An efficient implementation of the isogeny-based binSIDH and terSIDH key exchange protocols

Haydn Small (210182758)

May 2024

BSc Computer Science(Cyber Security)

Supervisor – Dr. Essam Ghadafi

Word count: 9567

# Abstract

With the development of quantum computers many public-key exchange protocols, will become insecure. To address this issue post quantum cryptography has been the focus of many cryptographic efforts to provide quantum secure protocols that are also practical. This dissertation provides a partially working C++ implementation of the binSIDH and terSIDH key exchange protocols, which are both quantum secure and isogeny-based. It also provides potential ways to fix some of the errors and build upon the provided implementation for future work. An analysis of the attacks on SIDH is also provided along with descriptions of a few protocols built around a variety of different countermeasures to these attacks along with their efficient implementations.

# Acknowledgements

I would like to thank my supervisor Dr. Essam Ghadafi for helping and guiding me throughout my dissertation.

# Declaration

I declare that this dissertation is my own work except where otherwise stated.

Contents

[Abstract 2](#_Toc166047261)

[Acknowledgements 2](#_Toc166047262)

[Declaration 2](#_Toc166047263)

[Introduction 4](#_Toc166047264)

[Motivation 4](#_Toc166047265)

[Aim and Objectives 4](#_Toc166047266)

[Project Structure 5](#_Toc166047267)

[Technical Background 5](#_Toc166047268)

[SIDH 5](#_Toc166047269)

[SIDH attacks 6](#_Toc166047270)

[Secure supersingular protocols 6](#_Toc166047271)

[BinSIDH and terSIDH 8](#_Toc166047272)

[OPRFs 8](#_Toc166047273)

[What was done and how? 8](#_Toc166047274)

[Methodology 8](#_Toc166047275)

[Requirements 9](#_Toc166047276)

[Design 9](#_Toc166047277)

[Technology/implementation 17](#_Toc166047278)

[Testing 18](#_Toc166047279)

[Results and Evaluations 18](#_Toc166047280)

[Results 18](#_Toc166047281)

[Evaluation of design 20](#_Toc166047282)

[Evaluation of implementation/technologies 20](#_Toc166047283)

[Evaluation of testing 21](#_Toc166047284)

[Conclusion 21](#_Toc166047285)

[Completion of objectives 21](#_Toc166047286)

[Future work 22](#_Toc166047287)

[References 22](#_Toc166047288)

# Introduction

## Motivation

Cryptography is essential for the secure communication of data between two parties. With the revelation of quantum computers many current cryptographic methods will be broken[1]. Namely the previously considered intractable factorisation and discrete logarithm problems are completely broken due to the application of Shor’s algorithm with quantum computers[2][3]. This means that current public key encryption schemes such as RSA and Diffie-Hellman will become insecure with the revolution of quantum computers.

One post-quantum cryptographic approach to key exchange that has been proposed is isogeny-based cryptography[4], which has smaller key sizes than other post-quantum approaches to public key exchange protocols at the cost of larger computing times, with SIDH-based systems being at the forefront of the area[5]. Recently, SIDH has been subject to a variety of post-quantum attacks, meaning that researchers have been in a race to attempt to create efficient countermeasures to these attacks.

To combat these attacks and the larger computing times created from the cost of currently proposed countermeasures to the attacks Basso et al.[6] presented the binSIDH and terSIDH protocols which feature new efficient countermeasures to the SIDH attacks. These protocols use artificial orientations, which are known as split Cartan structures to prevent the attacks on SIDH. Another very important suggestion in their proposal was the hybrid variants of each protocol. These allow for one side to compute larger bin or terSIDH-like isogenies, while the other side computes smaller SIDH-like isogenies. This could potentially pave the way for practical implementations of post-quantum isogeny based key exchange protocols, which with correct vetting and analysis of security from a variety of different researchers could become a widely used key exchange protocol in society. This would be very beneficial as quantum computers are coming and pose an inevitable threat to currently used key exchange protocols, so, by having an alternative that presents small key sizes, low computation time for the client and is quantum secure, we can look forward to a quantum-secure future. A proof-of-concept implementation of the bin and terSIDH protocols already exists using SageMath[7], which provides competitive run times with efficient implementations of other protocols that use different countermeasures to the SIDH attacks.

## Aim and Objectives

**Aim:** To create an efficient implementation of the binSIDH and terSIDH protocols for the revolution of quantum computers and compare its performance to similar implementations of it and other isogeny-based post-quantum cryptographic protocols, namely CSIDH[8], M-SIDH[9] and FESTA[10].

**Objectives:**

1. Research and summarise at least 3 attacks on SIDH isogeny-based protocols.
2. Research and summarise at least 3 other quantum secure isogeny-based cryptographic protocols and implementations of those protocols.
3. Implement binSIDH and terSIDH protocols and hybrid variants of each in C++.
4. Evaluate the space and time performance of my implementation against at least 3 isogeny-based post-quantum cryptographic protocols.
5. Deliver a report of findings and work done.

Due to timing constraints, I removed the objective in my project proposal, where I planned to implement an oblivious pseudo random function based on the binSIDH and terSIDH protocols, however, I still plan to talk about how this could be completed as future work in my conclusion as I feel this protocol would lead to significant speedups in quantum-secure isogeny-based OPRFs.

## Project Structure

In this piece, I start by identifying and evaluating the current attacks on SIDH based public key exchange protocols. I then go on to identify current SIDH-based protocols that use different countermeasures and efficient implementations of each. From here I go on to describe the methodology and requirements of my implementation. I then proceed to discuss and explain the class diagram and design of my implementations, before talking about the actual implementation and the technologies used during it, before discussing the approach to testing I opted to take. Following on from this I talk about the outputs produced by my project and evaluate how effective these are, as well as how they can be used for future work. I then evaluate my approach to the implementation of my protocol before forming a conclusion linking back to my aim and objectives as well as suggestions for future work.

# Technical Background

## SIDH

SIDH[5] was an advancement in elliptic curve cryptography in 2011 that was presumed to be quantum secure. The protocol used supersingular elliptic curves, which differ from ordinary elliptic curves in the sense that a group scheme of order p is disconnected (consists of only isolated points), the Hasse invariant of the curve is zero, and the kernel of the multiplication-by-p map consists of only the identity element. This is useful as it allows us to identify an isogeny just by specifying its kernel and allows us to compute an isogeny given a kernel subgroup using Vélu’s Formulas[11]. Another useful property is that they contain an isogeny invariant, meaning that if an isogeny maps a supersingular curve to another curve, the curve it has been mapped to will also be supersingular; this is a key property for the SIDH protocol. Supersingular curves also have a very large endomorphism ring compared to ordinary elliptic curves, meaning the potential mappings(isogenies) from one curve to another are very large making it useful for cryptography. The SIDH protocol works very similarly to the Diffie-Hellman protocol, where both parties will compute their public key to send across the network before each using their secret key to compute the shared key from the other parties' public key. The curves are defined over a finite field (a finite and cyclic group of numbers) and the public parameters are produced from the setup of the field. The finite field is defined as the square of prime p where p is a prime formed by two small primes multiplied by each other and a cofactor plus or minus 1. These two small primes are used as the degrees of the isogenies computed by the two separate parties on the curves. The public parameters are then the initial supersingular curve defined over the square of prime p, and the two separate values of P and Q as bases which are used to generate isogenies of the degrees specified by the two small primes. To compute the public key both parties perform a random walk(compute a chain of isogenies) from the initial curve to a curve defined by their secret key. They then share the curve they arrived at along with a torsion image of the isogeny computed on the public bases of the opposing party. Then to compute the shared key, they each compute their secret isogeny on the images provided by the public key to arrive at the same elliptic curves. The protocol then calculates the j-invariant (a property) of the elliptic curve and this value is used as the shared key. Montgomery curves are used as the supersingular curves as this allows a speedup due to the efficient arithmetic that can be performed on their Kummer line. This protocol was believed to be secure for until it was recently completely broken in July 2022. SIKE[12] a SIDH variant even made it to the fourth round of the 2022 NIST[13] post-quantum competition.

