IT Security

Implementation of SIKE in Java

A detailed and comprehensive guide for understanding and implementing SIKE – A NIST Post-Quantum-Competition candidate

Bachelor thesis

For attainment of the academic degree of

Bachelor of Science in Engineering (BSc)

submitted by

Stefan Schubert

is161041

in the

University Course IT Security at St. Pölten University of Applied Sciences

The interior of this work has been composed in LaTeX.

Supervision

Advisor: FH-Prof. Univ.-Doz. Dipl.-Ing. Dr. Ernst Piller

Assistance:

St. Pölten, June 6, 2019 ________________ ________________

(Signature author) (Signature advisor)

Declaration

I declare that to the best of my knowledge and belief

- This thesis is my own, original work composed entirely by myself.
- I have made no use of sources, materials or assistance other than those which habe been acknowledged.
- This work has not previously been published, accepted in substance for any degree and is not being concurrently submitted in candidature for any degree.

I hereby give consent for my thesis, if accepted, to be available for photocopying and for inter-library loan, and for the title and summary to be made available to outside organisations.

Date *Signature*

Kurzfassung

Die Ära der Quantencomputer rückt mit großen Schritten näher, denn die Forschung im Bereich der Quantencomputer nimmt rasant zu. Ein Quantencomputer mit ausreichenden q-Bits wird einen großen Einfluss auf die konventionelle Kryptographie haben und möglicherweise deren Sicherheit durch die Anwendung von entsprechenden Algorithmen zunichte machen. Deshalb ist die sogenannte Post-Quantum-Kryptographie, welche ein neuer Bereich der Kryptographie ist, notwendig, um im Zeitalter der Quantencomputer weiterhin sicher kommunizierun zu können. Ein solches System existiert noch nicht und deshalb hat das NIST (National Institute of Standards and Technology) einen Call for Papers angekündigt, um Forscher zu ermutigen, ein Kryptosystem zu finden, das Angriffen von Quantencomputern standhalten kann, während es Schlüsselgrößen mit einer angemessenen Länge und guter mathematischer und rechnerischer Leistung aufweist. Einer der vielversprechendsten Kandidaten ist der SIDH (Supersingular Isogeny Diffie-Hellman) mit dem SIKE (Supersingular Isogeny Key Exchange), welche die gewünschten Eigenschaften bieten. Die Herausforderung beim SIDH/SIKE-Kryptosystem ist eine optimierte Implementierung, welche Schlüssel und Signaturen so klein wie möglich hält und gleichzeitig die isogenen Berechnungen so schnell wie möglich durchführt. Dies ist eine schwierige Aufgabe, da viele kleine IoT-Geräte und Smart Gadgets auf einer reduzierten Version von Java laufen, in der die Leistung und Kompatibilität ziemlich schlecht ist. Aus diesem Grund versucht diese Arbeit, ein schnelles und optimiertes Basis-Framework für SIDH/SIKE-Anwendungen in Java zu schaffen. Dies beinhaltet eine vollständige Erklärung des gesamten mathematischen und algorithmischen Konzepts, einschließlich einer SIDH-Implementierung in Sagemath und einer vollständigen SIDH/SIKE-Implementierung in Java. Jede wichtige Java-Funktion wird ausführlich erläutert und enthält Optimierungskonzepte, alternative Algorithmen und beschreibt die Rolle der Funktion im gesamten SIDH/SIKE-Kryptosystem.
Diese Implementierung wird dann auf ihre Rechengeschwindigkeit hin gemessen, um einen Vergleich zu ermöglichen und eine Grundlage für andere Implementierungen und zukünftige Optimierungen zu schaffen. Die gesammelten Informationen werden dann zu einem Sicherheitskonzept zusammengefasst, in dem die Bedeutung korrekter Berechnungen und der Quantensicherheit verdeutlicht wird. Desweiteren könnte dieses Basis-Framework auch in anderen isogenen Konzepten wie Authentifizierung verwendet werden.

Abstract

The era of quantum computers is approaching with great strides, because research in the field of quantum computers is increasing rapidly. A quantum computer with sufficient q-bits will have a great influence on conventional cryptography and possibly destroy its security by the application of appropriate algorithms. Therefore, the so-called post quantum cryptography, which is a new area of cryptography, is necessary to be able to communicate safely in the age of quantum computers. Such a system does not yet exist and therefore the NIST (National Institute of Standards and Technology) has announced a Call for Papers to encourage researchers to find a cryptosystem that can withstand attacks from quantum computers while providing key sizes with an appropriate length and good mathematical and computational performance. One of the most promising candidates is the SIDH (Supersingular Isogeny Diffie-Hellman) with the SIKE (Supersingular Isogeny Key Exchange), which offers the desired properties.

The challenge with the SIDH/SIKE cryptosystem is an optimized implementation that keeps keys and signatures as small as possible while performing isogenic calculations as fast as possible. This is a difficult task because many small IoT devices and smart gadgets run on a reduced version of Java where performance and compatibility is quite poor.

For this reason, this bachelor thesis tries to create a fast and optimized basic framework for SIDH/SIKE applications in Java. This includes a full explanation of the entire mathematical and algorithmic concept, including a SIDH implementation in Sagemath and a full SIDH/SIKE implementation in Java. Each important Java function is explained in detail and includes optimization concepts, alternative algorithms, and the role of the function in the entire SIDH/SIKE cryptosystem.

This implementation is then measured in terms of its computational speed to enable comparison and provide a basis for other implementations and future optimizations. The collected information is then combined into a security concept that explains the importance of correct calculations and quantum security. Furthermore, this basic framework could also be used in other isogenic concepts such as authentication.

Contents

1 Introduction . . . **1**
- 1.1 Motivation . . . 1
- 1.2 Structure . . . 3
- 1.3 Methodology . . . 5
- 1.4 Problem description and aim of the work . . . 6

2 Historical facts . . . **8**
- 2.1 Public-key cryptography . . . 8
- 2.2 Quantum-Computers . . . 9

3 Literature analysis . . . **11**
- 3.1 Post-quantum cryptography . . . 11
  - 3.1.1 Code-based post-quantum cryptography . . . 11
  - 3.1.2 Lattice-based post-quantum cryptography . . . 11
  - 3.1.3 Hash-based post-quantum cryptography . . . 12
  - 3.1.4 Multivariate-based post-quantum cryptography . . . 12
  - 3.1.5 Isogeny-based post-quantum cryptography . . . 12
- 3.2 SIDH-SIKE . . . 12

4 Mathematical Preliminaries . . . **15**
- 4.1 Modular arithmetic . . . 15
- 4.2 Algebraic structures and operations . . . 16
  - 4.2.1 Basic structure rules . . . 16
  - 4.2.2 Field . . . 19
  - 4.2.3 Ring . . . 21
  - 4.2.4 Group . . . 23

4.2.5 Morphisms 24
4.2.6 Endomorphism ring 25
4.2.7 Quadratic residue and non-residue 25
4.3 Finite fields 26
4.3.1 Finite Field $\mathbb{F}_p$ 27
4.3.2 Finite Field $\mathbb{F}_{p^2}$ 27
4.3.3 Polynomials over a field 27
4.4 Elliptic curves 28
4.4.1 Supersingular elliptic curves 29
4.4.2 Montgomery curves 29
4.4.3 Torsion 30
4.4.4 Twists 30
4.4.5 Point at Infinity 30
4.5 Mathematics of Montgomery curves 30
4.5.1 Point addition 30
4.5.2 Point doubling 31
4.5.3 Point tripling 31
4.5.4 j-invariant 31
4.5.5 Montgomery ladder 32
4.6 Isogenies 32
4.7 Isogenic calculations on elliptic curves 32
4.8 Graph theory 32

5 SIDH-Sage 35
5.1 Graphical display of the SIDH-Algorithm 36
5.2 Implementation in Sagemath 37

6 Java Huldra Library 41
6.1 Immutable vs. Mutable Objects 41
6.2 Performance 43
6.3 BigInt mathematical operation examples 44

7 SIKE-Java 46

7.1 The SIDH/SIKE implementation 46
7.2 Finite Field $\mathbb{F}_p$ class implementation 46
7.2.1 Implementation of the Tonelli-Shanks square root algorithm in $\mathbb{F}_p$ 46
7.2.2 Implementation of the quadratic residue function in the finite field $\mathbb{F}_p$ 48
7.2.3 Implementation of the quadratic non-residue function in the finite field $\mathbb{F}_p$ 49
7.2.4 Implementation of the pow function in the finite field $\mathbb{F}_p$ 49
7.2.5 Implementation of the power-of-2 amount function in the finite field $\mathbb{F}_p$ 50
7.2.6 Implementation of the binary logarithmic order function in the finite field $\mathbb{F}_p$ 50
7.2.7 Implementation of a fast multiplicative inverse function in the finite field $\mathbb{F}_p$ 51
7.2.8 Implementation of general class functions for the finite field $\mathbb{F}_p$ class 53
7.3 Finite Field $\mathbb{F}_{p^2}$ class implementation 57
7.3.1 Implementation of complex number multiplication in $\mathbb{F}_{p^2}$ 57
7.3.2 Implementation of the complex number multiplicative inverse in $\mathbb{F}_{p^2}$ 59
7.3.3 Implementation of the complex number pow function in $\mathbb{F}_{p^2}$ 59
7.3.4 Implementation of the quadratic residue function in $\mathbb{F}_{p^2}$ 60
7.3.5 Implementation of the complex number square roots $\mathbb{F}_{p^2}$ 61
7.3.6 Implementation of the multiplication by I function in $\mathbb{F}_{p^2}$ 63
7.4 Montgomery and isogeny math class implementation 68
7.4.1 Implementation of the Montgomery point doubling method 68
7.4.2 Repeated Montgomery point doubling method 70
7.4.3 Implementation of the Montgomery point addition method 70
7.4.4 Implementation of the Montgomery point tripling method 72
7.4.5 Repeated Montgomery point tripling method 72
7.4.6 Implementation of the Montgomery double-and-add method 73
7.4.7 Implementation of the j-Invariant of a Montgomery curve 74
7.4.8 Implementation of the 4-isogeny curve coefficient algorithm 76
7.4.9 Implementation of the 4-isogeny curve point algorithm 77
7.4.10 Implementation of the 3-isogeny curve coefficient algorithm 80
7.4.11 Implementation of the 3-isogeny curve point algorithm 81
7.4.12 Computation and evaluation of a 2^e-isogeny 82
7.4.13 Computation and evaluation of a 3^e-isogeny 82
7.4.14 Implementation of the x-Coordinate recovery algorithm 83

7.4.15 Implementation of the y-Coordinate recovery algorithm . . . 84
7.4.16 Implementation of the 2^e-isogeny secret key generation . . . 86
7.4.17 Implementation of the 3^e-isogeny secret key generation . . . 86
7.4.18 Implementation of the 2^e-isogeny public key generation . . . 87
7.4.19 Implementation of the 3^e-isogeny public key generation . . . 88
7.4.20 Implementation of the 2^e-isogeny key exchange . . . 89
7.4.21 Implementation of the 3^e-isogeny key exchange . . . 90
7.5 Montgomery $\mathbb{F}_{p^2}$ Point class implementation . . . 91
7.6 Public-key class implementation . . . 93
7.7 IsoReturn data-hoalding class implementation . . . 94
7.8 Main engine implementation of the SIDH/SIKE algorithm . . . 96
7.8.1 Main SIDH-Algorithm . . . 101
7.8.2 Public-Key-Encryption algorithm implementation . . . 102
7.8.3 Key-Encapuslation-Mechanism algorithm implementation . . . 104
7.9 Transfer class implementation . . . 106

8 Measurements . . . 108

9 Authentication schemes with isogenies . . . 109
9.1 Authentication scheme based on SIDH . . . 109
9.2 Authentication scheme based on CSIDH . . . 109

10 Security . . . 111

11 Conclusion and future prospects . . . 113
11.1 Conclusion . . . 113
11.2 Future prospects . . . 114

List of Figures . . . 115

List of Tables . . . 116

Glossary . . . 118

Bibliography . . . 119

1 Introduction

1.1 Motivation

Great inventions are the driving force in the digital world and especially important in the field of cryptography and theoretical computer architecture. Major innovations were achieved as early as 1970 in the field of cryptography and 1985 in theoretical computer architecture. On the one hand, there was the invention of public-key cryptography and, on the other hand, the idea of a new type of computer that uses quantum mechanics for calculations. Although both technologies are very valuable for our digital lives, their development causes them to interact with each other and thus impair their efficiency. Quantum computers are able to solve some mathematical problems that are not possible to solve for classical linear computers in polynomial time. Unfortunately, all modern public-key cryptography-systems are based on those mathematical problems. This means, given enough time in development and research, that quantum computers can break today's public-key cryptography. Since privacy must be possible and is a key factor in a modern and digitally thriving world, it is necessary to create a new kind of public key cryptography. It can't really be estimated when quantum computers will have a real and practical impact. An estimation would also not be able to make a statement, since the security of many messages already stored must also be guaranteed. Many secret data sets and communication records require long-term data security. Since the development and implementation of new cryptographic systems takes a long time, classical and well-tried methods are often still in use. One example would be DES cryptography, which is still used on many kinds of digital equipment. For this reason, there is a need to research and implement many new areas of post-quantum cryptography as quickly as possible. In this bachelor thesis, a comprehensive guide for implementing one of those post-quantum primitives is given. It focuses on SIKE, a candidate in NIST's Post-Quantum competition, that provides a key-exchange and encapsulation system. However, Supersingular-Isogeny-Elliptic-Curve primitives can also provide encryption, zero-knowledge proves and authentication schemes. Although compared to other post-quantum systems, these applications do not seem to be that practical in terms of speed and key-sizes, so this thesis sticks to the key-exchange. During the creation of this thesis, SIKE advanced to the second round of this competition. The second round decision will be held during the CRYPTO conference in August 2019. SIKE is the only competitor that can be compared to the most used key-exchange system today, the Diffie-Hellman key-exchange. Part of this work shows that there may be ways to develop an authentication scheme that has the performance, capabilities and key sizes to compete with other submissions. The research and implementation of post-quantum secure algorithms for cryptography is therefore one of the most important topics in modern cryptography and essential for security in the digital world. Social behavior has changed considerably in recent years. As more and more people reveal private information in social media and exchange sensitive messages via various digital communication channels such as WhatsApp, Telegram and similar messengers, it is becoming increasingly important to store this data securely for the future and protect it from attacks by quantum computers. The number of non-human electronic devices on the internet will increase sharply in the coming years. These include intelligent IoT systems, autonomous vehicle systems and smart gadgets. In addition, the internet will be connected to critical infrastructures such as power plants and water treatment plants, opening up new attack vectors for cybercrime. All these participants rely more than ever on the security and data protection of modern cryptographic systems. This work aims to advance the implementation of post quantum systems in the digital world and to achieve a better understanding of the different processes involved. Many of these post quantum systems have the problem, that developers lack the know-how about complex mathematical systems and their fast and secure application. The paper explains the basic problem description of SIKE, gives some historical facts and an overview of the current state of research. An important part is the detailed description of the mathematics needed to understand and implement this post quantum algorithm. This is done with the help of Sagemath, as it shows some of the functions in a simpler programming language and makes them easier to understand, even though it is a complex mathematical system. After that, the thesis provides the complete code description of the Java implementation. It shows how to use an external library to speed up calculations, save memory, and remove Java dependencies. The final part of the paper shows some speed measurements, authentication schemes, and some security implications of SIDH.

