The need to avoid calls to slow Gröbner basis functions inspired the derivation of efficient, exact formulas for $(2,2)$-isogenies, which yielded significant performance enhancements. As our implementation currently stands, the code runs 8 to 15 times faster than the original Magma proof of concept.

However, as previously mentioned, since the publication of [CD22a] there was another, independently discovered attack by [MM22] which used similar machinery, but recovered Bob's secret isogeny directly from the kernel of the genus two (N_A, N_A)-isogeny. This attack inspired a short note [Oud22] which described how the Castryck-Decru attack could be modified to recover the secret isogeny as soon as the first call to the oracle returns `true`. A similar extension was also shared in the note [Wes22] and the higher dimension, proven polynomial time attack in [Rob22].

The direct computation attack following [Oud22] has been implemented in the same repository as this reimplementation and has resulted in a dramatic performance enhancement. The full secret isogeny can now be recovered in a matter of seconds (even in the highest parameter sets) once the first digits have been found. At the time of writing, this direct computation recovers Bob's key from the `$IKEp217` challenge in only nine seconds, `$IKEp434` in twenty seconds and `$IKEp751` in under an hour (note: here almost all of the compute time is spent on the 3^6 guesses of the first six digits). A summary of these results are shown in the last row of Table 2.

For this new direct computation, the bottleneck of the algorithm is to find the first β_1 digits of the secret. In the original attack, only auxiliary isogenies with a degree represented as the sum of two squares are considered. By modifying the degree of the auxiliary isogenies, we can remove some of the slowdown by picking "tweak values" to lower the value of β_1. This was already suggested as an extension in [CD22a], and discussed in more detail in [MM22, Rob22]. We defer further discussion of how tweaks are performed to the cited references.

To give an example of the power of the direct computation on large characteristic, our attack was extended to the `SIKEp964` parameter set.[13] As we can write $x = 2^a - 5 \cdot 3^b$ as the sum of two squares, we have $\beta_1 = 0$, and there is no need to make an initial guess. This fact allows the protocol to be broken deterministically in only 30 seconds. Note that, additionally, $x' = 2^{a-6} - 3^b$ can be written as the sum of two squares, and so could also be broken without any tweaks. Similarly, with tweaks to the auxiliary degree, the `$IKEp217` challenge is broken in only two seconds. A future project is to further explore tweaks to the auxiliary isogeny degree for other parameter sets.

For the SIKE parameters, where $\mathrm{End}(E_0)$ is known, the attack described in [Wes22] can be performed, and a degree x-isogeny can *always* be computed in polynomial time, meaning no guesses are *ever* required. This can be done easily when the factorisation of the degree is known, but the note further describes a method which does not rely on factoring the degree. We are unaware of a direct implementation of this attack at the time of writing this note.

Additionally, the attack can be generalised following [MM22] when $\mathrm{End}(E_0)$ is unknown, but this attack has sub-exponential complexity. However, [Rob22] shows that provable polynomial time attacks are possible even for random starting curves by considering attacks in dimension eight. The Maino-Martindale attack has a SageMath implementation which will be made available soon. Robert is working on adapting the AVIsogenies library [BCR] to allow for an efficient implementation of the dimension eight attack.

Another piece of work in progress is including the work of [Kun22] which describes an efficient algorithm for the genus two $(2^n, 2^n)$-isogeny chain. Kunzweiler additionally released Magma and SageMath implementations of her algorithm when the paper was published, and has been busy porting her work into the implementation in a fork of our repository. From initial tests, it seems that including Kunzweiler's algorithm offers an additional

[13] `SIKEp964` is defined by $a = 486$, $b = 301$, but was retired in round one of the NIST Post-Quantum Cryptography Project.

20% performance boost, which is particularly helpful when guessing the first digits of Bob's secret.

As for the future of isogeny-based cryptography, many schemes remain secure after the publication of these polynomial time torsion-point attacks, but it would be naïve not to recognise the change in perception such a brilliant attack will have on the area. As is the case when fantastic research is published, all that can happen next is more research. We look forward to seeing how everything evolves over the coming months and years.