## SIDH attacks

In July 2022 an attack on SIDH was an efficient key recovery attack on SIDH[14]. This attack aims to leverage the torsion images provided by the SIDH protocol to recover the key and relies on the endomorphism ring of the supersingular curve and knowledge of the degree of the secret isogeny being known. This attack was used to prove the insecurity of SIKE and could recover the keys of the security levels specified by SIKE between 55 seconds and 3h 15 minutes for the varying levels on a single-core processor. The attack uses Kani’s reducibility theorem[15] to determine whether an isogeny emanating from a product of two elliptic curves takes us again to a product of elliptic curves instead of to the Jacobian of a genus 2 curve. The attack uses an auxiliary isogeny generated from the initial curve to another curve over the known endomorphism ring. Then using Kani’s criterion as a decision tool a chain of (2-2)-isogenies can be efficiently computed and a search-to-decision can be employed until the secret isogeny applied to the torsion images before they were released is discovered. This works on the assumption that landing on a product is extremely unlikely in the case that the public curve and the torsion point image are a valid public key tuple. The attack can be used to recover both parties’ secret keys. It can be optimised to run in polynomial time using non-scalar endomorphisms of other very small degrees, which if they occur on the starting curve can be found using the KLPT[16] algorithm to find the desired auxiliary isogeny. This can be done on any curve whose endomorphism is known. This attack has since been improved in a recent paper[17] to speed up the time to recover the secret isogeny.

In August 2022 an attack came to light that could recover the secret key without a known endomorphism ring[18]. This attack similarly to the one above relies on computing the product of two curves together, however rather than using a decision-based strategy this attack recovers the key directly. The attack aims to compute a polarised isogeny originating from the abelian surface produced by the product of the starting curve and the curve in the public key. The attacker chooses a curve E. 2 isogenies of E are then computed to get the starting curve of the public protocol and separate curve F defined by the attacker. A product polarization can then be formed between the product of E and the curve in the public key, and curve F and the starting curve of the protocol allowing for the computation of a polarized isogeny, allowing the secret isogeny to be recovered. Once again, this attack leverages information revealed from the torsion images revealed in SIDH.

Later in August 2022 a new attack, which built upon both previous attacks proved it was possible to break SIDH in polynomial time[19] even with a random starting curve. This attack shows that by going to dimension 8 over the elliptic curves it is possible to discover any hidden parameter of SIDH in polynomial time. The attack uses the isogeny information from the public starting curve and the public curve generated by the public key, to recover the kernel of the isogeny. Then by computing a diagonal embedding of the isogeny to obtain an isogeny between abelian surfaces to dimension 4, they can use an integral matrix also in the 4th dimension to compute an 8th dimension isogeny. From here the parameters of the SIDH protocol can be extracted through various techniques applied to the kernel of the 8th dimension isogeny. Like the above attacks, this attack is only possible because of the information released in the torsion points.

## Secure supersingular protocols

In January 2023 two new protocols (M-SIDH and MD-SIDH[9]), which are secure to the above attacks were proposed using either masking for the torsion point information or masking for the degree of the secret isogeny. Both protocols use a secret scalar to mask the information revealed by the public torsion points, with the masked degree protocol using isogenies that are no longer fixed and are random divisors of the secret isogeny, and the masked protocol using fixed isogenies whose degrees have 2 times the security parameter number of distinct prime divisors. Although adaptive versions of the previous attacks can be applied to these protocols, with correct parameter selection they are believed to be quantum secure. M-SIDH is believed to have the best security-efficiency trade-off out of the two however to provide security against adaptive attacks both protocols require very large parameters causing high running times and significant communication costs. A new version of M-SIDH[20] has since been developed using compression techniques and using Vélu’s formula on Montgomery curves, however even with these techniques M-SIDH is still lacking in performance for public use.

Although proposed in 2018 CSIDH[21] is another protocol in the supersingular isogeny field that is secure against the SIDH attacks. Unlike other protocols in the field, CSIDH does not reveal any torsion point data so is not susceptible to any of the previously mentioned SIDH attacks. Instead, the protocol relies on the application of ideal class groups and their commutativity through isogenies to perform a key exchange. Upon initial review, CSIDH seems like the ideal isogeny-based boasting small bandwidth requirements with the proposed parameter set. However, since its initial exposure, CSIDH has been susceptible to a variety of different attacks. Two variations of CSIDH(dCSIDH and CTIDH[8]) have since been developed to attempt to assess the practicality of the CSIDH protocol. dCSIDH aims to provide extra security against physical attacks by eliminating randomness requirements and using dummy operations to provide deterministic and dummy-free behaviour, whereas CTIDH aims to optimize purely for performance using batching techniques. Although these new protocols are significantly faster than other high-security CSIDH implementations (dCSIDH being 2 times faster than other implementations, and CTIDH being 3 times faster than dCSIDH) these protocols still have notable handshake latency and are predicted to be outperformed computationally by an efficient implementation of SWOOSH[22](a lattice-based protocol). However, CSIDH implementation still boasts significantly smaller key sizes.

In May 2023 a new protocol built upon theories developed in the SIDH attacks called FESTA[10] was developed. FESTA uses the attacks on SIDH to develop a family of quantum-resistant trapdoor functions. Like M-SIDH, FESTA scales the torsion images, but it differs as it uses a secret matrix to invert the scaling of the torsion images; from here an inverter can then apply the SIDH attacks to recover the input. It is proposed that an IND-CCA public key construction can then be constructed by combining OAEP[23] with the FESTA trapdoor function in the Quantum Random Oracle Model. This is done under the assumption the FESTA trapdoor function is a partial-domain trapdoor function, which is claimed with the provision of proofs. A proof-of-concept implementation of FESTA provides competitive running times with optimised implementations of other isogeny-based public key exchanges. Recently a new version of this protocol named QFESTA[24] has been proposed, which constructs a new algorithm based on quaternion algebra from the SIDH attack to compute an isogeny of non-smooth degree. As a result, the protocol only requires (2, 2)-isogeny computations and 3-isogeny computations compared to FESTA’s high-degree isogeny computations for key generation and encryption. QFESTA is also claimed to be IND-CCA secure with the provision of proofs and a proof-of-concept implementation suggests QFESTA uses smaller key sizes than FESTA. The comparison of the proof-of-concept implementations of FESTA and QFESTA suggests that QFESTA outperforms FESTA at 128-bit, 192-bit and 256-bit security levels, with the largest differences in runtime being at the 256-bit level.

## BinSIDH and terSIDH

The binSIDH and terSIDH protocols were proposed by Basso et al.[6] in December 2023 proposed new efficient countermeasures to the SIDH attack. This approach aimed to use artificial orientations also known in mathematics as split Cartan structures to defend against the SIDH attacks. The terSIDH protocol provides a slightly faster speeds than the binSIDH protocol, however, this comes at the cost of relying on variable degree isogenies. They also provide hybrid versions of each, which allow for one side to compute smaller SIDH-like isogenies at the cost of the other side computing much larger binSIDH or terSIDH like isogenies. This is a large advantage of this proposal, the prospect of offloading most of the computation onto a server and allowing a client to undertake less of the computing load could allow for practical run times with an efficient implementation. This would be the first practical run time implementation of a quantum secure isogeny-based public key exchange protocol and could solve the current threat quantum computers pose to public key exchange protocols. An advantage this approach could have over lattice-based approaches is the difference in key sizes, lattice-based approaches are known for having lower run times at the cost of larger key sizes, so with this implementation if we can reach a practical run time while maintaining smaller key sizes it could be a suitable contender.