1.2 Structure

Chapter 1 (Introduction): The first chapter shows the motivation, outlines the structure of the work, shows the methodology and the way the research has started and presents the problem description and the goal of the work. The emphasis is on comprehensibility and reproducibility, as this is a newer and niche area of mathematics and information and learning materials are scarce.

Chapter 2 (Historical Facts): The second chapter deals with historical facts and describes the path of cryptography from its beginnings to today's problems in the digital world and its applications. It also shows the most important information concerning the early developments of isogenic mathematics up to the state of the art.

Chapter 3 (Literature Analysis): The literature analysis concentrates on a post-quantum system which was discussed in the first chapter. This system tries to solve some of today's problems in cryptography with new approaches. However, new challenges arise which the current research in cryptography deals with.

Chapter 4 (Mathematical Preliminaries): In this chapter, the most basic mathematical principles for calculating fields and isogenic elliptic curves are presented. These concepts are first described formally and in mathematical notation. In order to create an increased comprehensibility, there are also examples with texts to make it accessible also for non-mathematicians. The mathematics described here has been practiced and taught at university level for centuries, but only in the last century was it associated with cryptography and the digital world.

Chapter 5 (SIDH-Sage): Sagemath is a math toolkit that contains all the mathematical functions needed to represent the basic functionality of the SIDH algorithm. With Sage it is possible to create complex fields and isogenic elliptical curves in a fast way and to represent them in an understandable manner. With this tool and the basic mathematical knowledge it is easier to understand the complete implementation as well as the certain algorithms that are needed in the background.

Chapter 6 (Java Huldra Library): Programming with large numbers in Java is not really efficient with the vanilla Java libraries. Huldra is a third party library that can significantly speed up operations and make better use of memory. This chapter discusses how and why this works.

Chapter 7 (SIDH/SIKE-Java): The seventh chapter deals with the complete description of SIDH/SIKE together with the program code in Java. Each part of the code is described in detail. The function of each part of the code is explained by a description of the algorithms and the program itself. Furthermore it is shown why it is important how the code was optimized or how further improvements may be possible. Many of these optimizations are based on scientific work and ideas developed by cryptographers and programmers.

Chapter 8 (Measurements): This chapter deals with the measurement results obtained by the Java implementation. These results focus primarily on the SIDH algorithm and the rapid calculation of isogenies over a field of complex numbers.

Chapter 9 (Authentication with isogenies): In addition to key exchanges, there are other principles and cryptographic methods which are made possible by isogenic and supersingular elliptic curves. In the ninth chapter two authentication schemes are presented which are based on this idea. The first of those scheme types is based on SIDH. The second authentication scheme type is based on CSIDH and is called SeaSign.

Chapter 10 (Security): This chapter explains the basic security aspects of SIDH. Various conclusions resulting from different scientific studies are analysed, structured and processed to an accumulated result. Many of these results contain very recent scientific work, as many researchers are currently addressing the topic of post quantum cryptography and isogenic mathematics.

Chapter 11 (Conclusion and future prospects): The last chapter gives an outlook on what will happen in post-quantum cryptography in the coming years. In particular, cryptographic systems based on isogenic mathematics have great potential and there is still room for improvement in research. These insights and predictions give rise to new questions and insights that can be solved through research and cryptographic know-how.

1.3 Methodology

This thesis is a detailed and comprehensive guide, to understand the principles of SIDH and its implementation. To achieve this goal, the first chapter explains the principles of algebra and fields. Furthermore, some modern approaches of mathematics in cryptography are presented in order to better understand quantum computers and their functioning. This chapter is supported by a factual literature analysis on post-quantum cryptography and SIDH/SIKE.

After that, the mathematical preliminaries are shown. First, these are explained in a formal way in which the basic principles are presented, along with some text explanations and graphics. These principles provide a basis for ensuring that even readers with no knowledge about this field of mathematics can understand them.

The next important thing is to show the complex algorithm in the form of an analysis and description in Sagemath. Sagemath is a mathematical programming language that makes it easier to represent algorithms because it includes all the mathematical functions and provides an interface to use them. This should give the reader a more complete understanding of SIDH before showing the Java implementation.

Because Java does not efficiently calculate large numbers, an additional library is used that is able to perform these calculations with better performance. This chapter tries to explain why it is used in this implementation, compares pros and cons, and presents the differences in performance and speed.

The full implementation has been programmed in Java and is based on the mathematical operations on the NIST round 2 competition implementation. However, it uses more advanced and optimized calculation techniques to ensure that it is also efficient in Java. Implementation optimizations and possible improvements are not fully completed in the programming area and there are still plenty of opportunities to further improve performance. The code is broken down into several parts, which are described in detail if necessary and supported by well-known algorithms. The implementation of the isogenic functions can also serve as a basis or library for other cryptographic applications. In cryptography, isogenies can also be used to implement other methods such as authentication procedures or zero knowledge proofs. The last part of this chapter shows the SIKE encapsulation method, which is used to achieve an acceleration and offers a possibility for a static key exchange.

Empiric data on the speed of the code is given as a base for comparison and further improvements and mathematical as well as programmatical optimizations. From these results some further insights can be derived as to how the implementation could work on different platforms and protocols. Since post-quantum cryptography will play an important role in the future, it is important to find different interfaces and compatibilities and provide a framework for further research and development.

Then a complete overview of the SIDH/SIKE security analysis is given. The aim is to show the reader which factors in isogenic mathematics have an effect on cryptography and therefore play an important role. This allows certain conclusions to be drawn about security and allows future research and developers to improve the security of their implementations even further.

In the last part of the paper some conclusions about SIDH/SIKE are given. This also includes considerations on how post-quantum cryptography can be put into practice and how this field will develop in the future. The digital world is a flourishing system and must sustain the post-quantum era by using good cryptography and fast security techniques to outpeform quantum computers.

1.4 Problem description and aim of the work

The combination of quantum computers with sufficient power and their application in mathematical principles becomes a growing problem of asymmetric cryptography. Sufficiently large quantum computers are able to crack these cryptographic procedures in polynomial time, which of course has serious effects on the digital world and its secure communication. Key exchange and authentication are particularly at risk, as these cannot be performed so easily with symmetrical primitives. NIST uses a call for papers, in order to find a candidate for a secure cryptosystem in the post-quantum era. To provide security for the development of autonomous cars, IoT and cloud computing it is necessary to find suitable candidates that can replace the existing systems. This raises several questions: How can this be achieved? What method is feasible? What is the best use-case for different systems?
While this has been in discussion since 2003, there were a lot of ideas that tried to answer these questions, but none of them were proven to be secure and efficient up to a certain point. One of the candidates is SIDH, where the discrete logarithm problem is replaced with the problem of finding isogenies between certain elliptical curves. This provides really good compatibility with modern ECDH or ECDSA and bases its security on the same ideas, but lacks speed and key-size. Compared to other candidates the SIDH/SIKE

cryptosystem still has the shortest key-lengths. For this reason, the work shows how this system works from the ground up, as it can be a challenge to understand and apply the different methods and algorithms without prior knowledge in isogenic mathematics and everything related to it. The mathematics prove as a challenge to grasp in the beginning, so tries to ease into the mathematics with descriptions and graphical applications that try to show the functionality. A lot of background information, historical facts and state-of-the-art security analysis try to give enough base information to understand the workings of SIDH/SIKE as a whole. This also shows, that there is still room for improvements and further research. This implicates that the implementation is still in an ongoing living process to make it more efficient, adapt more principles and fix several issues that may appear in the future. This work can also be used to show the algorithm to students at universities and cryptographers, since programmatic and mathematical foundations are given. Although ongoing research is progressing at a rapid pace, this work attempts to focus on primitives that have been proven safe and are able to work in the post-quantum era. The security primitives and algorithms are subject to change and therefore it is important to look for new research and implementations.

2 Historical facts

2.1 Public-key cryptography

Cryptography is one of the most important fields in computer science and mathematics. It provides all important principles and primitives such as confidentiality, integrity, authentication and non-repudiation. These principles are indispensable for data protection and secure communication channels and provide a digitally secure world that everybody relies on. In modern cryptography there are two different types of systems, symmetric and asymmetric. Symmetric systems were already known to the ancient Greeks, but they have two major flaws, which makes today's use almost unthinkable. The first problem is that the key distribution is very complex. A modern application would require a lot of computing power and time to make this key distribution system applicable in practice. Each communication requires $\frac{n(n-1)}{2}$ keys, where n is the number of participants. So for twelve people it is necessary to calculate and distribute 66 different keys. This small example already shows the effort behind symmetrical distribution methods and the associated complexity, which makes them impractical in a real application. The second problem deals with the distribution of these keys. Transferring them digitally is not an option because interception of the key would significantly compromise and destroy the security of all data transmission. Direct key exchange is therefore not secure at all. For centuries, this has been a fundamental problem for the secure exchange of all kinds of data. However, through research and inventions in computer science, a solution to this problem was found in the 1970s. The first algorithm that solved the key exchange problem was named after its inventors, the Diffie-Hellman key exchange. This algorithm was published in 1976 [1]. The main difference to existing key exchange methods was the use of asymmetric keys. This means that the respective senders and receivers have different and unique information on the respective side. However, the transmitted information can only be read with the corresponding asymmetric key. Symmetric crypto-systems always used primitives like substitution and transposition, but rarely mathematical formulas. This changed with the invention of the Diffie-Hellman key exchange and later with the invention of the RSA cryptosystem [2]. In an asymmetric system, public information (public or shared key) and secret information (private or secret key) are available for data exchange. Those systems are based on so-called mathematical trapdoor functions. They are comparatively easy to calculate in one direction, but difficult to reverse if the private key is partially or completely missing. Asymmetric crypto-systems can also be used for other purposes than encryption and key exchange. There are different algorithms that utilize this cryptographic method in authentication schemes and integrity checks. However, there is a disadvantage in asymmetric cryptography. Since substitution and transposition are easy to calculate for computers, symmetric systems can be calculated faster than asymmetric systems, which are based on mathematical structures and algorithms. That's why the usage of hybrid systems is very popular. These systems encrypt the data with a symmetric key and distribute this symmetric key with an asymmetric system. So the overall data security relies on the security of the asymmetric key. Most modern asymmetric cryptosystems use the prime factorization or discrete logarithm problem in their calculations. If the symmetric key can be cracked, it is possible to read the original message in plain text. This would lead to major problems with authentication and integrity.

2.2 Quantum-Computers

Richard P. Feynman's first thoughts on calculating with a "quantum computer" came in 1982, when he asked himself whether a classical computer could probabilistically simulate quantum systems:

"If you take the computer to be the classical kind I've described so far, (not the quantum kind described in the last section) and there're no changes in any laws, and there's no hocus-pocus, the answer is certainly, No! This is called the hidden-variable problem: it is impossible to represent the results of quantum mechanics with a classical universal device." [3]

Which means that a quantum-computing-device can simulate quantum-systems and therefore also classical devices. Conversely, this means that a quantum-computer is universal. Feynman also showed an experiment with entangled photons and stated that nature is never classical. He said if you want to simulate it, make it quantum mechanical.

A more technical approach came in the year 1985 from David Deutsch, an Isrealic-British physicist. In his paper "Quantum Theory, the Church-Turing Principle and the Universal Quantum Computer", he thought of a Turing machine that can use the laws of quantum physics [4]. Besides inventing the idea of such Turing machine, he also showed that the Church-Turing principle, first named by Stephen Kleene in 1952 [5], is correct. Turing proofed that the "Entscheidungsproblem", also known as decision problem, was not solvable. This was stated by Hilbert and Ackerman in 1928 and released in the book "Principles of Mathematical Logic" 1938 [6]. Turing's conclusion to formulate the mathematical axioms as a model was the well known Turing machine from the year 1936 [7]. In the same year, Alonzo Church, an American mathematician, formulated his thesis on the λ-calculus [8].

As later shown, they both proved Goedels incompleteness-theorem [9] in different ways. This substantiated, that abstract mathematical axioms can be deduced into a practical mechanical model. This has the further implication that a quantum-mechanical calculating device could be practical.With the help of these findings, Deutsch has been able to create a new field in computer science. The biggest problem at that time was that nobody knew how such a device could be built or used for mathematical calculations. The first to find a solution for calculating and programming a quantum computer was Peter Shor in 1995[10]. His algorithm, later called the Shor-algorithm, showed how to solve the discrete logarithm problem and prime factorization in polynomial time. By the discovery of the algorithm and its important application, the research and the knowledge in the field of quantum computation has increased enormously in the last decades. These successes can also be seen today, such as Google's Bristlecone, which is a 72-qubit quantum processor. [11].

3 Literature analysis

3.1 Post-quantum cryptography

The term post-quantum cryptography was first mentioned by DJB (Daniel J. Bernstein) in 2003. He was the first to recognize that quantum computers endanger modern public key cryptography, and since then he has worked hard to advance research in this field, which has produced several new methods for a new category of cryptography. However, it is not yet known whether these ideas will work in the future and whether post-quantum cryptography will be able to use these principles. These new mathematical calculation principles can be divided into five categories, with different cryptographic approaches fitting into different categories. Research and challenges such as NIST's post-quantum competition will show how well these different algorithms will work in practice and in terms of efficiency and security.

3.1.1 Code-based post-quantum cryptography

The idea of code-based cryptography is based on the use of error correction codes. They are usually used to detect errors in data transmissions and, at best, to correct them. The security is based on the problem of decoding a random linear code. Even when using quantum computers, only exponential algorithms can be used since no other algorithms are known yet. The most well-known of these methods was invented in 1978 and is called McEliece. Based on this method there are already several modern implementations [12].