A SageMath implementation of explicit formulas from Section 4

```
# Computation of elliptic curves gluing and image of (Pc,P) and (Qc,Q)
def FromProdToJac(C, E, P_c, Q_c, P, Q, a):
    Fp2 = C.base()
    Rx.<x> = PolynomialRing(Fp2)
    # Isotropic torsion subgroup (P_c2, P2), (Q_c2, Q2)
    P_c2 = 2^(a-1)*P_c
    Q_c2 = 2^(a-1)*Q_c
    P2 = 2^(a-1)*P
    Q2 = 2^(a-1)*Q
    # Corresponding roots
    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()

    # We need the image of (P_c, P) and (Q_c, Q) in J
    JPc = J([s1 * x^2 + s2 - P_c[0], Rx(P_c[1] / s1)])
    JQc = J([s1 * x^2 + s2 - Q_c[0], Rx(Q_c[1] / s1)])

    JP = J([(P[0] - t2) * x^2 - t1, P[1] * x^3 / t1])
    JQ = J([(Q[0] - t2) * x^2 - t1, Q[1] * x^3 / t1])

    imPcP = JP + JPc
    imQcQ = JQ + JQc
    return h, imPcP[0], imPcP[1], imQcQ[0], imQcQ[1]
```

```
# A class for computation of Richelot correspondence
# with shared precomputed variables.
class RichelotCorr:
    def __init__(self, G1, G2, H1, H2, hnew):
        assert G1[2].is_one() and G2[2].is_one()
        self.G1 = G1
        self.G2 = G2
        self.H1 = H1
        self.H11 = H1*H1
        self.H12 = H1*H2
        self.H22 = H2*H2
        self.hnew = hnew
        self.x = hnew.parent().gen()

    def map(self, D):
        "Computes (non-monic) Mumford coordinates for the image of D"
        U, V = D
        if not U[2].is_one():
            U = U / U[2]
        # Sum and product of (xa, xb)
        s, p = -U[1], U[0]
        # Compute X coordinates (non reduced, degree 4)
        g1red = self.G1 - U
        g2red = self.G2 - U
        assert g1red[2].is_zero() and g2red[2].is_zero()
        g11, g10 = g1red[1], g1red[0]
        g21, g20 = g2red[1], g2red[0]
        Px = (g11*g11*p + g11*g10*s + g10*g10) * self.H11 \
          + (2*g11*g21*p + (g11*g20+g21*g10)*s + 2*g10*g20) * self.H12 \
          + (g21*g21*p + g21*g20*s + g20*g20) * self.H22

        # Compute Y coordinates (non reduced, degree 3)
        assert V[2].is_zero()
        v1, v0 = V[1], V[0]
        Py2 = v1*v1*p + v1*v0*s + v0*v0
        Py1 = (2*v1*g11*p + v1*g10*s + v0*g11*s + 2*v0*g10)*self.x \
         - (v1*g11*s*p + 2*v1*g10*p + v0*g11*(s*s-2*p) + v0*g10*s)
        Py1 *= self.H1
        Py0 = self.H11 * U * (g11*g11*p + g11*g10*s + g10*g10)

        # Now reduce the divisor, and compute Cantor reduction.
        _, Py1inv, _ = Py1.xgcd(Px)
        Py = (- Py1inv * (Py2 * self.hnew + Py0)) % Px
        assert Px.degree() == 4
        assert Py.degree() == 3

        Dx = ((self.hnew - Py ** 2) // Px)
        Dy = (-Py) % Dx
        return (Dx, Dy)
```

References

[ACC+20] Reza Azarderakhsh, Matthew Campagna, Craig Costello, Luca De Feo, Basil Hess, Aaron Hutchinson, Amir Jalali, David Jao, Koray Karabina, Brian Koziel, Brian LaMacchia, Patrick Longa, Michael Naehrig, Geovandro Pereira, Joost Renes, Vladimir Soukharev, and David Urbanik. Supersingular isogeny key encapsulation, 2020. `http://sike.org`.

[BCP97] Wieb Bosma, John Cannon, and Catherine Playoust. The Magma algebra system. I. The user language. `http://dx.doi.org/10.1006/jsco.1996.0125`, 1997. Computational algebra and number theory (London, 1993).

[BCR] Gaetan Bisson, Romain Cosset, and Damien Robert. AVIsogenies, a library for computing isogenies between abelian varieties. Available at `https://www.math.u-bordeaux.fr/~damienrobert/avisogenies/`.

[BED21] Jérémy Berthomieu, Christian Eder, and Mohab Safey El Din. msolve. In *Proceedings of the 2021 on International Symposium on Symbolic and Algebraic Computation*. ACM, jul 2021. `https://doi.org/10.1145%2F3452143.3465545`.

[CD21] Wouter Castryck and Thomas Decru. Multiradical isogenies. Cryptology ePrint Archive, Paper 2021/1133, 2021. `https://eprint.iacr.org/2021/1133`.

[CD22a] Wouter Castryck and Thomas Decru. An efficient key recovery attack on SIDH (preliminary version). Cryptology ePrint Archive, Paper 2022/975, 2022. `https://eprint.iacr.org/2022/975`.

[CD22b] Wouter Castryck and Thomas Decru. Magma Implementation of the SIDH key recovery attack. `https://homes.esat.kuleuven.be/~wcastryc/`, 2022. [Last accessed 11-August-2022].

[CDS19] Wouter Castryck, Thomas Decru, and Benjamin Smith. Hash functions from superspecial genus-2 curves using Richelot isogenies, 2019. `https://arxiv.org/abs/1903.06451`.

[Cos21] Craig Costello. The case for SIKE: A decade of the supersingular isogeny problem. *IACR Cryptol. ePrint Arch.*, page 543, 2021. `https://eprint.iacr.org/2021/543`.

[Cry22] CryptoHack. A modern cryptography learning platform, 2022. `https://cryptohack.org`.