## OPRFs

An oblivious pseudorandom function(OPRF) is a protocol where an interaction between a server has a key for a secure pseudorandom function stored on the server and the client has an input for the function. The client only learns the outcome of the function given the key and the client input and the server learns nothing. The protocol is verifiable if the client can verify that the outcome of the secure pseudorandom function is correct from a key the client previously sent to the server. Post-quantum verifiable OPRFs based on isogenies are hindered by the inefficiency of SIDH countermeasures with an M-SIDH-based implementation[25] requiring 8.7MB of bandwidth to complete and lattice-based variations requiring significantly larger bandwidth due to their large key sizes[26]. There also exists SIDH-based OPRFs which are no longer secure but more notably a CSIDH-based OPRF, which achieves a significantly smaller bandwidth at 424KB[27]; however, this is unfortunately not verifiable.

# What was done and how?

## Methodology

For my methodology I opted to practice a waterfall approach to my implementation. This is a change from the iterative approach I suggested in my project proposal; I decided to make this change as I quickly realised a lot of the test data I would need, such as the finite field with the desired properties and torsion points over Split Cartan level structures would be implemented later in my code. Since these would be used to test all the underlying functions, I decided to complete the full implementation first, and then test after. In addition to this a proof-of-concept[28] implementation of the bin/terSIDH protocols had already been implemented, meaning I was able to get a good grasp of the requirements and structure of the project before starting the implementation. I planned to base my approach heavily on the structure of this proof-of-concept implementation, making it ideal for a waterfall approach.

## Requirements

For my implementation to be considered successful it is required that the calculation of the shared key from the public key of A and private key of B is equal to the shared key calculated from the public key of B and the private key of A. I also aim to allow the user to choose between regular and hybrid versions of the protocol through the command line. As well as 128-bit, 192-bit and 256-bit security.

## Design

For the design of my implementation, I decided to take an object orientated approach to the as this would allow to create objects from classes, which I use throughout my implementation. I use a variety of techniques for speedup, such as the computation of the codomain using Vélu's formula[29], using Vélu's formula for the faster computation of isogenies over large prime degrees[30], Miller’s Weil pairing algorithm[31], using efficient x-only Montgomery Isogenies of even[32] and odd degree isogenies[33].

A screenshot of a computer

Description automatically generated

For the design of the isogenies, I decided it would be beneficial to use x-only Kummer points which allow for more efficient computation. A library implementation of these methods already exists[34] in python using sage math, therefore my approach is heavily based upon this and utilises a variety of useful attributes and methods, which I planned to implement an equivalent C++ implementation of. The KummerLine and KummerPoint classes provide x-only arithmetic on Montgomery elliptic curves which are packaged into the classes. These classes make use of mathematical theorem on Kummer surfaces[35] to represent torsion data.

The KummerLine class provides the functionality to allow for calculating the j-invariant, checking if a KummerLine is an isomorphic transformation from another KummerLine by checking if the j invariant of the KummerLine is equal to the j invariant of the other KummerLine. The curve\_j\_invariant method is very useful and allows us to calculate the j invariant of the associated elliptic curve, which is critical to the protocol as the curve j invariant acts as the shared key. This is implemented using a modification of the code from the sage math library[36] to calculate the j invariant of specifically a Montgomery elliptic curve.

The KummerPoint class allows the representation of a Point on a KummerLine and is represented by the projective coordinates (X:Z) stored as p\_x and p\_z. The x() method allows for the calculation of the affine x coordinate from projective coordinates which can then be used to calculate the affine y coordinate. The xDBL, xADD and xDBLADD methods implement algorithms from efficient Montgomery theory to double a point, perform differential addition between two points and compute a step in a Montgomery ladder by performing simultaneous doubling and differential addition[37]. This concept is important as it allows us to exploit some of the overlap in the computations in the xADD and xDBL methods by combining them into one method. The multiply method initialises the loop for the Montgomery ladder initialising the parameters for the xDBLADD method and calling it at each step. The ladder\_3\_pt method allows us to perform efficient calculation of a point added to a multiplication of another point. The add and twice methods allow easy and correct initialisation of the xDBL and xADD methods, whereas the multiples methods return all points associated with an associated Kummer line.

The KummerLineIsogenyGeneric class acts as a parent class to composite, generic and Vélu implementations of isogenies. It is needed in the case of composite isogenies where a chain of different types of isogenies may be executed on a point. By having a parent class, it allows us to store all the child classes in a vector of type pointers to the parent class and access the call method without throwing any compiler errors. It is important that we can access the call method as this is what allows us to use the isogenies to map one KummerPoint to another KummerPoint.

A diagram of a computer

Description automatically generated with medium confidence

Continuing with the design of isogenies the above image shows the implementation of isogenies using either Vélu’s formula for large prime degrees or alternative isogenies for smaller degrees. The KummerLineIsogenyComposite in combination with the evaluate isogenies class allows us to compute a chain of isogenies on a point.

The Helper class allows us to factor a degree or find the index of a vector of addresses, where the address contains the data, you are searching for. The find\_index method contains a simple implementation, whereas the factoring method makes use of the trail prime factorisation algorithm to efficiently compute the factors[38].

The KummerLineIsogeny class is based on torsion data of even[32] and odd degrees[33] and allows storage of the degree, the kernel, the domain, the codomain, and the Edwards multiples which are precomputed once using the precompute\_edwards\_multiplers(d) method. The Edwards multiples are crucial as once computed they allow us to speed up the calculation of isogenies of odd degree torsion points. The compute\_constants() method makes use of the birational equivalence between the x coordinate of Montgomery curves and the y coordinate of twisted Edwards curves to allow us to efficiently compute the codomain of the isogeny. The evaluate\_isogeny(P) method makes use of the precomputed Edwards multiples to map a KummerPoint P to another KummerPoint through the isogeny. The compute\_codomain\_constants\_even() makes use of group law theory that allows us to extract the codomain constants of a Montgomery curve easily and efficiently, provided the kernel does not contain the KummerPoint (0, 0). The evaluate\_isogeny\_even(P) method then makes use of x-only projective coordinates to calculate the isogeny of an isogeny of degree 2, while avoiding the expensive square root computations that are otherwise required without knowledge of a point of order 8 above (0, 0). The compute\_codomain() method allows for the precompution of the codomain constant’s and calls the corresponding method either 2-degree isogeny or otherwise. This method is called inside the constructor, as the precomputation must be done before the class can be used to compute an isogeny of a KummerPoint. The call(P) method then evaluates the isogeny mapping of a KummerPoint and returns the KummerPoint it was mapped to.