3.1.2 Lattice-based post-quantum cryptography

The basic problem with this post quantum cryptography system is called the "shortest vector problem". It is a problem of mathematics and computer science which is based on finding the shortest vector in a high-dimensional lattice. One example is the NTRU public key encryption system [13].

3.1.3 Hash-based post-quantum cryptography

Hash-based cryptography is very different from code- and lattice-based cryptography. Hash functions are so-called one-way functions. A hash function needs a set of characters and assign them to a shorter set with a certain size. Independent of the input length, the resulting hash has always the same length. This was first invented by Ralph Merkle in [14]. Two other well-known hash-based methods are [15] and [16]. Hash-based signatures rely on the fact that it is not possible to guess the plaintext used as input to the hash function. Therefore, each function has a different signature scheme. The main problem is that there can only be 2^n signed messages, where n is the maximum size of the Merkle tree.

3.1.4 Multivariate-based post-quantum cryptography

Multivariate post-quantum cryptosystems are based on polynomials, which are calculated over a finite field. Such systems have proven to be NP-hard or NP-complete and can therefore not be solved in polynomial time. For this reason, they are considered good candidates for post-quantum cryptography, especially for signature schemes.

3.1.5 Isogeny-based post-quantum cryptography

In classical cryptography, elliptical curves are often used for security and various other purposes. Mathematical operations such as point addition and point multiplication are used to create security in cryptographic applications. Isogenic cryptography is based on this principle, but uses many curves instead of one. An isogeny is therefore an operation between different curves. The so-called isogenies are functions that create mappings and connections between curves and these functions usually have very different properties that makes a distinction between them possible. The first implementation of a supersingular isogenic cryptosystem was done in 2011 by de Feo, Jao and Pluka [17]. This implementation was called "Supersingular Isogeny Diffie-Hellman", which is abbreviated to SIDH, and will be referenced as such in the later work. In the next chapter a more detailed description of this innovative cryptographic system will be given.

3.2 SIDH-SIKE

The first use of isogenies for a cryptographic purpose occurred in 2006 in a contribution by Rostovsev and Stolbunov[18]. The paper showed some theoretical applications and described how isogenic mathematics works in cryptography. Furthermore, a public key encryption method was presented, which used isogenies as a mathematical basis. A security analysis under consideration of some security parameters looked promising and had the potential to be a good post-quantum cryptography candidate. A more advanced version, including a complete Diffie-Hellman-scheme was released in 2010 by Stolbunov[18]. The former fastest classical algorithm was invented by Galbraith and Stolbunov[19]. This algorithm turned out to be exponential and had a worst case runtime of $O(\sqrt[4]{q})$. The downside was, that Stolbunovs Diffie-Hellman version had two major Problems. First, his cryptographic calculation scheme was very slow and it took about 230 seconds for a complete key exchange to take place. The second problem was, that the key could be calculated on quantum computers in polynomial time. This is a much more serious problem in the world of post quantum systems and was proven by Childs, Jao and Soukharev[20]. The idea of a post-quantum scheme with isogenic elliptic curves was therefore abandoned in 2010. A year later, in 2011, de Feo, Jao and Plut showed a new way to use isogens for cryptographic systems and proved this to a scientific work, which they even revised in 2014[17]. Instead of ordinary elliptic curves they used supersingular isogenic elliptic curves. Changing the algorithm to this particular curve type solved both the speed problem and the safety problem Stolbunov had with ordinary isogenic curves. The principles of the scientific article are still the standard for all systems of supersingular isogenic cryptography and are therefore used as a basis for further research. The system was renamed SIDH (Supersingular Isogeny Diffie-Hellman) and further progress was made in the following years. De Feo, Jao, and Plut released an enhanced version of their paper in 2014 [17]. It added a zero-knowledge scheme, better description of the security and a faster algorithm. Also in 2014 Jao and Soukharev showed the first signature-scheme based on isogeny cryptography [21]. A significant boost came in the year 2016 where Costello [22] released a more efficient algorithm and Galbraith proofed the security [23] and showed new identification protocols and authentication schemes [24]. In 2017, SIDH became really practical. First Costello released a paper with further compression on the keys [25] and Jalali brought up an implementation for the 64-bit ARM architecture [26]. Microsoft had already implemented SIDH as a full programmed beta library (c and asm) [27]. One of the problems was, that keys were only ephemeral but never static. A collaboration between researchers and engineers at Amazon, Florida Atlantic University, Infosec Global, Microsoft Research, Radboud University, Texas Instruments, Université de Versailles, and the University of Waterloo addressed this problem. This led to a key encapsulation method which is called SIKE and was introduced in late 2017 [28]. It was submitted to the NIST post-quantum cryptography-system contest and presented at RealWorldCrypto 2018 [29]. It could be shown, that with SIKE, the SIDH algorithm is faster than before. In May 2018, a new method of isogeny cryptography was released. It is called CSIDH and has some improvements to normal SIDH which needs to be investigated in the future [30]. In late 2018 and revised in released 2019 Galbraith and De Feo proposed a new signature scheme based on CSIDH called SeaSign[31]

4 Mathematical Preliminaries

This chapter explains basic and advanced mathematics needed to holistically understand the SIDH algorithm and isogeny-based computations. Most of this knowledge is state of the art and is taught at many universities. For this reason, there are very few quotes in this chapter. Only special calculations are referenced to the respective scientific work or book. To read more about mathematics of elliptic curves and SIDH/SIKE, the book from Silverman [32] and the paper from De Feo [33] can be consulted. In this chapter it should be possible to get a comprehensive overview of the SIDH/SIKE algorithm. Although the mathematical concept is very complex, it should be presented in a simpler and more understandable, yet correct form. This is also the reason why there are simple explanations and supporting graphics for each mathematical description.

4.1 Modular arithmetic

Modular arithmetic is based on whole numbers, and calculated the same way as a division is calculated. However, as the result, the residue is taken. Modular arithmetic is often used in cryptography, and many of the modern systems are calculated with it. It functions like a clock that starts and ends with the modulo number p. It's the base of understanding how discrete mathematics work.

Example: For the modulo number p 5 is taken and let a be an element of $\mathbb{N}$ It can be seen that the five numbers start all over from the beginning after 5 steps:

$$\begin{aligned}
a &\pmod{p}\\
1 &\pmod{5} = 1\\
2 &\pmod{5} = 2\\
3 &\pmod{5} = 3\\
4 &\pmod{5} = 4\\
5 &\pmod{5} = 0\\
6 &\pmod{5} = 1\\
7 &\pmod{5} = 2\\
8 &\pmod{5} = 3\\
9 &\pmod{5} = 4\\
10 &\pmod{5} = 0
\end{aligned}$$

4.2 Algebraic structures and operations

At first glance, algebraic structures seem very complicated and strange for people who have never heard of them at school or university. However, their use in mathematics is not difficult to understand. This usage basically includes everything that is needed for calculations in modern public-key cryptography. They also form the basis for so-called "discrete mathematics", which works closely with linear algebra and logic. Therefore these three mathematical areas are really necessary to understand the calculations used in today's cryptography. In this chapter the basic elements and principles are defined first, before more complex topics are addressed. A structure in mathematics means that several mathematical operations and rules are used in one unit. So each set, no matter what it consists of, can be regarded as a mathematical object, which behaves in a certain way if the rules are followed. This set can consist of numbers, points or even spins of ions. For this reason, a set of corresponding numbers and operations can be calculated in the same way as points on a curve if they are in the same structure.

4.2.1 Basic structure rules

There are a lot of different possible structures. This thesis will only focus on the most important ones. They are defined by a set of rules, also called the structure axioms, that are applied to them. There are two basic rules that apply to every structure. The first rule describes the presence of an identity element in every structure and second rule describes the existence of an inverse element.

Identity element

$$a \odot e = a$$

The circle stands for any operator. This could be an addition-operator or multiplication-operator.

Example: The identity element e in the set of $\mathbb{N}$ is calculated with the addition operator:

$$\begin{aligned} a + e &= a \\ e &= a - a \\ e &= 0 \end{aligned}$$

This proves, that the identity element in the natural numbers is zero. If we take any given number, like 34 and add 0 to it, it stays at 34. The multiplicative rule works the same way.The next example shows how to find the identity element e in $\mathbb{R}$ with the multiplication as an operator:

$$\begin{aligned} a \cdot e &= a \\ e &= \frac{a}{a} \\ e &= 1 \end{aligned}$$

In this case, the number 1 is the identity element. On later parts, it is shown, that it is always possible to find an identity element for the structures used in this papers. They might look different, but the definition of rules stays the same.

Inverse element

The second really important rule is, that there is the possibility to find a inverse element for a certain value. This means the opposite in laymans terms. In mathematics it is just defined as the element that gives the identity element to a respective element:

$$a \odot a^{-1} = e$$

Example: In this example the inverse element a^{-1} in the set of $\mathbb{N}$ is calculated with the addition operator, the identity element with addition stays at zero:

$$\begin{aligned} a + a^{-1} &= e \\ a^{-1} &= e - a \\ a^{-1} &= -a \end{aligned}$$

This shows, that the inverse of any number in $\mathbb{N}$ should be a negative number. In this case there is no negative number in $\mathbb{N}$. That shows, that there is no inverse with the addition-operator. So the inverse element rule does not work in $\mathbb{N}$. In the next example, the set of whole numbers $\mathbb{Z}$ is used.

$$\begin{aligned} a + a^{-1} &= e \\ a^{-1} &= e - a \\ a^{-1} &= -a \end{aligned}$$

It is the same calculation as in $\mathbb{N}$. With the difference, that there are negative numbers in $\mathbb{Z}$. So for any number in $\mathbb{Z}$, it is possible to find an inverse. For any number in n $\mathbb{Z}$ this inverse is the same negated number:

$$\begin{aligned} 7 + 7^{-1} &= 0 \\ 7^{-1} &= 0 - 7 \\ 7^{-1} &= -7 \end{aligned}$$

For the subsequent definition of structures, there are additional rules that only apply to the respective structures. If this is not the case, this structure cannot be referred to as a group or field. The first thing on this chapter is the fields. They have most of the rules that are applied to them.

4.2.2 Field

A field in mathematics is defined as a set to which a conglomerate of rules are applied at the sime time. This means that for example addition, subtraction, multiplication and division are defined and usable. An example of this are the real numbers $\mathbb{R}$. The classic rules for a set are:

- Associativity (addition and multiplication): $a + (b + c) = (a + b) + c$ and $a \cdot (b \cdot c) = (a \cdot b) \cdot c$
- Commutativity (addition and multiplication): $a + b = b + a$ and $a \cdot b = b \cdot a$
- Distributivity(multiplication over addition): $a \cdot (b + c) = (a \cdot b) + (a \cdot c)$
- Identity element (addition and multiplication)
- Inverse element (addition and multiplication)
- Closure(addition and multiplication): For all a, b in the field also $a + b$ and $a \cdot b$ have to be defined

Example: This example shows the prove, that elements in $\mathbb{R}$ follow all the rules above:

Associativity(addition):

$$5 + (3 + 6) = (5 + 3) + 6$$
$$5 + 9 = 8 + 6$$
$$14 = 14$$

Associativity(multiplication):

$$5 \cdot (3 \cdot 6) = (5 \cdot 3) \cdot 6$$
$$5 \cdot 18 = 15 \cdot 6$$
$$90 = 90$$

Commutativity(addition):

$$5 + 3 = 3 + 5$$
$$8 = 8$$

Commutativity(multiplication)

$$5 * 3 = 3 * 5$$
$$15 = 15$$

Distributivity(multiplication over addition):

$$5 \cdot (3+6) = (5 \cdot 3) + (5 \cdot 6)$$
$$5 \cdot 9 = 15 + 30$$
$$45 = 45$$

Identity element(addition):

$$e = 0$$
$$5 + 0 = 5$$

Identity element(addition):

$$e = 1$$
$$5 * 1 = 5$$

Inverse element(addition):

$$5 + (-5) = 0$$

Inverse element(multiplication):

$$5 * \frac{1}{5} = 1$$

Closure(addition):

$$5 + 3 = 8$$

8 is an element in $\mathbb{R}$

Closure(multiplication):

$$5 \cdot 3 = 15$$

15 is an element in $\mathbb{R}$

4.2.3 Ring

A ring is another type of algebraic structure. It is also an abelian group with the difference, that it has only two operators applied to it. Some rules for fields don't apply to a ring and there are many different types of rings. In this section, only basic rings are mentioned:

- Associativity (addition and multiplication): $a + (b + c) = (a + b) + c$ and $a \cdot (b \cdot c) = (a \cdot b) \cdot c$
- Distributivity(multiplication over addition): $a \cdot (b + c) = (a \cdot b) + (a \cdot c)$
- Identity element (addition and multiplication)
- Inverse element (addition and multiplication)
- Closure(addition and multiplication): For all a, b in the ring also $a + b$ and $a \cdot b$ have to be defined

Example: This example shows the prove, that elements in $\mathbb{Z}$ follow all the rules above:

Associativity(addition):

$$5 + (3 + 6) = (5 + 3) + 6$$
$$5 + 9 = 8 + 6$$
$$14 = 14$$

Associativity(multiplication):

$$5 \cdot (3 \cdot 6) = (5 \cdot 3) \cdot 6$$
$$5 \cdot 18 = 15 \cdot 6$$
$$90 = 90$$

Distributivity(multiplication over addition):

$$5 \cdot (3 + 6) = (5 \cdot 3) + (5 \cdot 6)$$
$$5 \cdot 9 = 15 + 30$$
$$45 = 45$$

Identity element(addition):

$$e = 0$$
$$5 + 0 = 5$$

Identity element(addition):

$$e = 1$$

$$5 * 1 = 5$$

Inverse element(addition):

$$5 + (-5) = 0$$

Inverse element(multiplication):

$$5 * \frac{1}{5} = 1$$

Closure(addition):

$$5 + 3 = 8$$

8 is an element in $\mathbb{R}$

Closure(multiplication):

$$5 \cdot 3 = 15$$

15 is an element in $\mathbb{R}$

4.2.4 Group

A group is also a set of elements with the difference, that only one rule applies. Groups are less restrictive, so forming a group is easier than forming a field. However, we can perform fewer operations on it, which means that a group is a more restrictive form of a ring and/or field in terms of usage. It can only be used with a single operation. Groups that always hold commutativity of addition are called abelian groups. The definition of a basic group is the following:

- Associativity (addition): $a + (b + c) = (a + b) + c$
- Identity element (addition)
- Inverse element (addition)
- Closure(addition and multiplication): For all a, b in the group also $a + b$ has to be defined

Example: Here we prove that elements in $\mathbb{Z}$ follow all the rules above:

Associativity(addition):

$$5 + (3 + 6) = (5 + 3) + 6$$
$$5 + 9 = 8 + 6$$
$$14 = 14$$

Identity element(addition):

$$e = 0$$
$$5 + 0 = 5$$

Inverse element(addition):

$$5 + (-5) = 0$$

Closure:

$$5 + 3 = 8$$

8 is an element in $\mathbb{Z}$

In $\mathbb{Z}$ there is no inverse element for multiplication as seen in the following example:

Inverse element(multiplication):

$$5 \cdot b = 1$$
$$5 = \frac{1}{b}$$

Since we know that $\frac{1}{b}$ is not an element in $\mathbb{Z}$ there is no multiplicative inverse. So $\mathbb{Z}$ in combination with the addition-operator forms a group.