[DGPS22] Wolfram Decker, Gert-Martin Greuel, Gerhard Pfister, and Hans Schönemann. Singular 4-3-0 — A computer algebra system for polynomial computations. `http://www.singular.uni-kl.de`, 2022.

[dQKL+20] Victoria de Quehen, Péter Kutas, Chris Leonardi, Chloe Martindale, Lorenz Panny, Christophe Petit, and Katherine E. Stange. Improved torsion-point attacks on SIDH variants. Cryptology ePrint Archive, Paper 2020/633, 2020. `https://eprint.iacr.org/2020/633`.

[Fou22] Tako Boris Fouotsa. SIDH with masked torsion point images. Cryptology ePrint Archive, Paper 2022/1054, 2022. https://eprint.iacr.org/2022/1054.

[FT19] E. V. Flynn and Yan Bo Ti. Genus two isogeny cryptography. Cryptology ePrint Archive, Paper 2019/177, 2019. https://eprint.iacr.org/2019/177.

[GPST16] Steven D. Galbraith, Christophe Petit, Barak Shani, and Yan Bo Ti. On the security of supersingular isogeny cryptosystems. Cryptology ePrint Archive, Paper 2016/859, 2016. https://eprint.iacr.org/2016/859.

[GS] Daniel R. Grayson and Michael E. Stillman. Macaulay2, a software system for research in algebraic geometry. Available at http://www.math.uiuc.edu/Macaulay2/.

[HLP00] Everett W. Howe, Franck Leprevost, and Bjorn Poonen. Large torsion subgroups of split Jacobians of curves of genus two or three. *Forum Mathematicum*, 12(3):315–364, 2000. https://arxiv.org/abs/math/9809210.

[JdF11] David Jao and Luca de Feo. Towards Quantum-Resistant Cryptosystems from Supersingular Elliptic Curve Isogenies Post-Quantum Cryptography. In *Post-Quantum Cryptography*, volume 7071, pages 19–34. Springer Berlin / Heidelberg, November 2011. https://hal.inria.fr/hal-00652846.

[Kan97] Ernst Kani. The number of curves of genus two with elliptic differentials. 1997(485):93–122, 1997. https://doi.org/10.1515/crll.1997.485.93.

[KMP$^+$20] Péter Kutas, Chloe Martindale, Lorenz Panny, Christophe Petit, and Katherine E. Stange. Weak instances of SIDH variants under improved torsion-point attacks. Cryptology ePrint Archive, Report 2020/633, 2020. https://eprint.iacr.org/2020/633.

[KP22] Péter Kutas and Christophe Petit. Torsion point attacks on "SIDH-like" cryptosystems. Cryptology ePrint Archive, Paper 2022/654, 2022. https://eprint.iacr.org/2022/654.

[Kun22] Sabrina Kunzweiler. Efficient computation of $(2^n, 2^n)$-isogenies. Cryptology ePrint Archive, Paper 2022/990, 2022. https://eprint.iacr.org/2022/990.

[Mic21] Microsoft Research. SIKE Cryptographic Challenge, 2021. https://www.microsoft.com/en-us/msrc/sike-cryptographic-challenge.

[MM22] Luciano Maino and Chloe Martindale. An attack on SIDH with arbitrary starting curve. Cryptology ePrint Archive, Paper 2022/1026, 2022. https://eprint.iacr.org/2022/1026.

[Mor22] Tomoki Moriya. Masked-degree SIDH. Cryptology ePrint Archive, Paper 2022/1019, 2022. https://eprint.iacr.org/2022/1019.

[OPP$^+$22] Rémy Oudompheng, Lorenz Panny, Giacomo Pope, et al. SageMath Reimplementation of the SIDH key recovery attack. https://github.com/jack4818/Castryck-Decru-SageMath, 2022.

[Oud22] Rémy Oudompheng. A note on implementing direct isogeny determination in the Castryck-Decru SIKE attack. Online, 2022. https://www.normalesup.org/~oudomphe/textes/202208-castryck-decru-shortcut.pdf.

[Pet17] Christophe Petit. Faster algorithms for isogeny problems using torsion point images. Cryptology ePrint Archive, Paper 2017/571, 2017. https://eprint.iacr.org/2017/571.

[Rob22] Damien Robert. Breaking SIDH in polynomial time. Cryptology ePrint Archive, Paper 2022/1038, 2022. https://eprint.iacr.org/2022/1038.

[Sag22] Sage Developers. *SageMath, the Sage Mathematics Software System (Version 9.6)*, 2022. https://www.sagemath.org.

[Smi05] Benjamin Smith. Explicit endormorphisms and correspondences. PhD dissertation, 2005. available online at http://iml.univ-mrs.fr/~kohel/phd/thesis_smith.pdf.

[Wes22] Benjamin Wesolowski. Understanding and improving the Castryck-Decru attack on SIDH. Online, 2022. https://mycore.core-cloud.net/index.php/s/iW1psgMzoo8gFqe#pdfviewer.