The KummerLineIsogeny\_VeluSqrt class stores similar data to the KummerLineIsogeny class with some extra variables to store the results of some precomputation operations. The hI\_precomputation(b, c) method which creates product tree constructed from polynomials over a finite field relevant to the kernel of the isogeny. The hI\_resultant(poly) method allows us to calculate the polynomial resultant off any polynomial over a finite field applied to the product tree. This allows us to speedup efficiency as this means the product tree of polynomials has already been precomputed allowing us to calculate the polynomial resultant off different polynomials applied to the product tree, which is critical to using Vélu’s formulas to speedup isogeny calculation. The Fs(X1, X2) method is used to calculates the biquadratic polynomials associated given two polynomials X1, and X2. The EJ\_precomputation(b) method then iteratively uses the Fs method for b iterations until biquadratic polynomials are computed and stored in the EJ\_parts attribute. This is an important step as it precomputes data relevant to the computation of the isogenies and the codomain constants. The hK\_precomputation(b, c) method returns a product of the polynomials generated with relation to the kernel and degree, which will be used to both compute the codomain constants and the isogeny. The compute\_codomain\_constants() method makes use of the precomputed data and birationally equivalence between Montgomery curves and twisted Edwards curves to efficiently compute the codomain constants[29]. The compute\_codomain() methods initialise the KummerLine for the codomain from the finite field and the codomain constants calculated from compute\_codomain\_constants(). The constructor initialises all the attributes from parameters, through calling the precomputation methods and through calling the compute\_codomain method. The evaluate\_isogeny(P) method then makes use of the precomputed data to calculate polynomial resultants of the x-only KummerPoint which allow for a speedup in the calculation of the isogeny applied to a KummerPoint. The call method simply acts as an override to the virtual call(P) method in the parent class and will call the evaluate\_isogeny(P) method allowing an isogeny on a KummerPoint P to be calculated.

The EvaluateIsogenies class allows for the computation of chain of prime degree isogenies. The recursive\_sparse\_isogeny(Q, l, k, split, algorithm) method computes a chain of isogenies on with Q as the kernel and l as the degree k times. The algorithm determines the type of class to create either KummerLineIsogeny or KummerLineIsogeny\_VeluSqrt with values of l > 1000 being computed using the KummerLineIsogeny\_VeluSqrt class and values of l <= 1000 using the KummerLineIsogeny class. The split variable is used to determine whether most of the computation will take place and is a value between 0 and 1. For values closer to 0 the focus will be on computing more isogenies whereas with values closer to 1 will focus the algorithm more on multiplication computations as supposed to isogenies. My implementation uses a value of 0.8 for the split variable as this is shown to provide an optimised performance[5]. The sparse\_isogeny\_prime\_power(P, l, e, split, threshold) method is used to call the recursive\_sparse\_isogeny method and determines checks if l is greater than the threshold, if so, the isogenies computed will use the KummerLineIsogeny\_VeluSqrt class and if not, they will use the KummerLineIsogeny class. For my implementation the value of threshold is set to 1000 as this is where the VeluSqrt implementation starts to outperform the regular implementation. The factored\_kummer\_isogeny(K, P, order, threshold) method is used to generate a chain of prime isogenies formed from the prime factors of the order. K acts as the domain of the isogenies and P acts as the kernel, after a set of prime factors l^e the kernel point P is evaluated through that chain before being used as a kernel for the next chain of l^e where l is the prime factor and e is the number of times the factor appears in the order. The evaluate\_factored\_kummer\_isogeny(phi\_list, P) method accepts a chain of isogenies and a KummerPoint P, the chain of isogenies are then computed on the KummerPoint P to produce another KummerPoint P which is then returned.

The KummerLineIsogenyComposite class allows storage of a chain of isogenies, the degree of the combined isogenies, the domain, and the codomain. The constructor takes in the domain, kernel, degree, and threshold and calls the factored\_kummer\_isogeny method which provides the chain of isogenies computed from the degree, domain, and kernel. The domain of the class is then set to the first isogeny in the list of isogenies and the codomain is set to the last isogeny in the list of isogenies. The call(P) method overrides the call method from the parent class and makes use of this precomputed data and will call evaluate\_factored\_kummer\_isogenies method passing in the chain of isogenies precomputed and the KummerPoint P passed to the call method.

A screenshot of a computer

Description automatically generated

For generating the torsion data, I opted to create a class to represent an elliptic curve and a class to represent a point on an elliptic curve and perform some operations relevant to generating the torsion data using a split Cartan structure. The TorsionData class makes use of these classes and helper functions provided by the ProductTree and helper class to generate the torsion data for the binSIDH and terSIDH protocols.

The EllipticCurve class contains the curve coefficients a1, …, a6 in the format:

y^2 + a1xy + a3y = x^3 + a2x^2 + a4x + a6

However, for my implementation I will only be using Montgomery curves so will only be setting a4 to 1 and a2 to 6. The class also contains a finite field, the order of the finite field and the characteristic of the finite field; this data will be relevant for calculations involving the elliptic curve. The Elliptic curve provides basic constructor methods to allow for the creation of an elliptic curve as well as the equal(other) method which allows us to check if the curve is equal to another curve.

The Point class allows storage of the projective (X:Y:Z) coordinates as polynomials over a finite field in the attributes p\_x, p\_y and p\_z. The order allows us to store the order of the finite field and the curve allows us to determine on what curve the point exists. The line\_slope(P, Q) simply calculates the gradient of the line between two points P and Q and serves as a helper function to the line\_equation method. The line\_equation(E, P, Q, D) method allows us to calculate the equation of the line between two points R and P on curve E, with point Q adding some extra relevant data to the calculation. This method serves as a helper function to the millers\_loop method. The millers\_loop(E, P, Q, D) method is an important function that is used to assist in calculating the Weil pairing between two points on an elliptic curve and follows an efficient implementation Algorithm 4.2 proposed by Blake et al.[39] making use of the binary representation of D. E is the elliptic curve, P and Q are points and D is the order of the elliptic curve, which is interpreted in binary for the sake of efficiency. This method is necessary to my design as I will need to be able to generate correct torsion data otherwise the implementation will be susceptible to the SIDH attacks mentioned in the technical background. The weil\_pairing(E, P, Q, D, prng) makes use of the miller’s loop method to calculate the Weil pairing between two points E and Q, allowing us to correctly generate torsion data. The line\_equation, millers\_loop and weil\_pairing methods are all sourced and adapted from Stetsyk’s[40] implementation in C++. The equal method simply allows us to check if a point is equal to another point, and the design of the add\_points, subtract and scalar multiplication methods is not dissimilar from the designs of these methods for KummerPoints with the main difference being calculating the y-coordinate as well, which is important for the purpose of generating torsion data. I have also chosen to implement a generate\_random\_element(E, prng) method which will randomly generate a point on the elliptic curve E using the prng as a pseudo random number generator. This is an important method as without this I would be unable to generate random data for the torsion data, making the protocol insecure.

The TorsionData class allows us to generate the torsion data for our protocol, which is critical to ensure our protocol is secure against the SIDH attacks names in the technical background. The two batch\_cofactor\_mul\_generic methods are helper functions which recursively apply the group action to an element in a list of group elements using the prime factors of the group order. They are used to determine if removing any prime factors of the order gives us the identity of the group, which if they did would reveal extra information about the torsion data making them susceptible to attacks. The has\_order\_constants method is used to generate a constant used to generate the first group element from the order and the prime factors of the order which will be used in the batch\_cofactor\_mul\_generic methods. The has\_order\_D methods check that a group element G has order exactly D, allowing for both multiplicative and additive groups, this is much faster than determining an elements order generally and is enough for the checks involved in computing torsion points. The generate\_random\_point method makes use of the generate\_random\_element method provided by the Point class to generate a random point on an elliptic curve. This is necessary to ensure randomness when computing torsion data for the protocol. The generate\_point\_of\_order\_D(E, D, p, prng), which iteratively generates points on curve E until the has\_order\_D method returns true on the point. D is an integer which divides p + 1, this is important in ensuring D divides the one of the two large factors involved in generating the split Cartan level structure, ensuring that D belongs to exclusively one of the two groups generated. The generate\_linearly\_independent(E, P, D, p, prng) generates a random point of order D. It Then checks if that point is linearly independent from P. It does this by calculating the Weil pairing between the two points. The Weil pairing is then checked to ensure it has order D and if so, the linearly independent point is returned. This method is critical to the generation of the torsion data as without this method we are unable to determine whether the torsion data generated contains two points from same side of the split Cartan structure, which is necessary to ensure the correct computation of the secret key from both parties. The torsion\_basis method makes use of the underlying methods to generate two linearly independent points of order D. These points are then returned allowing the protocol to make use of the data generated to construct a protocol secure against the SIDH attacks.