4.2.5 Morphisms

A morphism in mathematics is a rational map, that that creates a function from one algebraic structure to another. This is called a walk. It can be seen as some kind of roads or edges, that connect several structures in some way. They can be described as a function from source to target. In mathematical notion this can be written the following way:

$$f : X \mapsto Y$$

Morphisms can also exist with several rules applied to them:

- Identity: For every X there is a morphism $id_x : X \mapsto X$, the so-called identity morphism, which is the opposite map.
- Associative: $a \odot (b \odot c) = (a \odot b) \odot c$

Example: For a source X, f makes a connection to Y. Y has also a connection to Z, called g. This makes it possible to go directly from X to Z by calculating $g \odot f$. The following figure shows this morphism.

Figure 4.1: Morphism from $X \mapsto Z$

$$\begin{tikzcd}
X \arrow[r, "f"] \arrow[dr, "f\odot g"'] & Y \arrow[d, "g"] \\
& Z
\end{tikzcd}$$

4.2.6 Endomorphism ring

In mathematics, an endomorphism is a morphism to itself. This morphism can be written as $End(X)$ and serves an important use for special curves in SIKE. The endomorphism of an abelian group forms the endomorphism ring with a multiplicative identity.

4.2.7 Quadratic residue and non-residue

Quadratic residues are so-called congruence classes which are formed by calculating the modular operation $a \not\equiv 0 \pmod{p}$ for n numbers. Certain numbers, will yield the same quadratic residue, in which case they are in the same congruence class. A value of 0 means that the number a is not a quadratic residue and therefore called quadratic non-residue. Put simply, this means that when a modulo operation is performed on a square number, all numbers that remain are a quadratic-residue, and all other numbers that are not generated by it are non-residues.
This can be mathematically defined with the Legendre-symbol:

$$\left(\frac{a}{p}\right) \equiv a^{\frac{p-1}{2}} \pmod{p}$$

$$\left(\frac{a}{p}\right) = \begin{cases} 1, & \text{if } a \text{ is a quadratic residue and } a \not\equiv 0 \pmod{p}. \\ 0, & \text{if } a \equiv 0 \pmod{p}. \\ -1, & \text{if } a \text{ is a quadratic non-residue.} \end{cases}$$

Example: The legendre notation can also be explained in a normal mathematical notation:

For the following example, the finite field $\mathbb{F}_5$ is used:

$$1^2 \pmod 5 = 1$$
$$2^2 \pmod 5 = 4$$
$$3^2 \pmod 5 = 4$$
$$4^2 \pmod 5 = 1$$
$$5^2 \pmod 5 = 0$$
$$6^2 \pmod 5 = 1$$
$$7^2 \pmod 5 = 4$$
$$8^2 \pmod 5 = 4$$
$$9^2 \pmod 5 = 1$$

The example shows that the square residues are 1 and 4 and the non-residues are 2 and 3.

4.3 Finite fields

The Galois fields or more commonly named finite fields, are mathematical field structures that contain a fixed amount of elements.[34][35]

In number theory, group theory, cryptography and general coding and computing, a finite field is mostly used as a field of integer numbers modulo a prime number. Although the properties allow the usage of arbitrary numbers like floating point values or complex numbers.[36]

The main properties of finite fields are[37][35]:

- Every finite fields order is a prime power.
- Every prime number has a finite field with the same number of elements.
- Finite fields with the same number of elements are isomorphic.
- The operations of addition and multiplication are defined over the set.
- In every field exists a multiplicative identity element and an additive identity element.
- Every element has an additive inverse and every element that is not zero has a multiplicative inverse.

4.3.1 Finite Field $\mathbb{F}_p$

The finite field $\mathbb{F}_p$ is used for some underlying calculations in the SIDH-Algorithm and certain isogenic computations. The mathematical basics are defined by the following key aspects:

- $\mathbb{F}_p = \mathbb{Z}_p = \{0, 1, ..., p-1\}$
- General: $a, b \in \mathbb{F}_p$
- Addition: $a + b \equiv r \mod p$ where $r \in [0, p-1]$
- Multiplication: $a * b \equiv r \mod p$ where $r \in [0, p-1]$
- Additive Inverse: $a + x \equiv 0 \mod p$ where $x = -a$
- Multiplicative Inverse: $ax \equiv 1 \mod p$ where $x = a^{-1}$

4.3.2 Finite Field $\mathbb{F}_{p^2}$

The finite field $\mathbb{F}_{p^2}$ is an extension of the finite field $\mathbb{F}_p$. The extension is achieved by utilising a primitive polynomial of degree n.[36]

4.3.3 Polynomials over a field

Polynomials are equations that have the following form:

$$f(x) = a_0 + a_1 x + ... + a_n x^n$$

When taking a field $(F, +, \cdot)$, the coefficients $a_0, a_1...$ have to be in this field. If a finite field is used for this, the solutions for the field polynomial are also finite. Some of these polynomials can be irreducible polynomials, which means that they cannot be incorporated into a product of two non-constant polynomials. Primitive polynomials are always irreducible and must consist of a constant nonzero term. They act like primitive roots and can be used as a generator called a "generator polynomial".
For the next example, the finite field $\mathbb{F}_3$ is used, so the possible coefficients are 1,2 and 3. The polynomial could be $f(x) = a_0 + a_1 + a_2$ and the solutions to it are:

$$f(x_1) = 1_0 + 1_1 + 1_3$$

$$f(x_2) = 1_0 + 2_1 + 1_3$$

$$f(x_3) = 1_0 + 2_1 + 2_3$$

$$f(x_4) = 2_0 + 2_1 + 3_3$$

$$...$$

The function above showcases our natural numbers that we use for calculation on a daily basis. The number 489 can be written as $4 \cdot 10^3 + 8 \cdot 10^2 + 9 \cdot 10^1$ in polynomial form.

4.4 Elliptic curves

The aforementioned examples, were all calculated in modular arithmetic. But there is also another group which is important for cryptography. It is based on points of an elliptic curve over a finite field. In the next example, a special polynomial is given over a finite field:

$$y^2 = x^3 + Ax + B \pmod{p}$$

The solution to this polynomial returns points in an n-dimensional space, depending on what kind of field is used for the coefficients. One important rule is, that $A^3 + 27B^2 \neq 0 \pmod{p}$. Following this rule is important to ensure, that $x^3 + Ax + B = 0 \pmod{p}$ has no repeated roots. In $\mathbb{F}_p$ there are only natural numbers, so the space is 2-dimensional. In the case of $\mathbb{F}_{p^2}$, complex numbers are used and the projection space is 4-dimensional. To form an algebraic structure for calculations, the axioms have to be proven. In this case, the identity-element is a point at infinity, written as $\mathcal{O}$ and the inverse of it is the opposite point on the other end of the field. It is proven, that addition and multiplication with points on elliptic curves are also feasible. So points on elliptic curves over a finite field can be used in the same way as normal integer addition and multiplication. That's why it is possible to use cryptographic protocols like the Diffie-Hellman key-exchange with elliptic curves in the same way as it is possible with normal numbers.

Example of an elliptic curve: For the standard elliptic curve polynomial, the finite field $\mathbb{F}_{13}$, with the coefficients $a = 1$ and $b = 0$ is used:

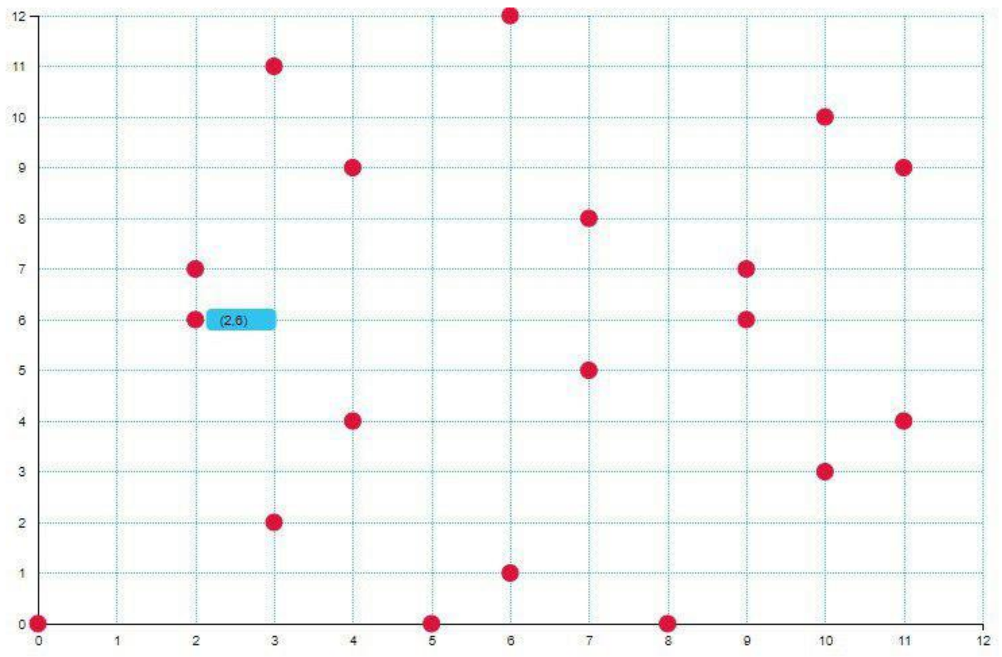


Figure 4.2: Points on elliptic curve over $\mathbb{F}_{13}$[38]

4.4.1 Supersingular elliptic curves

Elliptic curves are divided into two basic types:

- Ordinary elliptic curve used for ECDH or ECDSA
- Supersingular elliptic curve used in SIDH/SIKE

The main difference between them is, that a supersingular curve forms a really large endomorphism ring. The term "supersingular" is not related to the singularity of points or that the curve itself is somehow unique/singular. It is called supersingular, because the j-invariant of the curve has a singular value, which is important for the SIDH/SIKE algorithm in order to calculate the shared/secret key.

4.4.2 Montgomery curves

The Montgomery curve is a form of an elliptic curve, that differs from the normal Weierstrass form. However, every elliptic curve could be written in Weierstrass or Montgomery form. The basic notation is:

$$By^2 = x^3 + Ax^2 + x$$

The notation above is often used in cryptography, because there is a remarkably speed up in calculation time and performance in certain programs. The reason for this is, that there are also different mathematical notations for mathematical operations on Montgomery curves.

4.4.3 Torsion

A Torsion group includes all elements g in a group and a natural number n that satisfies the equation $g^n = 0$ or $g \cdot n = 1$. Torsion points or division points are points of algebraic variety, which are computed as division polynomials. In SIDH/SIKE the torsion groups are building the kernel of the isogenies.

4.4.4 Twists

In algebraic geometry, any elliptic curve over a field has a quadratic twist. This could be another curve, which is is isomorphic to the first one. An isomorphism between elliptic curves with degree 1, is called an isogeny that is invertible. There can be twists of higher order and they both have the same j-invariant. (more on isogenies->verweis)

4.4.5 Point at Infinity

The point at infinity($\mathcal{O}$) is a point that is used to get the projective completion. In an elliptic curve it is the identity element. So a point and its inverse point add up to the point at infinity.

4.5 Mathematics of Montgomery curves

4.5.1 Point addition

Given two points $P(x_P, y_P)$, $Q(x_Q, y_Q)$ where $P \neq \pm Q$ the point addition on a Montgomery curve $E_{A,B}$ and $A, B \in \mathbb{F}_p$ is done by computing λ, x_R and y_R. The point addition is also important for calculating the order of a point which is known as $ord(P)$ in mathematical terms.[39]

Equation for λ which is needed in the addition:

$$\lambda = \frac{(y_P - y_Q)}{(x_P - x_Q)}$$

With λ the new Point $R(x_R, y_R) = P + Q$ can be calculated with the following equations:

$$x_R = B\lambda^2 - (x_P + x_Q) - A$$

$$y_R = \lambda(x_P - x_R) - y_P$$

The order of a point can be calculated by adding a point P to itself for n times. The order is the smallest amount that a point can be added to itself before reaching the point at infinity($\mathcal{O}$). The order is important to differentiate all curve points into different classes. This can be used to optimize certain computations and filter out points that are unable to work with the SIDH algorithm.

$$\underbrace{P + P + ... + P}\_n = [n]P \text{ until } [n]P = \mathcal{O}$$

4.5.2 Point doubling

The second primitive point operation is the point doubling. It takes advantage of a point whose order does not divide 2. It is essentially the same as adding a point to itself with the main difference, that it only needs the values of x_P and A in the calculation. This leaves room for speed and performance optimizations compared to the ordinary point addition. Furthermore, the performance is even better when computing in the finite field $\mathbb{F}_{p^2}$ which is taking advantage of fewer variables and less memory allocations while calculating with complex numbers.[39]

For $P = (x_P, y_P) \in (E_{A,B} \wedge ord(P) \mod 2 \neq 0)$ the doubled point $[2]P = (x_{[2]P}, y_{[2]P})$ can be calculated with:

$$(x_{[2]P}, y_{[2]P}) = \left(\frac{(x_P^2 - 1)^2}{4x_P(x_P^2 + Ax_P + 1)}, y_P * \frac{(x_P^2 - 1)(x_P^4 + 2Ax_P^3 + 6x_P^2 + 2x_P + 1)}{8x_P^2(x_P^2 + Ax_P + 1)^2} \right)$$

4.5.3 Point tripling

The third primitive point operation is the point tripling method. It takes advantage of a point whose order does not divide 3. For $P = (x_P, y_P) \in (E_{A,B} \wedge ord(P) \mod 3 \neq 0)$ the doubled point $[3]P = (x_{[3]P}, y_{[3]P})$ can be calculated with[39]:

$$(x_{[3]P}) = \frac{(x_P^4 - 4Ax_P - 6x_P^2 - 3)^2 x_P}{(4x_P^3 + 3x_P^4 + 6x_P^2 - 1)^2}$$

$$(y_{[3]P}) = \frac{(x_P^4 - 4Ax_P - 6x_P^2 - 3)(x_P^8 + 4Ax_P^7 + 28x_P^6 + 28Ax_P^5 + (16A^2 6)x_P^4 + 28Ax_P^3 + 28x_P^2 + 4Ax_P + 1)}{(4Ax_P^3 + 3x_P^4 + 6x_P^2 - 1)^3}$$

4.5.4 j-invariant

The j-function plays an important role in the mathematics of supersingular elliptic curves and their functions. It is a basic module function, which is used for many applications in SIKE and comparable cryptographic applications. The j-invariant produces a certain value that describes the uniqueness of a curve up to isomorphism. This value is calculated by taking A from $(E_{A,B}$ and executing the following function[39]:

$$j(\mathcal{E}_{A,B}) = \frac{256(A^2 - 3)^3}{A^2 - 4}$$

4.5.5 Montgomery ladder

The Montgomery ladder has been proposed by Peter L. Montgomery in the year 1987 [40]. This ladder tries to compute point multiplications in a fixed amount of time. It can basically removes side-channel attacks from any algorithm, because it calculates the same number of point additions, independent of the value of the multiplicand d.