A screenshot of a computer screen

Description automatically generated

The bin\_terSIDH class allows for the setup and running of the bin and terSIDH protocols. The make\_prime(p, f) function ensures that a multiple of the two large integers used to generate the split Cartan level structure multiplied together is converted into a prime by subtracting 1. This is necessary as for a lot of the underlying operations to work it relies on the finite field being constructed from a prime number. At the same time, we must be able to reliably deduce two factors which we are using for the split Cartan level structure. We use passing f by reference to assign to f the integer that the two large integers of the multiplicative groups are multiplied by so that we can easily access the data we need. The compute\_kernel\_scalars(secret, Alice) method generates the secret scalars for the torsion points to compute isogeny kernels and the orders of those points, the parameter Alice is used to determine whether generating scalars for Alice or Bob. This is important for the correct execution of the protocol. The keygen(A\_points, B\_points, sk\_choices, t, KummerLine, modulus, prng, Alice) method is used to generate a public and private key pairing for one party. To generate the public and private key for A we would pass A’s torsion data into A\_points and B’s torsion data into B\_points and for B we would do the opposite. The protocol first generates the secret data from sk\_choices and t using the compute\_kernel\_scalars method, then we compute the 2 isogeny kernels from A\_points. We setup the first isogeny from the KummerLine, the first isogeny kernel, and the order associated with that isogeny kernel. Then compute the first isogeny on the second isogeny kernel and on the auxiliary data provided by B\_points. We then setup the second isogeny from the codomain of the first isogeny, the second isogeny kernel, and the order associated with the second isogeny kernel. Then evaluate the second isogeny on the auxiliary data. We then generate a masking value and an inverse too it and we scale one of the two auxiliary points by each. Then return the secret key as the set of kernel scalars used for the isogenies and the orders associated with each, and we return the public key as the codomain of the second isogeny and the final computation of the two auxiliary points. To compute the shared key, we use the shared(sk, KummerLine, P, Q) method, this method uses the scalars stored in the secret key to scale the torsion points in the public key two compute two isogeny kernels. We then setup the first isogeny using the KummerLine associated with the public key, the first isogeny kernel and the order associated with the first isogeny scalar. We then compute the isogeny on the second isogeny kernel. Then setup the second isogeny using the codomain of the first isogeny, the second isogeny kernel, and the order associated with the second isogeny kernel. We then return the j\_invariant of the Montgomery curve associated with the codomain of the second isogeny as this is the shared key. The bin\_terSIDH\_core(t, sk\_choices) method initiates the protocol where t is the security parameter of the protocol and sk\_choices is the choices from which each digit of the secret key can be set too: [0, 1, 2] for terSIDH and [1, 2] for binSIDH. The method sets up the finite field and generates the torsion data for each side of the split Cartan level structure. It then calls the previous methods to generate keys for A and B, compute the shared keys of A and B and check that the shared key computed by A is equal to the shared key computed by B. It also displays the time before and after calling the keygen method and the shared method as this is important in evaluating the performance of my implementation.

The bin\_terSIDH\_\_hybrid class allows for the setup and running of the hybrid versions of the bin and terSIDH protocols. This is an important feature as one of the main benefits of the bin and terSIDH protocols is the opportunity to use the hybrid variants which offload a lot of the computational load onto one side of the key exchange, which could allow for practical runtimes between a client and a server. The compute\_kernel\_scalars\_hybrid(secret) method is used to generate the secret scalars to scale the isogeny kernels by, and the order of the isogeny kernels. The keygenA(prng, A, B, xQA, xPA, xPQA, xPB, xQB, KummerLine) method allows for the computation of SIDH like isogenies and requires smaller computations. The method generates a random integer between 0 and A and sets is as the secret key. The isogeny kernel is then computed from the points xQA, xPA and xPQA passed in and the secret key. An isogeny is then setup using the KummerLine, the kernel computed and A as the degree. The isogeny is then applied to the torsion points xPB and xQB and one is masked by random masking data generated from B and the other is masked by the inverse of the masking data. The secret key is then returned, and the public key is returned as the codomain of the isogeny and the masked torsion points. The keygenB(sk\_choices, t, xPB, xQB, KummerLine, xPA, xQA, xPQA) method allows for the computation of longer orientated isogenies and requires larger computations. The method makes use of the compute\_kernel\_scalars\_hybrid method to compute the secret kernel scalars and the orders associated with those kernels. The isogeny kernels xPB and xQB are then scaled by the secret scalars. The first isogeny is then setup using the KummerLine, the first computed isogeny kernel, and the order associated with the first isogeny kernel. The first isogeny is then applied to the second isogeny kernel, along with the torsion data xPA, xQA and xPQA. The second isogeny is then setup using the codomain of the first isogeny, the second isogeny kernel and the order associated with the second isogeny. The second isogeny is then applied to the torsion data. The data generated by compute\_kernel\_scalars\_hybrid is returned as the secret key and the codomain of the second isogeny, and the torsion data passed through both isogenies is returned as the public key. The sharedA(skA, EB, RA, SA, RSA, A) method allows for the computation of the shared key provided with A’s secret key and B’s public key. The isogeny kernel is computed from the torsion points SA, RA and RSA, and the skA. An isogeny is then setup using the KummerLine EB, the kernel generated from the torsion points, and A is used as the degree. The j\_invariant of the Montgomery curve associated with the codomain of the isogeny is then returned as the shared secret. The sharedB(skB, EA, RB, SB) method allows for the computation of the shared key provided B’s secret key and A’s public key. This is done by computing the 2 isogeny kernels from the torsion data RB and SB, along with the secret kernel scalars provided by skB. The first isogeny is then setup using the KummerLine EA, the first isogeny kernel computed, and the order associated with it stored in skB. The first isogeny is then applied to the second isogeny kernel. The second isogeny is then setup using the codomain of the first isogeny, the second isogeny kernel, and the order associated with the second isogeny kernel stored in skB. The j\_invariant of the Montgomery curve associated with the codomain of the second isogeny is then returned as the shared key. The bin\_terSIDH\_\_hybrid\_core(t, sk\_choices, security) method computes a finite field or the size relevant to the security and t parameters. It then generates the torsion points for A and B before calling both keygen methods and both shared methods. The time is output before and after each method call to allow us to measure the performance of my implementation. I also check that the shared key computed by A is equal to the shared key computed by B as if not the key exchange protocol has failed and is not working as expected.

For the testing class I implement a core\_testing() method which will call all the testing methods and run them to ensure the implementation is running as expected. The bin\_terSIDH\_tests() allow us to run the protocol for all the available parameter options of the bin and terSIDH protocol, and the bin\_terSIDH\_hybrid\_tests() allow us to run the protocol for all the available parameter options of the bin and terSIDH hybrid variants.

The main method allows for input parsing and either calls the core\_testing() method or calls one of the core methods associated with the regular or hybrid variants of the implementation passing in the associated parameters with the users’ inputs.

## Technology/implementation

I chose to use Microsoft Visual Studio as my IDE with MSVC C++ version 20 as my compiler. This is because the IDE and compiler provide a variety of features such as debugging features, object orientated support, checking for correct syntax, compiling, and running a project with code located in a variety of different files.

I had initially planned to make use of C++ elliptic curve libraries to implement some of the basic functionality. I initially looked at PQCrypto-SIDH[41], however due to how interlinked a lot of the functionality was I found it difficult to extract any useful underlying functionality from this library. The second library I looked at was libff[42], which provided a variety of helpful functionality for elliptic curves and finite fields in C++, however due to compatibility issues with windows I was unable to get this library to work on my computer without emulating through Ubuntu. I opted not to carry out my implementation in Ubuntu as it seemed counter intuitive for an efficient implementation to be run in a virtual machine, which would hinder the performance. I also looked at the GaloisCPP[43] library, however it lacked some core functionality such as calculating the characteristic of a finite field and dealing with large numbers.

As such I decided to start my implementation with the idea that I would implement the finite field as its own class and polynomials over the finite field. However, later into my implementation I came to the realization that my implementation of these classes could not deal with numbers of the size my implementation required and was likely an overcomplication.

I decided to check for any potential libraries which was when I found the FLINT: Fast number library for number theory[44]. This library provides efficient implementations and functionality for finite fields and polynomials over finite fields which are relevant for my implementation. The library also provides an efficient way to work with large numbers which cannot be stored in the default C++ variables due to being too large. This is important as the protocol deals with numbers too large for the default C++ variables.

## Testing

For testing I chose to test the overall output from the protocols as supposed to unit testing. I opted for this approach as due to the design of my protocol a lot of the code was interlinked making it difficult to individually test each unit. By testing the overall output of my protocol, I hoped to gain an understanding of the speeds achieved by the protocol as well as it’s correctness. In the event of issues in the execution of my code I aimed to use the debugger and std::cout statements at critical points to attempt to identify the location of the problems as well as in the future implement some more in depth testing around the specific areas.

# Results and Evaluations

## Results

For the outputs of my dissertation, I have the C++ code used to implement all the optimisations, which is in large part complete except for the presence of some run time access violation errors, which will need fixing for future work. This code base provides efficient functionality for the computation of chain isogenies using a variety of different efficient approaches, and the generation of torsion data for split Cartan structures in C++. It also provides efficient implementations of the bin and terSIDH protocols in C++. In addition to this there is a binary product tree implementation, an elliptic curve implementation including an implementation for points on an elliptic curve and relevant functions such as calculating the Weil pairing. My code base also provides an implementation of x-only KummerLines and KummerPoints constructed from Montgomery curves. For these to be accessed it would require a small amount of debugging and more thorough testing, however once done, would pave the way for a new efficient isogeny-based key exchange protocol, which could potentially have practical run times. There are also no C++ implementations of some of the underlying functionality presented in my protocols such as the generation of torsion data for split Cartan structures as well as the computation of chain isogenies using a variety of different efficient approaches. This functionality could be used to help implement other C++ cryptographic protocols in the future.

In addition to this the core code used to run and setup the bin and terSIDH protocols and hybrid variants is implemented in C++. Although currently we cannot run this due to access violation errors at run time. With some future work this implementation could provide practical run times for an isogeny-based key exchange protocol. Namely the hybrid variants which can offload most of the computation on to a server allowing a client to compute smaller SIDH-like isogenies. This could be critical for the future as if fixed, we would have a quantum secure public key exchange protocol that offers small key sizes and practical computation times, meaning we could look forward to a quantum secure society where it could replace the currently used public key exchange protocols.

Currently the run time errors in my code are related to access violations, I predict these could be prevented if instead of using \* as a pointer, I could instead use the built-in smart pointers provided by the C++. These include std::unique\_ptr, std::shared\_ptr and std::weak\_ptr classes provided by C++. These classes provide functionality to automatically manage memory, meaning we should be able resolve the current errors in my implementation and protect against any memory errors. This is critical to working on my implementation as the continued presence of memory errors will lead to impacts in both the performance and execution of my implementation.

Currently the outputs of my code are limited to:

The generation of the values used for the split Cartan structure.

A black background with white numbers

Description automatically generated

The generation of the size of the finite field using the prime formed from (a multiple of A\*B) -1. This takes 2 minutes and 14 seconds, however this is just the precomputation of data, so can be calculated and stored before generating the public-private key pairings and generating the shared key.

A number on a black background

Description automatically generated

The generation of the elliptic curve and the order associated with it. This is a useful output as the implementations of elliptic curves that allows for the calculation of a Weil pairing is not publicly implemented in C++.  
A black background with white numbers

Description automatically generated

Then during the generation of the torsion data, we encounter an access violation error when attempting to call the Point constructor, when attempting to generate a random point.  
A black screen with white text

Description automatically generated

This is likely caused by a Null pointer error and could most likely be avoided in my implementation if instead of using \* to denote a pointer I use on of the prebuilt classes such as the smart pointer classes provided by C++. These would allow me to denote unique and shared pointers with automatic memory management.

I can also present the background research I have undertaken as an output from my project. This provides a detailed background into the attacks on supersingular-isogeny-based approaches to public key exchange, and an overview of current implementations of other isogeny-based protocols resistant to the SIDH attacks. This will allow for comparison between a fixed version of my implementation and these implementations, to determine whether bin and terSIDH can provide practical times for a client.

While my current results have obvious limitation because the implementation currently cannot run without the presence of access violation errors. I believe that with future work this implementation could lead to an improvement on current isogeny-based key exchange implementations, potentially even allowing for practical runtimes.

## Evaluation of design

For my approach to the implementation, I opted to choose a waterfall methodology for my implementation. While I do believe this approach suited my implementation due to designing the full protocol before the implementation it did include some oversights into potential risks to my project. However, given the fact that there exists a proof-of-concept implementation I still believe that a waterfall methodology was the methodology most suited for my implementation, rather I suggest a change in my approach to the testing of the protocol would have been more suitable.

The object orientated approach I took to the implementation was the correct choice. Due to the nature of isogenies and points it allowed me to easily create new objects from the classes without needing to rewrite code as would be needed in a functional approach. Another benefit of the object orientated approach is in C++ it allows for easier management of memory. This could be done using the smart pointers provided by C++ I mentioned earlier as a potential fix to my current implementation.

## Evaluation of implementation/technologies

For my implementation, I believe the choice of using the FLINT library[44] was correct and suited my implementation. However, upon evaluation, I had initially spent time looking at libraries to help with my implementation, where I believed I was unable to find a library that could help with my implementation. At this point I decided to spend time implementing my own versions of a finite field and polynomials over a finite field, as I became concerned I was spending too much time trying to get libraries to work and would end up having to write it from scratch anyway. This was clearly an oversight on my part, as if I had broadened the scope of my searching of libraries to large number libraries and finite fields, as supposed to mainly attempting to find elliptic curve implementations I believe I would have found the FLINT library[44] earlier and my implementation would likely be complete by now.

I believe the choice of using the Microsoft visual studio as my IDE with MSVC C++ version 20 as my compiler was the correct choice. The IDE provided me with many useful tools such as debugging and syntax checks. It also allowed me to link the external libraries I had downloaded windows ports for using the vcpkg[45] package manager. It also allowed me to link multiple files together as a project, which served useful for my implementation as I used a variety of C++ code files and header files throughout my implementation.

## Evaluation of testing

I believe a change in my approach to testing of the protocol would be suitable. I believe a test-driven development process, whereby I write the tests before implementing the code would have been a more concrete way to develop the protocol. By doing this I would be able to ensure that at each point during development every bit worked as I expected. With the way I approached the implementation it meant that I ended up with code that is complete, however, encounters run time errors, which I need to then test by using the debugger and by using print statements. With designing class specific tests after as a final resort if the program executed but the shared key created by parties A and B is different. A test-driven approach would avoid this roundabout approach to implementing then designing test and then re implementing, by simply writing tests first I would be able to write tests and then implement and see immediately whether my implementations of the underlying functions work as expected.