4.6 Isogenies

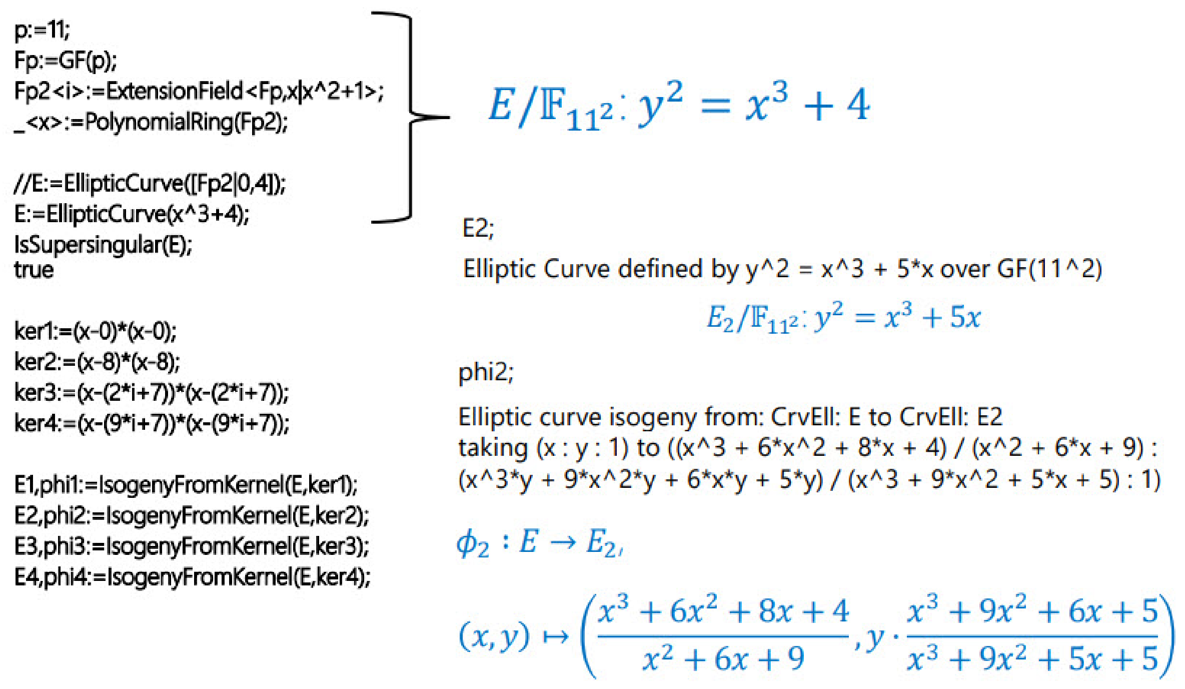
In general algebraic geometry an isogeny is a rational map between two abelian varieties, where the degree of an isogeny, is the number of elements in its kernel. It is a morphism, more precisely, an isomorphism, that connects an elliptic curve to another curve, and also preserves the identity. They are called isomorphic if the j-invariant of the curves are the same. Endomorphism and Isomorphism are just special cases of isogenies. The degree $\ell + 1$ specifies how many curves are connected to a base curve. So an isogeny with degree 3 has 4 other curves connected to it. This is one of the most important features of SIDH/SIKE, because safety is based on the difficulty of calculating these isogenies. This means that the "path" is known when the torsion points are not known. The kernel points of the torsion subgroup has a degree, which is an endomorphism. So a degree 3 torsion subgroup in its respective kernel means, that by adding itself together three times leads back to itself.

4.7 Isogenic calculations on elliptic curves

The following picture shows a complete isogeny calculation over an elliptic curve. With complex numbers in $\mathbb{F}_{11^2}$, there are 4 torsion subgroups that build the kernel. From there on the isogeny function is calculated and generates the map to go from E to $E2$

4.8 Graph theory

Graphs are basic objects in mathematics. They consist of nodes or vertices and edges or connections. They can display a path from one node via an edge to another node. There are directional and non-directional graphs. These are very important for discrete mathematics and computer science. Examples are binary trees or the map of hyperlinks in a website. Graphs are also directly linked to algebra and for this reason they can also be used together with algebraic structures. An example of this is the SIDH/SIKE, which uses elliptical curves as edges and represents the isogeny between the curves using graphs. Each curve has $\ell + 1$



```
p:=11;
Fp:=GF(p);
Fp2<i>:=ExtensionField<Fp,x|x^2+1>;
_<x>:=PolynomialRing(Fp2);

//E:=EllipticCurve([Fp2|0,4]);
E:=EllipticCurve(x^3+4);
IsSupersingular(E);
true

ker1:=(x-0)*(x-0);
ker2:=(x-8)*(x-8);
ker3:=(x-(2*i+7))*(x-(2*i+7));
ker4:=(x-(9*i+7))*(x-(9*i+7));

E1,phi1:=IsogenyFromKernel(E,ker1);
E2,phi2:=IsogenyFromKernel(E,ker2);
E3,phi3:=IsogenyFromKernel(E,ker3);
E4,phi4:=IsogenyFromKernel(E,ker4);
```

$E/\mathbb{F}_{11^2}: y^2 = x^3 + 4$

```
E2;
Elliptic Curve defined by y^2 = x^3 + 5*x over GF(11^2)
```

$E_2/\mathbb{F}_{11^2}: y^2 = x^3 + 5x$

```
phi2;
Elliptic curve isogeny from: CrvEll: E to CrvEll: E2
taking (x : y : 1) to ((x^3 + 6*x^2 + 8*x + 4) / (x^2 + 6*x + 9) :
(x^3*y + 9*x^2*y + 6*x*y + 5*y) / (x^3 + 9*x^2 + 5*x + 5) : 1)
```

$\phi_2 : E \rightarrow E_2,$

$$(x, y) \mapsto \left(\frac{x^3 + 6x^2 + 8x + 4}{x^2 + 6x + 9}, y \cdot \frac{x^3 + 9x^2 + 6x + 5}{x^3 + 9x^2 + 5x + 5}\right)$$

Figure 4.3: Isogenic calculation over $\mathbb{F}_{11^2}$[41]

edges or links to other curves in the field. The first use in computer science for isogeny calculations was a hash function. This means that you want to avoid collisions with graphs because it is really unlikely to take completely identical paths in a graph[42]. 4.4 shows an isogenic diagram with degree 2 and 4.5 shows an isogenic diagram with degree 3.[41]

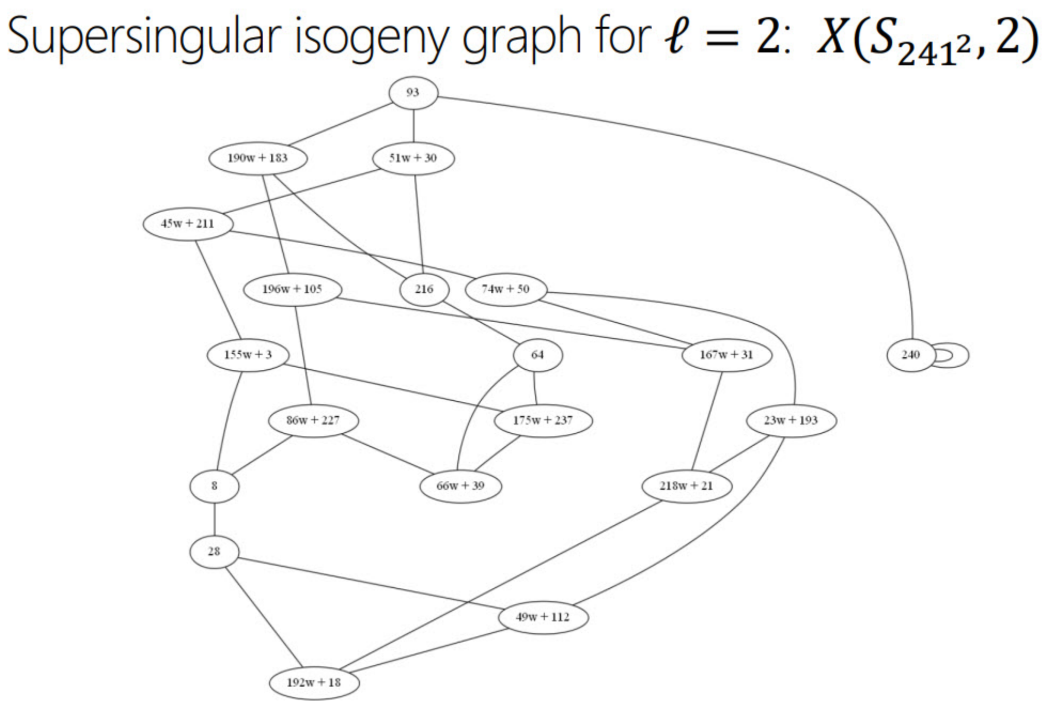


Figure 4.4: Isogenic graph with degree 2[41]

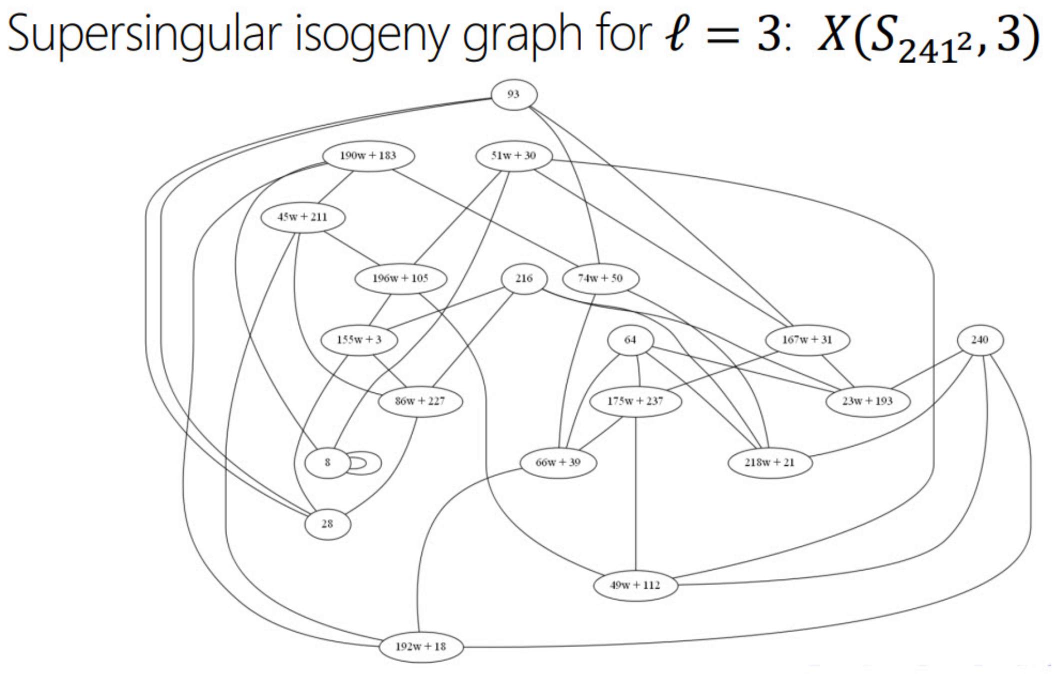


Figure 4.5: Isogenic graph with degree 3[41]

5 SIDH-Sage

Sagemath is a mathematical calculation tool that provides a mathematical programming language with an interface. The main advantages are that this tool provides the ability to implement, analyze and learn complex algorithms without having to worry about the underlying instructions and routines. This makes it much easier to create programming templates and troubleshoot algorithms. The main advantage that Sagemath offers for this work is that the isogenic computations and the complex finite field mathematics are fully implemented and ready to use. This is a significant help in understanding this complex topic. For this reason, the following chapter explains the complete SIDH algorithm in detail and provides background information and explanations on certain algorithms.
The main mathematical concepts, that Sage provides for this thesis are:

- Elliptic curves
- Finite fields and prime number computations
- Supersingular point operations
- Isogenies and j-invariants
- Codomains
- Kernels
- Random number and random point functions
- Graphs

5.1 Graphical display of the SIDH-Algorithm

The following graphic is the complete SIDH-Algorithm in a mathematical notation. Every step, starting from the finite field generation to the comparison of the shared secret is explained in code and described in detail. This provides a good basis to further understand the Sage code and the full implementation in Java.

$$\begin{tikzcd}
Alice \arrow[d] & \begin{array}{c} E/\mathbb{F}_p \\ P_A, Q_A, P_B, Q_B \in E \end{array} \arrow[l] \arrow[r] & Bob \arrow[d] \\
\begin{array}{c} m_A, n_A \in \mathbb{Z} \\ R = m_A * P_A + n_A * Q_A \\ \varphi : E \xrightarrow{R} E_A \\ E_A, \varphi(P_B), \varphi(Q_B) \end{array} \arrow[drr] & & \begin{array}{c} m_B, n_B \in \mathbb{Z} \\ R = m_B * P_B + n_B * Q_B \\ \varphi : E \xrightarrow{R} E_B \\ E_B, \varphi(P_A), \varphi(Q_A) \end{array} \arrow[dll] \\
E_B, \varphi(P_A), \varphi(Q_A) \arrow[d] & & E_A, \varphi(P_B), \varphi(Q_B) \arrow[d] \\
\begin{array}{c} R_{BA} = m_A * \varphi(P_A) + n_A * \varphi(Q_A) \\ \varphi : E_B \xrightarrow{R_{BA}} E_{BA} \end{array} \arrow[dr] & & \begin{array}{c} R_{AB} = m_B * \varphi(P_B) + n_B * \varphi(Q_B) \\ \varphi : E_A \xrightarrow{R_{AB}} E_{AB} \end{array} \arrow[dl] \\
& j(E_{BA}) = j(E_{AB}) &
\end{tikzcd}$$

5.2 Implementation in Sagemath

Listing 5.1: SIDH parameter setup[43]

```
sage: nA, nB = 2, 3
sage: eA, eB = 3, 4
sage: p = nA ^ eA * nB ^ eB -1
sage: p.is_prime()
True
sage: p % 4 == 3
True
sage: f = GF(p)
sage: f
Finite Field of size 647
sage: E = EllipticCurve(f, [1,0])
sage: E
Elliptic Curve defined by y^2 = x^3 + x over Finite Field of size 647
sage: E.is_supersingular()
True
sage: E.count_points()
648
```

Listing 5.1 shows the parameter setup that needs to be done in order to calculate the supersingular elliptic curve isogeny Diffie-Hellman. A prime number according to the following criteria needs to be chosen:

- $p = {n_A}^{e_A} * {n_B}^{e_B} - 1$
- $p \mod 4 = 3$

When the prime number is set up correctly, a galois field can be constructed. This is used to create a supersingular elliptic curve over a finite field. With the sage-functions is_supersingular() and count_points() the supersingularity and the amount of curve points on the elliptic curve can be checked.

Listing 5.2: Random public points generation

```
sage: points = []
sage: while len(points) != 4:
....:     p = E.random_point()
....:     if p not in points:
....:         points.append(p)
....:
sage: pA, pB, qA, qB = points
sage: pA, pB, qA, qB
((72 : 646 : 1), (66 : 335 : 1), (314 : 364 : 1), (474 : 165 : 1))
```

In listing 5.2 four public points are chosen. Each party needs two points for the later calculations. In SageMath this can be done with a loop which iterates all curve points and chooses four random elements through this procedure.

Listing 5.3: Alice isogeny generation[44][45]

```
sage: mA, nA = 104, 225
sage: rA = mA * pA + nA * qA
sage: rA
(507 : 340 : 1)
sage: phiA = E.isogeny(rA)
sage: phiA
Isogeny of degree 81 from Elliptic Curve defined by y^2 = x^3 + x over
   Finite Field of size 647 to Elliptic Curve defined by y^2 = x^3 + 81*x
   + 247 over Finite Field of size 647
sage: E_A = phiA.codomain()
sage: E_A
Elliptic Curve defined by y^2 = x^3 + 81*x + 247 over Finite Field of size
    647
sage: E.is_isogenous(E_A)
True
sage: phiA_pB, phiA_qB = phiA(pB), phiA(qB)
sage: phiA_pB
(567 : 590 : 1)
sage: phiA_qB
(99 : 212 : 1)
```

Alice now chooses two random integer numbers that are then used to calculate a new random point with point arithmetics as shown in listing 5.3. The new point R_A does not lie on the curve E which was used to sample the other four public points. This is where the so-called isogeny needs to be calculated.[46] An isogeny is a map between two curves that maps a point on one curve to a point on another curve. In this example it is done via the SageMath function isogeny(). The result of this function is an isogeny of a certain degree that maps E to a new curve E_A. The curve E_A is defined by the codomain of the isogeny.[33] By using the isogeny, Alice calculates the isogenous points P_B and Q_B, which are Bob's public points, into E_A. These points are then called φ_{P_B} and φ_{Q_B}. With the function is_isogenous() the isogenous characteristic of two corresponding curves can be checked. $E_A, \varphi_{P_B}, \varphi_{Q_B}$ are then sent over to Bob for further SIDH calculations.

Listing 5.4: Bob isogeny generation[44][45]

```
sage: mB, nB = 112, 325
sage: rB = mB * pB + nB * qB
sage: rB
(642 : 556 : 1)
sage: phiB = E.isogeny(rB)
sage: phiB
Isogeny of degree 108 from Elliptic Curve defined by y^2 = x^3 + x over
    Finite Field of size 647 to Elliptic Curve defined by y^2 = x^3 + 309*x
     + 597 over Finite Field of size 647
sage: E_B = phiB.codomain()
sage: E_B
Elliptic Curve defined by y^2 = x^3 + 309*x + 597 over Finite Field of
    size 647
sage: E.is_isogenous(E_B)
True
sage: phiB_pA, phiB_qA = phiB(pA), phiB(qA)
sage: phiB_pA, phiB_qA
((124 : 72 : 1), (0 : 1 : 0))
```

Bob's isogeny calculation in listing 5.4 is the same as the isogeny calculation done by Alice in listing 5.3. The only difference is that Bob chooses his own random integer values for the point calculation. Afterwards he sends C to Alice.[33]

Listing 5.5: Alice secret point calculation[44][43]

```
sage: secretB_A = mA * phiB_pA + nA * phiB_qA
sage: secretB_A
(124 : 575 : 1)
sage: phiB_A = E_B.isogeny(secretB_A)
sage: phiB_A
Isogeny of degree 3 from Elliptic Curve defined by y^2 = x^3 + 309*x + 597
    over Finite Field of size 647 to Elliptic Curve defined by y^2 = x^3 +
    485*x over Finite Field of size 647
sage: secretA = phiB_A.codomain().j_invariant()
sage: secretA
434
```

With the just exchanged values $E_A, \varphi_{P_B}, \varphi_{Q_B}$ Alice now performs another round of isogeny calculations. In listing 5.5 Alice creates a new secret point with her private integer values and the two points that she has just received from Bob. The new point is then used to compute an isogeny from the elliptic curve E_B. The j-invariant of the isogeny codomain is the shared secret of both parties.

Listing 5.6: Bob secret point calculation[44][43]

```
sage: secretA_B = mB * phiA_pB + nB * phiA_qB
sage: secretA_B
(99 : 212 : 1)
sage: phiA_B = E_A.isogeny(secretA_B)
sage: phiA_B
Isogeny of degree 4 from Elliptic Curve defined by y^2 = x^3 + 81*x + 247
   over Finite Field of size 647 to Elliptic Curve defined by y^2 = x^3 +
   485*x over Finite Field of size 647
sage: secretB=phiA_B.codomain().j_invariant()
sage: secretB
434
```

Bob does the same calculations as Alice with his own private values. When everything is done he should end up with an isomorphic curve that has the same j-invariant as the curve Alice calculated.

Listing 5.7: Same j-invariant values for both parties

```
sage: secretA == secretB
True
```

The final check in listing 5.7 shows that both secret values are the same and that the SIDH algorithm has worked successfully. The j-invariant can now be used as the private shared secret in further communications.

6 Java Huldra Library

The Huldra-Library is a faster alternative to the classical Java internal BigInteger class. BigInteger usually provides the capability to calculate Integer values with arbitrary bit-length. Although the BigInteger class provides a good basis, it is not optimized nor intended to use for cryptographic applications. The main difference between the BigInt class from the Huldra-Library and the BigInteger class from Java is the different usage of fast computation methods and algorithms, as well as accessibility and fast data and object storage.[47][48][49]

6.1 Immutable vs. Mutable Objects

Mutability and Immutability is a very controversial topic in Java programs. The coding style differs a lot for each way of allocating objects and it depends on the programmer, the wanted speed and the computational complexity to decide whats better for the overall programming scheme.
A mutable object is a object that can be directly used by its own mathematical operations and stores the result by overwriting existing values.[50]

```
BigInt value1 = new BigInt("1");
BigInt value2 = new BigInt("2");

value1.add(value2);
```

In the example above, a mutable BigInt object named *value1* was created. A second BigInt object named *value2* was also created and then added to value1. The add function now takes the object *value2* and adds it to *value1* internally. The result is directly stored in *value1*. This means that no intermediate object had to be created and the operational and memory overhead is kept to a minimum.[47]

This is a huge difference to immutable objects. Immutability means, that during a mathematical operation, the object can't be changed. The impact of this property is, that the result of an addition can't be directly saved back to the object.[48]

```
        BigInteger value1 = new BigInteger("1");
        BigInteger value2 = new BigInteger("2");

        BigInteger value3 = value1.add(value2);
```

In the example above, an immutable BigInteger object named *value1* was created. A second BigInteger object named *value2* was also created and then added to value1. Because the immutable object can't be changed directly and is returned to the calling function. This is why a new BigInteger object *value3* needs to be created to hold the sum of *value1* and *value2*.

6.2 Performance

The performance is mainly demonstrated by the BigInt mutable property of the Huldra library. Any value that is created while computing mathematical operations can be directly saved back into the object and do not need to be stored in new objects. This provides a great advantage for IOT devices and smart cards in particular, as it allows limited resources to be used efficiently. Because BigInteger is immutable, a new object must be created and assigned for each cached result during the execution of calculations. For longer calculations such as multiplications or divisions, this could create a massive performance impact. BigInt from the Huldra library, on the other hand, performs the operation directly on the internal array used to store the digits. All digits are stored in base 2 and are defined by a value that determines the number length. BigInt stores and extends the array when it no longer has capacity. This function can theoratically compared to the java-internal ArrayList.[47]

Table 6.1: Performance benchmark BigInteger vs. BigInt

	BigInteger	Huldra-Library BigInt
Addition	1.840s	0.832s
Subtraction	1.287s	0.574s
Multiplication	0.714s	0.479s
Many small multiplications	0.852s	0.413s
Big multiplications	0.279	0.129
Division	2.608s	2.069s

The direct benchmark comparison in 6.1 shows the real advantage of using the external BigInt Library instead of the native BigInteger class. The benchmark was done with an Intel i7 6700HQ 2.6 GHz processor with four processing cores. The following mathematical operations have been tested:

- Addition: Adding two 100.000-digit numbers 100.000 times
- Subtraction: Subtracting two 100.000-digit numbers 100.000 times
- Multiplication: Multiplication of a 300-digit number 1000 times
- Many small multiplications: Calculation of 50.000 factorial
- Big multiplication: Multiplication of two 500.000-digit numbers
- Division: Divide two 400.000-digit numbers 1000 times

The loop values have been exaggerated to provide readable results in the range of seconds. This is not comparable to a SIKE computation because the loop values are lot shorter, but provide a good preview into the future when more quantum security is needed or algorithms do a lot more iterations. For example more iterations in the SIKE Montgomery ladder and more isogenic calculations.

6.3 BigInt mathematical operation examples

In the following section there are some short examples of the functionality of the BigInt class.
The addition consits of multiple functions with the two most important ones being *uadd* and *uaddMag*. These two handle the actual addition while *realloc* handles changes in the memory size and expands it if needed.

uadd, which stands for for unsigned add, first checks if the value that needs to be added is positive. When this is not the case, the function calls the subtraction with *usubMag*. Otherwise it calls the *uaddMag* function that includes the addition.
It is important to note, that the variable *a* is not the the actual value that needs to be added, but the representative magnitude of the value. This magnitude was already calculated when defining a BigInt object and always represents any value stored in such an object. This is why only a BigInt can be added to another BigInt object and no intercompatibility is given.

```
/*
* Adds an unsigned int to this number.
*
* @param a       The amount to add (treated as unsigned).
* @complexity    O(n)
* @amortized     O(1)
*/
public void uadd(final int a){

if(sign<0){
    if(len>1 || (dig[0] AND mask)>(a AND mask)){
        usubMag(a);
        return;
    }
```

```
            sign = 1;
            dig[0] = a-dig[0];
            return;
        }
        uaddMag(a);
    }
```

The *uaddMag* adds the correct magnitude to the existing value. This means that it first checks the new size and correctly allocates memory. Afterwards it increases the array represented by *dig* and correctly moves over the carry bit. If it reaches a certain size while increasing the value, the *realloc* function is called on demand. The function itself does not have any return operators because BigInt is mutable and changes itself to represent the actual value.

```
    private void uaddMag(final long a){

        if(dig.length<=2){
            realloc(3);
            len = 2;
        }

        final long ah = a>>>32, al = a AND mask;
        long carry = (dig[0] AND mask) + al;
        dig[0] = (int)carry;
        carry >>>= 32;
        carry = (dig[1] AND mask) + ah + carry;
        dig[1] = (int)carry;
        if((carry>>32)!=0){
            int i = 2;
            for(; i<len AND ++dig[i]==0; i++);
            if(i==len){
                if(len==dig.length){
                    realloc();
                    dig[len++] = 1;
                }
            }
        }else if(len==2  AND  dig[1]==0){
            --len;
        }
    }
```

7 SIKE-Java

7.1 The SIDH/SIKE implementation

This chapter shows and describes the complete implementation of SIDH/SIKE in Java. The mathematical and programmatic purpose of each class is described in great detail and attempts to serve as a basic framework and guide for future implementations. Each class has different main functionalities, which are shown first. At the end of each class, generic functions and class attributes are described. The end of the chapter deals with the main engine class of SIDH/SIKE and shows how the cryptosystem works in general.

7.2 Finite Field $\mathbb{F}_p$ class implementation

The Finite Field $\mathbb{F}_p$ class implementation is the base class for all numeric values in the cryptosystem. The class itself is based on the BigInt implementation of the Huldra-Library as described in chapter 6.