# Conclusion

## Completion of objectives

In this work I have completed an implementation of the binSIDH and terSIDH protocols, however, the implementation runs into some memory errors associated with pointers, which I believe can be fixed in the future. I have also successfully researched and evaluated attacks on the SIDH protocol and researched SIDH-based key exchange implementations using a variety of different countermeasures.

As far as the completion of specific objectives I believe I have met objectives 1 and 2 with the research provided in my technical background. I also believe I have met objective 5 by completing this report and reporting the findings of my work. However, as far as meeting objectives 3 and 4 I believe the extent to which I have met each of them is limited. While I have completed an implementation of the binSIDH and terSIDH protocols and the hybrid variants in C++, due to the memory allocation errors associated with the pointers, I don’t have any concrete results to prove the correctness of the protocol nor the efficiency. As such I would say due to timing issues I have partially completed objective 3 and have not been able to complete objective 4. Although with the research I have provided, once fixed it would be easy to quickly compare my implementations against the other contenders in the field.

I conclude that I have only partially reached the aim of this project, as the aim of this project was to create an efficient implementation of the binSIDH and terSIDH protocols and compare it to efficient implementations of CSIDH, M-SIDH and FESTA, which I have been unable to complete. Although I have researched efficient implementations of CSIDH, M-SIDH and FESTA, as well as mostly implemented the binSIDH and terSIDH protocols with the implementations, currently being limited by the memory management of pointers.

## Future work

For future work, I believe the errors in my implementation could be amended by using smart pointers provided by the built-in C++ library as well as correct allocation of memory. In addition to this, I would then also suggest comparing the performance of the fixed implementation against the efficient implementation of SIDH-based protocols provided in the technical background.

Building upon this further I believe that the binSIDH and terSIDH protocols could be used to lead to significant speed ups in the development of quantum secure verifiable oblivious pseudo random functions. Basso.[25] presents a new round-optimal verifiable oblivious pseudorandom function based on SIDH and uses the countermeasures provided by M-SIDH[9], which currently hinders the performance of his proposal. I suggest by combining my fixed implementation with his optimisation suggestions for verifiable pseudorandom functions it could lead to significant speed ups in the area. An especially important point is the functions require a large transmission of data due to multiple key exchanges in the protocol, which can cause lattice-based approaches[26] to take a toll on the bandwidth. This issue could mean that an improved isogeny-based approach with a quicker run time would be an ideal solution due to the reduced key sizes offered by isogeny-based protocols.

# References

[1] L. O. Mailloux, C. D. Lewis II, C. Riggs and M. R. Grimaila, "Post-Quantum Cryptography: What Advancements in Quantum Computing Mean for IT Professionals," in IT Professional, vol. 18, no. 5, pp. 42-47, Sept.-Oct. 2016, doi: 10.1109/MITP.2016.77.

[2] V. Bhatia and K. R. Ramkumar, "An Efficient Quantum Computing technique for cracking RSA using Shor’s Algorithm," 2020 IEEE 5th International Conference on Computing Communication and Automation (ICCCA), Greater Noida, India, Oct. 2020, pp. 89-94, doi: 10.1109/ICCCA49541.2020.9250806.

[3] D. Sun, N. Zhang and F. Franchetti, "Optimization and Performance Analysis of Shor's Algorithm in Qiskit," 2023 IEEE High Performance Extreme Computing Conference (HPEC), Boston, MA, USA, 2023, pp. 1-7, doi: 10.1109/HPEC58863.2023.10363522.

[4] C. Peng, J. Chen, S. Zeadally and D. He, "Isogeny-Based Cryptography: A Promising Post-Quantum Technique," in IT Professional, vol. 21, no. 6, pp. 27-32, 1 Nov.-Dec. 2019, doi: 10.1109/MITP.2019.2943136.

[5] Jao, D., De Feo, L. “Towards Quantum-Resistant Cryptosystems from Supersingular Elliptic Curve Isogenies”. In: Yang, BY. (eds) Post-Quantum Cryptography*. PQCrypto 2011*. Lecture Notes in Computer Science, vol 7071. Springer, Berlin, Heidelberg. 2011. https://doi.org/10.1007/978-3-642-25405-5\_2.

[6] Basso, A., Fouotsa, T.B. “New SIDH Countermeasures for a More Efficient Key Exchange”. In: Guo, J., Steinfeld, R. (eds) Advances in Cryptology – ASIACRYPT 2023. *ASIACRYPT 2023*. Lecture Notes in Computer Science, vol 14445. Springer, Singapore. 2023. https://doi.org/10.1007/978-981-99-8742-9\_7.

[7] SageMath, “the Sage Mathematics Software System (Version 10.3)”, The Sage Developers, 2024, https://www.sagemath.org.

[8] Fabio Campos, Jorge Chavez-Saab, Jesús-Javier Chi-Domínguez, Michael Meyer, Krĳn Reijnders, Francisco Rodríguez-Henríquez, Peter Schwabe, and Thom Wiggers. “Optimizations and Practicality of High-Security CSIDH”. *Cryptology ePrint Archive, Paper 2023,793*. 2023. https://eprint.iacr.org/2023/793.

[9] Fouotsa, T.B., Moriya, T., Petit, C. “M-SIDH and MD-SIDH: Countering SIDH Attacks by Masking Information”. In: Hazay, C., Stam, M. (eds) Advances in Cryptology – EUROCRYPT 2023. EUROCRYPT 2023. Lecture Notes in Computer Science, vol 14008. Springer, Cham. 2023. https://doi.org/10.1007/978-3-031-30589-4\_10.

[10] Basso, A., Maino, L. and Pope, G. “FESTA: Fast Encryption from Supersingular Torsion Attacks”, in Guo, J. and Steinfeld, R. (eds) *Advances in Cryptology -- ASIACRYPT 2023*. Singapore: Springer Nature Singapore, pp. 98–126. 2023. https://doi.org/10.1007/978-981-99-8739-9\_4.

[11] Jacques Vélu. “Isogénies entre courbes elliptiques”. C. R. Acad. Sci. Paris Sér. A-B, 273:A238–A241, 1971. https://aghitza.org/publications/translation-velu.

[12] David J. et al. “SIKE – supersingular key encapsulation”. *SIKE.org*. https://sike.org (accessed 08-05-2024).

[13] NIST. “Post-Quantum Cryptography”. *NIST*. https://csrc.nist.gov/projects/post-quantum-cryptography (accessed 08-05-2024).

[14] Castryck, W., Decru, T. “An Efficient Key Recovery Attack on SIDH”. In: Hazay, C., Stam, M. (eds) Advances in Cryptology – EUROCRYPT 2023. *EUROCRYPT 2023*. Lecture Notes in Computer Science, vol 14008. Springer, Cham. 2023. https://doi.org/10.1007/978-3-031-30589-4\_15.

[15] Kani, Ernst. "The number of curves of genus two with elliptic differentials." *Journal für die reine und angewandte Mathematik*, vol. 1997, no. 485, 1997, pp. 93-122. 1997. https://doi.org/10.1515/crll.1997.485.93.

[16] D. Kohel, K. Lauter, C. Petit, and J.-P. Tignol, “On the quaternion l-isogeny path problem,” *LMS Journal of Computation and Mathematics*, vol. 17, no. A, pp. 418–432, 2014. doi:10.1112/S1461157014000151.