7.2.1 Implementation of the Tonelli-Shanks square root algorithm in $\mathbb{F}_p$

The Tonelli-Shanks algorithm[51] is a fast way to compute a square root in a finite field using modular arithmetics. The speed of the algorithm depends on the amount of quadratic non-residues and quadratic residues while computing the square root.[52]

```
public void sqrt(){
    if(value.equals(Engine.ZERO)){
        value = Engine.ZERO.copy();
        return;
    }

    if(!isQuadraticResidue()){
        value = null;
        return;
    }
```

The algorithm first checks if the value is already zero or the value is not a quadratic residue of the finite fields prime, which would lead to an immediate return of the algorithm. If the number is not a quadratic residue, the program stops with an error.[51]

```java
        BigInt q = Engine.PRIME.copy();
        q.sub(Engine.ONE);
        int s = numPowersOf2(q);
        q.shiftRight(s);

        Fp z = quadraticNonResidue();

        BigInt exp = q.copy();
        exp.add(Engine.ONE);
        exp.shiftRight(1);
```

In the next part of the algorithm the setup for the calculation loop needs to be done. s equals the the number of times 2 divides q which is the finite field prime number subtracted by 1. After that, q is being bit-shifted to the right by s, which in mathematical terms is the division by 2^s. The variable z is a random quadratic non-residue in $\mathbb{F}_p$. It is important to note, that half of all elements in this set will be quadratic non-residues. The search for a quadratic non-residue will be explained in the implementation in the quadratic non-residue function. The value exp is the original prime $q + 1$ divided by two. Everything will be calculated in the modular arithmetic field.

```java
        z.pow(q);
        Fp t = copy();
        Fp r = copy();

        t.pow(q);
        r.pow(exp);

        int i;
        while(!t.equals(Engine.FPONE)){
            i = t.findLog2Order();
            exp = Engine.ONE.copy();
            exp.shiftLeft(s-i-1);

            z.pow(exp);
            s = i;
            r.mult(z);
            z.square();
            t.mult(z);
        }

        value = r.getValue();
    }
```

The last part of the Tonelli-Shanks algorithm consists of the setup the the variables $z = z^q$, t =value^p, r =value^{exp}.The loop runs until t equals 1. When that happens, the square root will be in the variable r. i equals to the binary log order of t in every iteration of the loop.This also adjusts the exponent of $z = z^{exp}$ which is used in the multiplication of $r = r * z$ and $t = t * z$ until $t \equiv 1 \mod p$.[52]

7.2.2 Implementation of the quadratic residue function in the finite field $\mathbb{F}_p$

```
    private boolean isQuadraticResidue(){
        BigInt p = Engine.PRIME.copy();
        Fp base = copy();

        p.sub(Engine.ONE);
        p.shiftRight(1);
        base.pow(p);

        if(base.equals(Engine.FPONE)){
            return true;
        }
        return false;
    }
```

The quadratic residue function checks if a value is a quadratic residue of the prime p in the finite field $\mathbb{F}_p$[53]. This function can be explained with the Legendre symbol $\left(\frac{a}{p}\right)$.[54]

$$\left(\frac{a}{p}\right) \equiv a^{\frac{p-1}{2}} \pmod p$$

$$\left(\frac{a}{p}\right) = \begin{cases} 1, & \text{if } a \text{ is a quadratic residue and } a \not\equiv 0 \pmod p. \\ 0, & \text{if } a \equiv 0 \pmod p. \\ -1, & \text{if } a \text{ is a quadratic non-residue.} \end{cases}$$

The function will return 1(true) if a is a quadratic residue and $a \not\equiv 0 \pmod p$. Otherwise it will return 0(false) for the other two cases, since they do not need to be differentiated in the SIDH-Algorithm.[55]

7.2.3 Implementation of the quadratic non-residue function in the finite field $\mathbb{F}_p$

```
private Fp quadraticNonResidue(){
    Fp z;
    do{
        z = new Fp(Engine.genRandom());
    }while(z.equals(Engine.FPZERO) || z.isQuadraticResidue());
    return z;
}
```

In the SIDH-Algorithm some functions like the square root of a number in the finite field $\mathbb{F}_p$ need a quadratic non-residue value for further calculations. The quadratic non-residue function generates secure random values in the finite field $\mathbb{F}_p$ and checks if the generated value meets the criteria by utilizing the aforementioned quadratic residue function. Although this is a function that utilizes a random value, which means that the positive outcome of this function is also random, it produces a correct residue quite fast because half the field consists of quadratic non-residue values.[53][56]
The main downside of this implementation is the usage of the Java proprietary SecureRandom class and the genRandom() function which is further described in the Engine class implementation. This random generation needs to be changed depending on the underlying system on which this implementation is going to run on. For example, a Java smart card needs to use builtin pseudo-random number generators within the genRandom() function.[57][58]

7.2.4 Implementation of the pow function in the finite field $\mathbb{F}_p$

```
public void pow(BigInt exp){
    BigInt e = exp.copy();
    Fp base = copy();
    value = Engine.ONE.copy();

    while(e.compareTo(Engine.ZERO) == 1){
        if(e.testBit(0)){
            mult(base);
        }
        e.shiftRight(1);
        base.square();
    }
}
```

The pow function utilizes the so-called exponentiation by squaring algorithm, which is a common method for fast computation of large numbers. This algorithm is much faster than multiplying a value with itself n-times[59]. The mathematical notation of this recursive algorithm is the following:

$$(x^n) = \begin{cases} x(x^2)^{\frac{n-1}{2}}, & \text{if } n \text{ is odd.} \\ (x^2)^{\frac{n}{2}}, & \text{if } n \text{ is even.} \end{cases}$$

7.2.5 Implementation of the power-of-2 amount function in the finite field $\mathbb{F}_p$

```
private int numPowersOf2(BigInt n){
    int s = 0;
    BigInt q = n.copy();
    while(!q.testBit(0)){
        s++;
        q.shiftRight(1);
    }
    return s;
}
```

This function the returns an integer value which is equal to the number of $0 * 2^n$-bits that have been set in this number. The loop iterates itself while the value is an even value.

7.2.6 Implementation of the binary logarithmic order function in the finite field $\mathbb{F}_p$

```
private int findLog2Order(){
    Fp tsq = copy();
    int i = 0;

    while(!tsq.equals(Engine.FPONE)){
        i++;
        tsq.square();
    }
    return i;
}
```

The *log2Order* makes use of the $\mathbb{F}_p$ square root function by recursively using it on the same number until it reaches the value 1. Each iteration of the loop increases an integer counter which is returned after the loop is finished.[60]

7.2.7 Implementation of a fast multiplicative inverse function in the finite field $\mathbb{F}_p$

The multiplicative inverse is besides the isogenic calculations, one of the most important function in the SIDH-Algorithm. Because this function is widely used in the underlying mathematical construct, performance and time optimizations are a critical factor for the complete SIDH runtime.[61][62]

The algorithm below is the also known as right-shift algorithm or Penk's algorithm that computes a modular inverse. Some performance optimization have been made to the Penk's algorithm to achieve a faster and easier hardware implementation. Although this is the Java implementation, it also takes advantage of a faster and lower performance demanding code.[61][63]

```
public void multiplicativeInverse(){
        BigInt u = Engine.PRIME.copy();
        BigInt v = value.copy();
        BigInt r = Engine.ZERO.copy();
        BigInt s = Engine.ONE.copy();
        BigInt k = Engine.ZERO.copy();
```

The first part of the right-shift algorithm is the setup of all variables. u is allocated with the SIDH prime number. It will gradually right shift until it reaches zero. v is the starting value that needs to be inverted. r holds the intermediate and final value of the right-shift algorithm. s is used only for intermediate calculations. k will hold the number of right-shifts after each loop trough the extended euclidean algorithm.[61]

```
        while(v.compareTo(Engine.ZERO) == 1){
            if(!u.testBit(0)){
                if(!r.testBit(0)){
                    u.shiftRight(1);
                    r.shiftRight(1);
                    k.add(1);
                }else{
                    u.shiftRight(1);
                    r.add(Engine.PRIME);
                    r.shiftRight(1);
                    k.add(1);
                }
            }else if(!v.testBit(0)){
                if(!s.testBit(0)){
                    v.shiftRight(1);
                    s.shiftRight(1);
                    k.add(1);
                }else{
                    v.shiftRight(1);
                    s.add(Engine.PRIME);
                    s.shiftRight(1);
                    k.add(1);
                }
```

The next part of the algorithm continuously right-shifts both values u and v which are even and odd. If the value is odd, the prime number(which is odd) is added first.

```
            }else{
                BigInt x = u.copy();
                x.sub(v);
                if(x.compareTo(Engine.ZERO) == 1){
                    u = x.copy();
                    r.sub(s);
                    if(r.compareTo(Engine.ZERO) == -1){
                        r.add(Engine.PRIME);
                    }
                }else{
                    v = x.copy();
                    v.mul(-1);
                    s.sub(r);
                    if(s.compareTo(Engine.ZERO) == -1){
                        s.add(Engine.PRIME);
                    }
                }
            }
        }
        if(r.compareTo(Engine.PRIME) == 1){
            r.sub(Engine.PRIME);
        }
        if(r.compareTo(Engine.ZERO) == -1){
            r.add(Engine.PRIME);
        }
        value = r;
    }
```

The final part of the algorithm converts the values r and s, if they happen to be negative, to positive values in the same residue class by adding the prime number. As a final task, the multiplicative inverse, which is stored in r is saved back into the $value$ that was initially given to the function.[62]

7.2.8 Implementation of general class functions for the finite field $\mathbb{F}_p$ class

Several class functionalities are needed for a fully functional $\mathbb{F}_p$ calculation class. It provides an intermediate step between the BigInt data-structure from the Huldra-Library and the $\mathbb{F}_{p^2}$ class which uses the $\mathbb{F}_{p^2}$ data-structure for its underlying operations.

```
public class Fp {

    private BigInt value;

    public Fp(BigInt value) {
        this.value = value.copy();
        this.value.mod(Engine.PRIME);
    }
    public Fp(String value) {
        this.value = new BigInt(value);
        this.value.mod(Engine.PRIME);
    }

    public Fp copy(){
        return new Fp(value.copy());
    }

    public void assign(Fp fp){
        value.assign(fp.getValue().copy());
    }

    public void mod(){
        value.mod(Engine.PRIME);
    }
```

The first part of the general functions focuses on data initialization and value assignments. Every important function needs to be implemented twice with support for a String and a BigInt parameter for intercompatibility of different parts of the program. Every new allocated $\mathbb{F}_p$ object needs to pass its values via a copy function from one object to another. Otherwise it would be a call by reference which would loose the original value.

The copy function is implemented in the Huldra-Library and copies the data-fields which hold the values. The copy function of the $\mathbb{F}_p$ class returns a new $\mathbb{F}_p$ object. The assign function replaces the current value with a new one. After a new $\mathbb{F}_p$ has been created, the modulo function needs to be called, to ensure that the value is within bounds of the prime number p.

```
    public BigInt getValue() {
        return value;
    }

    public void setValue(BigInt value) {
        this.value = value;
    }
```

The get- and set-functions provide typical java functionality for setting a value or reading from the object. These functions are rarely used in the SIDH-Algorithm, because performance optimizations try to keep the reading and creating new objects to a minimum thus leading to a re-usage of most objects while calculating the algorithm.

```
    public void add(BigInt x){
        value.add(x);
        mod();
    }
    public void add(Fp x){
        add(x.getValue());
    }

    public void sub(BigInt x){
        value.sub(x);
        mod();
    }

    public void sub(Fp x){
        sub(x.getValue());
    }

    public void mult(BigInt x){
        value.mul(x);
        mod();
    }

    public void mult(Fp x){
        mult(x.getValue());
    }
    public void square(){
        mult(value);
    }
```

The classical mathematical functions of the $\mathbb{F}_p$ class mostly use the implementations of the Huldra-Library which already provides these basic functions. By adding the modulo operation after each mathematical operation, the addition, subtraction and multiplication in $\mathbb{F}_p$ can be provided. The square function utilizes the multiplication. Because it is used very rarely, a more performance optimized square function would make no measurable difference. Every basic function needs to provide support for BigInt and String parameters to ensure inter-compatibility.

```
    public void shiftRight(int amount){
        value.shiftRight(amount);
    }
```

In the next part of the $\mathbb{F}_p$ class, the functions mainly contain bit-wise operators and comparative operations. In the SIDH base algorithm only the right shift operation is needed. Although the Huldra-Library also provides fast lefts shifts, it is not implemented in $\mathbb{F}_p$.

```
    public boolean testBit(int bit){
        return value.testBit(bit);
    }
    public boolean isZero(){
        return value.isZero();
    }
```

The *testBit* function checks if a bit is set on a given position. The function responds with a Boolean value corresponding to true when the bit is set or false if the value on the given position is zero. The *isZero* function utilizes the functionality of the *testBit* function and checks if every bit in a number is zero, which in conclusion means that the whole value of the allocated number equals to zero.

```
    public int compareTo(Fp fp){
        return value.compareTo(fp.getValue());
    }
    public boolean equals(Fp fp) {
        return fp.getValue().equals(value);
    }
```

The *compareTo* functionality compares to $\mathbb{F}_p$ values and returns an Integer value, if one of the following conditions are met.

$$a.\text{compareTo}(b) = \begin{cases} 1, & \text{if } a \text{ greater than } b. \\ 0, & \text{if } a \text{ is equal to } b. \\ -1, & \text{if } a \text{ is smaller than } b. \end{cases}$$

The *equals* method uses the BigInt *compareTo* functionality of the Huldra-Library and returns a true Boolean value if the underlying function returns zero. The fast implementation of these function is important for the SIDH algorithm, because loops and comparative algorithm sections can then be executed in a more optimized way.

```
    public void print(){
        System.out.println(toString());
    }

    @Override
    public String toString() {
        return value.toString();
    }
}
```

The last functions of the finite field $\mathbb{F}_p$ class are general print and String functionalities for debug and visualization purposes.

7.3 Finite Field $\mathbb{F}_{p^2}$ class implementation

The $\mathbb{F}_{p^2}$ class builds onto the $\mathbb{F}_p$ class and the underlying Huldra-Library. It adds support for the complex number plane in $\mathbb{F}_{p^2}$. The SIDH-Algorithm bases its calculations almost exclusively on this class. Only some features like loop variables and temporary values that are kept rather small by using the $\mathbb{F}_p$ class.