[17] Maino, L., Martindale, C., Panny, L., Pope, G., Wesolowski, B. “A Direct Key Recovery Attack on SIDH”. In: Hazay, C., Stam, M. (eds) Advances in Cryptology – EUROCRYPT 2023*. EUROCRYPT 2023*. Lecture Notes in Computer Science, vol 14008. Springer, Cham. 2023. https://doi.org/10.1007/978-3-031-30589-4\_16.

[18] Luciano M., Chloe M. “An attack on SIDH with an arbitrary starting curve*”. Cryptology ePrint Archive*, Paper 2022/1026. 2022. https://eprint.iacr.org/2022/1026.

[19] Robert, D. “Breaking SIDH in Polynomial Time”. In: Hazay, C., Stam, M. (eds) Advances in Cryptology – EUROCRYPT 2023. *EUROCRYPT 2023*. Lecture Notes in Computer Science, vol 14008. Springer, Cham. 2023. https://doi.org/10.1007/978-3-031-30589-4\_17.

[20] Kaizhan L. et al. “Compressed M-SIDH: An Instance of Compressed SIDH-like Schemes with Isogenies of Highly Composite Degrees”. *Cryptology ePrint Archive*, Paper 2023/136. 2023. https://eprint.iacr.org/2023/136.

[21] Castryck, W., Lange, T., Martindale, C., Panny, L., Renes, J. “CSIDH: An Efficient Post-Quantum Commutative Group Action”. In: Peyrin, T., Galbraith, S. (eds) Advances in Cryptology – ASIACRYPT 2018. *ASIACRYPT 2018*. Lecture Notes in Computer Science(), vol 11274. Springer, Cham. 2018. https://doi.org/10.1007/978-3-030-03332-3\_15.

[22] Phillip G. et al. “Swoosh: Efficient Lattice-Based Non-Interactive Key Exchange”. *Cryptology ePrint Archive*, Paper 2023/271. 2023. https://eprint.iacr.org/2023/271.

[23] Ebrahimi, E. (2022). “Post-quantum Security of Plain OAEP Transform”. In: Hanaoka, G., Shikata, J., Watanabe, Y. (eds) Public-Key Cryptography – PKC 2022. PKC 2022. *Lecture Notes in Computer Science()*, vol 13177. Springer, Cham. 2022. https://doi.org/10.1007/978-3-030-97121-2\_2.

[24] Kohei N. Hiroshi O. “QFESTA: Efficient Algorithms and Parameters for FESTA using Quaternion Algebras”. *Cryptology ePrint Archive*, Paper 2023/1468. 2023. https://eprint.iacr.org/2023/1468.

[25] Basso, A. “A Post-Quantum Round-Optimal Oblivious PRF from Isogenies”. In: Carlet, C., Mandal, K., Rijmen, V. (eds) Selected Areas in Cryptography – SAC 2023. SAC 2023. *Lecture Notes in Computer Science*, vol 14201. Springer, Cham. 2024. https://doi.org/10.1007/978-3-031-53368-6\_8.

[26] Albrecht, M.R., Davidson, A., Deo, A., Smart, N.P. “Round-Optimal Verifiable Oblivious Pseudorandom Functions from Ideal Lattices”. In: Garay, J.A. (eds) Public-Key Cryptography – PKC 2021. PKC 2021. *Lecture Notes in Computer Science(),* vol 12711. 2021. Springer, Cham. https://doi.org/10.1007/978-3-030-75248-4\_10.

[27] Boneh, D., Kogan, D., Woo, K. “Oblivious Pseudorandom Functions from Isogenies”. In: Moriai, S., Wang, H. (eds) Advances in Cryptology – ASIACRYPT 2020. ASIACRYPT 2020. *Lecture Notes in Computer Science(),* vol 12492. Springer, Cham. 2020. https://doi.org/10.1007/978-3-030-64834-3\_18.

[28] Andrea Basso. “bin\_terSIDH-SageMath”. https://github.com/binary-ternarySIDH/bin-terSIDH-SageMath (accessed 08-05-2024).

[29] Micheal M., Steffen R. “A faster way to the CSIDH”. *Cryptology ePrint Archive*, Paper 2018/782. 2018. https://eprint.iacr.org/2018/782.

[30] Daniel J. Bernstein, Luca De Feo, Antonin Leroux, Benjamin Smith. "Faster computation of isogenies of large prime degree." *Algorithmic Number Theory Symposium 2020*. 2020. https://velusqrt.isogeny.org/velusqrt-20200616.pdf.

[31] Miller, V. “The Weil Pairing, and Its Efficient Calculation”. *J Cryptology* **17**, 235–261 (2004). 2004. https://doi.org/10.1007/s00145-004-0315-8.

[32] Renes, J. “Computing Isogenies Between Montgomery Curves Using the Action of (0, 0)”. In: Lange, T., Steinwandt, R. (eds) Post-Quantum Cryptography. PQCrypto 2018. *Lecture Notes in Computer Science()*, vol 10786. Springer, Cham. 2018. https://doi.org/10.1007/978-3-319-79063-3\_11.

[33] Costello, C., Hisil, H. “A Simple and Compact Algorithm for SIDH with Arbitrary Degree Isogenies”. In: Takagi, T., Peyrin, T. (eds) Advances in Cryptology – ASIACRYPT 2017. ASIACRYPT 2017. *Lecture Notes in Computer Science()*, vol 10625. 2017. Springer, Cham. 2017. https://doi.org/10.1007/978-3-319-70697-9\_11.

[34] Giacomo Pope. “KummerIsogeny”. https://github.com/GiacomoPope/KummerIsogeny (accessed 08-05-2024).

[35] Wikipedia. “Kummer surface”. https://en.wikipedia.org/wiki/Kummer\_surface (accessed 08-05-2024).

[36] SageMath. “Elliptic curves over a general ring”. https://github.com/sagemath/sage/blob/develop/src/sage/schemes/elliptic\_curves/ell\_generic.py (accessed 08-05-2024).

[37] Costello, C., Smith, B. “Montgomery curves and their arithmetic”. *J Cryptogr Eng***8**, 227–240 (2018). 2018. https://doi.org/10.1007/s13389-017-0157-6.

[38] Wikipedia. “Trial division”. https://en.wikipedia.org/wiki/Trial\_division (accessed 08-05-2024).

[39] Ian F. Blake, Kumar Murty, Guangwu Xu. “Refinements of Miller’s algorithm for computing the Weil/Tate pairing”. *J. Algorithms 58*, 2 (February 2006), 134-149. Feb. 2006. https://dl.acm.org/doi/abs/10.5555/1140415.1712378.

[40] Oleksii Stetsyk. “Weil-pairing-through-Miller-algorithm”. https://github.com/Stetsyk/Weil-pairing-through-Miller-algorithm (accessed 08-05-2024).

[41] Microsoft. “PQCrypto-SIDH”. https://github.com/Microsoft/PQCrypto-SIDH (accessed 08-05-2024).

[42] SCIPR Lab. “libff: C++ library for Finite fields and Elliptic Curves”. https://github.com/scipr-lab/libff (accessed 08-05-2024).

[43] Saied H. Khayat. “GaloisCPP: C++ Library for General Galois Field Arithmetic”. https://github.com/saiedhk/GaloisCPP (accessed 08-05-2024).

[44] The FLINT team. “FLINT: Fast Library for Number Theory”. Version 3.0.0, 2023. https://flintlib.org.

[45] Microsoft. “vcpkg: C++ Library Manager for Windows, Linux and MacOS”. https://vcpkg.io/en/ (accessed 08-05-2024).