7.3.1 Implementation of complex number multiplication in $\mathbb{F}_{p^2}$

```
public void mult(Fp2 fp2){
    BigInt c = fp2.getX0().getValue().copy();
    BigInt d = fp2.getX1().getValue().copy();

    Fp ac = x0.getValue().copy();
    Fp bd = x1.getValue().copy();

    ac.mult(c);
    bd.mult(d);

    x1.add(x0);
    c.add(d)
    x1.mult(c)
    x1.sub(ac)
    x1.sub(bd)

    x0 = ac;
    x0.sub(bd)
}
```

Multiplication in the complex finite field is one of the most important functions, along with multiplicative inverse and isogenic calculations in the SIDH algorithm. For this reason it is of utmost importance that these functions are implemented as optimized and resource-saving as possible. The multiplication function can be implemented in a classical way. This classical implementation uses four allocations, four multiplications and one subtraction and addition.

$$\mathbb{F}_{p^2}\, x = (a + b_i)$$

$$\mathbb{F}_{p^2}\, y = (c + d_i)$$

$$x * y = (ac - b_i d_i) * (ad_i + bc_i)_i$$

Although the classical implementation does not have a huge code and computational complexity, there are faster algorithms available. The following math segment shows the Karatsuba[64][65] algorithm, which is a complex and big number multiplication method that is widely used in computational applications and cryptography. The Karatsuba algorithm looks more compute intensive at first glance, but it only has three multiplications, two additions and three subtractions. This is actually faster than the classical algorithm, because a multiplication of a big number is a lot more resource intensive than any other base operation. In the Karatsuba algorithm, certain multiplied values can be reused, which makes the code faster than the classical multiplication. The downside is, that more variables need to be allocated. This can be ignored, since memory allocations become less of a problem in a quantum-computing future.[65]

$$\begin{aligned}
\mathbb{F}_{p^2}\, x &= (a + b_i) \\
\mathbb{F}_{p^2}\, y &= (c + d_i) \\
x * y &= (ac - b_i d_i) * ((a + b_i)(c + d_i) - ac - b_i d_i)_i
\end{aligned}$$

The algorithm calculates $a * c$, $b_i * d_i$. These values used twice in the calculation. Winograds theorem[66] proves, that the minimum number of multiplications needed in a complex number multiplication is limited to three. This means, that the Karatsuba algorithm utilizes the least amount of multiplications possible. The most promising way to further increase multiplication speed, is to optimize functions and algorithms further down the computational tree. This means that the underlying multiplication function from the Huldra library, which does the multiplication for two big numbers in the finite field $\mathbb{F}_p$ instead of $\mathbb{F}_{p^2}$ needs to be optimized the correct way. This is not an easy task, since finding the correct trade-off between speed, performance and security has been an ongoing problem in many cryptographic applications.
In the implementation of the SIDH-Algorithm, a derivative of the Karatsuba algorithm is also used in the $\mathbb{F}_p$ multiplication, to have a certain red thread throughout all computations. This also provides the possibility to compare different implementations to each other, and provide an analysis on the performance and memory usage of different algorithms.

7.3.2 Implementation of the complex number multiplicative inverse in $\mathbb{F}_{p^2}$

```
    public void multiplicativeInverse(){
        Fp x0t = x0.copy();
        Fp x1t = x1.copy();

        x0t.square();
        x1t.square();

        x0t.add(x1t);
        x0t.multiplicativeInverse();

        x1.negate();

        x0.mult(x0t);
        x1.mult(x1t);
    }
```

The multiplicative inverse of a complex number in the finite field is dependant on the speed of four components in the mathematical implementation. A fast basic multiplication, squaring in $\mathbb{F}_{p^2}$, multiplicative inverse in $\mathbb{F}_p$ and a fast addition. When these three things are as optimized as possible, then a fast multiplicative inverse in $\mathbb{F}_{p^2}$ is also possible, because the computational algorithm itself is rather short.

7.3.3 Implementation of the complex number pow function in $\mathbb{F}_{p^2}$

```
    public void pow(BigInt exp){
        BigInt e = exp.copy();
        Fp2 base = copy();
        x0 = Engine.FP2ONE.getX0().copy();
        x1 = Engine.FP2ONE.getX1().copy();

        while(e.compareTo(Engine.ZERO) == 1){
            if(e.testBit(0)){
                mult(base);
            }
            e.shiftRight(1);
            base.square();
        }
    }
```

The java SIDH-Implementation does not use the classical multiplicative method to do exponential calculations of complex numbers. Thea algorithm described above, is the so-called right-to-left binary exponentiation, which is a combination of the exponentiation by squaring algorithm and the usage of modulo arithmetic. To do calculations with this combined algorithm, the exponent must be considered as a power of the number two. Assuming that the exponent is $e = 2^n$, the result could be calculated by squaring the base number n times. If the exponent is not a power of two, additional multiplications must be performed. If a division by two without remainder is possible, then the base must be squared and the exponent divided with this intermediate result. If there is a remainder of the modulus operation, the current base must be multiplied by the intermediate result. The division is done by right shifts to calculate the algorithm according to the computational rules of the finite field $\mathbb{F}_{p^2}$.[59][67][68]

7.3.4 Implementation of the quadratic residue function in $\mathbb{F}_{p^2}$

```
private boolean isQuadraticResidue(){
    BigInt p = Engine.PRIME.copy();
    Fp2 base = copy();
    p.mul(p);
    p.sub(Engine.ONE);
    p.shiftRight(1);
    base.pow(p);
    base.print();
    if(base.equals(Engine.FP2ONE)){
        return true;
    }
    return false;
}
```

The quadratic residue can logically divide an integer x modulo p into different congruence classes. Such an integer in a congruence class is called a quadratic residue of p if there exits a integer y so that $x^2 \equiv y \mod p$. If that is not the case for the integer x, it is called a quadratic non-residue.
This can be best explained with the Legendre symbol in $\mathbb{F}_p$ and $\mathbb{F}_{p^2}$ as shown in 4.2.7 and 7.2.2.

7.3.5 Implementation of the complex number square roots $\mathbb{F}_{p^2}$

```
    public void sqrt(){
        if(equals(Engine.FP2ZERO)){
            return;
        }
        if(!isQuadraticResidue()){
            x0 = null;
            x1 = null;
            return;
        }

        BigInt pmod4 = Engine.PRIME.copy();
        BigInt three = Engine.THREE.copy();
        pmod4.and(three);

        if(pmod4.equals(three)){
            sqrt3mod4();
        }else{
            sqrt1mod4();
        }
    }
```

Calculating a square root in the finite field $\mathbb{F}_{p^2}$ takes more steps into consideration than it seems at first glance. The difficult problem is, that the case $p = 3 \pmod 4$ is pretty simple to calculate and $p = 1 \pmod 4$ is a much harder case. This is because $p = 4k + 3$ forms a quadratic residue group that has odd order $\frac{p-1}{2} = 2k + 1$. This means that for a quadratic residue q_p the unique square root is $x = q_p k + 1 \pmod p$[Lagrange 1769]. This does not apply for the case $p = 1 \pmod 4$. A good example for this situation is, that -1 is a quadratic residue, but it is not possible to find a power of -1 that can be a square root of -1. There are many procedures that try to solve this problem. Unfortunately all algorithms that are used are probabilistic. The $p = 1 \pmod 4$ case is calculated with an optimized alternative version of the Tonelli-Shanks/Cipolla algorithm.[69][70][71]

The *sqrt* function checks if the value is equal to zero. If that is the case, the function would exit and return to the calling function. Afterwards the value is checked if it is a quadratic residue in $\mathbb{F}_{p^2}$. Only values that are a quadratic residue can be used in the square root function. The algorithm will run into an error otherwise. The return value for the case of a quadratic non-residue is not defined and is generally not going to happen unless the SIKE algorithm was implemented in a wrong way. This is way returns a NULL value that force quits the program in the calling function.

```
    private void sqrt3mod4(){

        Fp2 neg1 = Engine.FP2ONE.copy();
        neg1.negate();

        BigInt p = Engine.PRIME.copy();
        p.shiftRight(2);

        Fp2 a1 = copy();
        a1.pow(p);
        Fp2 a2 = copy();
        a2.mult(a1);
        a1.mult(a2);
        if(a1.equals(neg1)){
            a2.multByI();
            x0 = a2.getX0();
            x1 = a2.getX1();
            return;
        }
        p = Engine.PRIME.copy();
        p.shiftRight(1);
        a1.add(Engine.FP2ONE);
        a1.pow(p);
        a1.mult(a2);
        x0 = a1.getX0();
        x1 = a1.getX1();
    }
```

The algorithm above can be best explained with a simplified example. The code is longer than the mathematical example, because it is the adapted version for $\mathbb{F}_{p^2}$ and tries to use the least amount of memory as possible whilst delivering good performance. The basic algorithm for computing a square root where $p = 3 \pmod 4$ follows the following notation:

$$p = 4k + 3$$

$$p = 7 = 4 * 1 + 3 \text{ with } k = 1$$

$$k = k + 1 = 2 \text{ is a quadratic residue}$$

$$2^1 = 4 \text{ is a square root}$$

$$4^2 = 16 \pmod 7 \equiv 2$$

```
    private void sqrt1mod4(){
        Fp a0 = x0.copy();
        Fp a1 = x1.copy();
        x0.square();
        x1.square();
        a1.add(a0);
        a1.sqrt();
        a1.sub(x0);
        if(a1.equals(Engine.FPZERO)){
            x0.sqrt();
            x1 = Engine.FPZERO.copy();
        }
        a1.div2();
        a1.sqrt();
        a0 = a1.copy();
        a0.multiplicativeInverse();
        a0.mult(x1);

        x0 = a0;
        x1 = a1;
    }
```

The $p = 1 \pmod 4$ algorithm implementation takes longer to compute, since a multiplicative inverse is needed in the algorithm and two square roots in the finite field $\mathbb{F}_p$ need to be found. $p = 1 \pmod 4$ is the least possible case for a square root in $\mathbb{F}_{p^2}$ and can be optimized by using more advanced algorithms. Altough speed comes often with a trade-off of either more memory usage or a more sophisticated execution.

7.3.6 Implementation of the multiplication by I function in $\mathbb{F}_{p^2}$

```
    public void multByI(){
        Fp temp = x0.copy();

        x0 = x1.copy();
        x0.negate();

        x1 = temp.copy();
    }
}
```

The multiplication by I($\Im$), which is the imaginary part of the a number on the complex number plane, describes the left-rotation of complex number on the plane by 90 degrees. The calculation is done by switching the real part($\Re$) of the complex number with the imaginary part($\Im$). After that, the new real part is negated. The following mathematical notation is an example of this rotation.

$$\mathbb{F}_{p^2}\, z = (a + b_i)$$

$$\mathbb{F}_{p^2}\, z_{\text{rot}} = (-b + a_i)$$

$$\mathbb{F}_{p^2}\, z = (3 + 4_i)$$

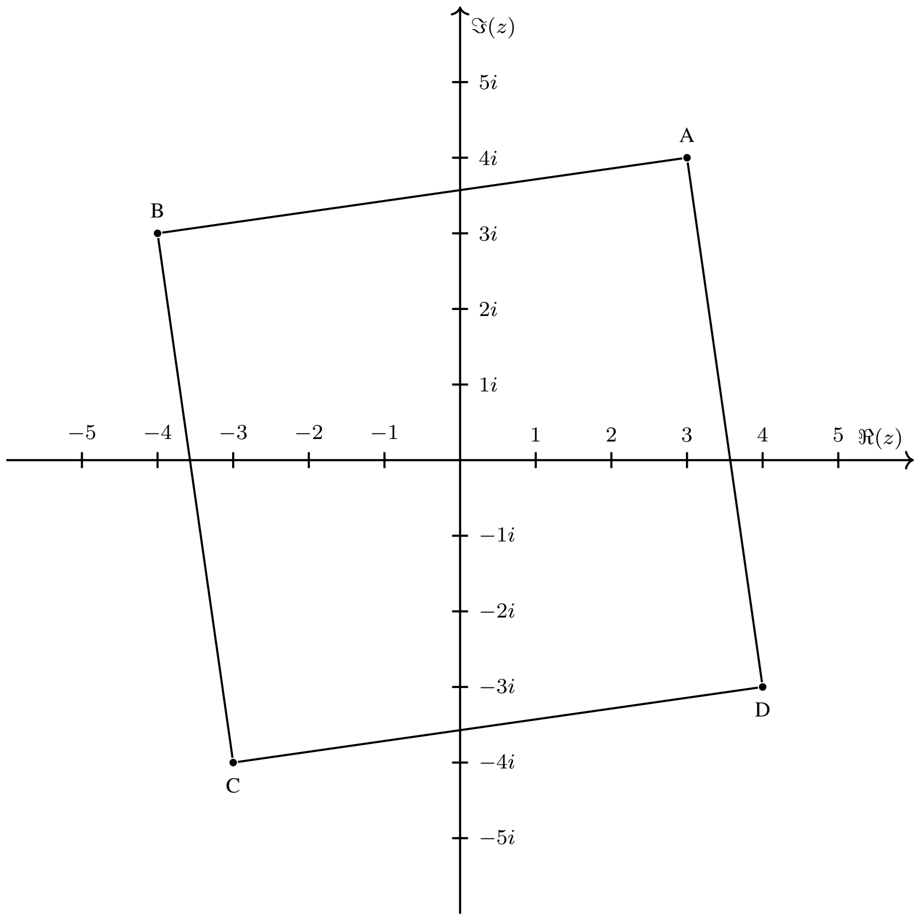
$$A(z) = (3 + 4_i)$$

$$B(z) = (-4 + 3_i)$$

$$C(z) = (-3 - 4_i)$$

$$D(z) = (+4 - 3_i)$$

Figure 7.1: Rotation of a coordinate on the complex plane



```
public class Fp2 {

    private Fp x0, x1;

    public Fp2(Fp x0, Fp x1) {
        this.x0 = x0.copy();
        this.x1 = x1.copy();
    }
    public Fp2(BigInt x0, BigInt x1) {
        this.x0 = new Fp(x0.copy());
        this.x1 = new Fp(x1.copy());
    }
    public Fp2(String x0, String x1) {
        this.x0 = new Fp(new BigInt(x0));
        this.x1 = new Fp(new BigInt(x1));
    }
```

The first part of the finite field $\mathbb{F}_{p^2}$ class is the general data structure that holds the real part($\Re$) and the imaginary part($\Im$) of the complex number. These values are represented by x_0 for $\Re$ and x_1 for $\Im$. Each part is a value in the finite field $\mathbb{F}_p$. The class needs three input methods for compatibility, easier development and future expansions. A number in the complex field $\mathbb{F}_{p^2}$ can be allocated by using either two $\mathbb{F}_p$, BigInt or String values. The latter two are both being converted to a value in $\mathbb{F}_p$. If a value is too big to initially exist in $\mathbb{F}_{p^2}$ or $\mathbb{F}_p$ it is automatically run through the modulo operation.

```
    public Fp getX0() {
        return x0;
    }
    public void setX0(Fp x0) {
        this.x0 = x0;
    }
    public Fp getX1() {
        return x1;
    }
    public void setX1(Fp x1) {
        this.x1 = x1;
    }
```

Get- and set-methods are used for easier access to the real part($\Re$) and the imaginary part($\Im$). This is most commonly used in the key-recovery and isogenic algorithms.

```
    public void add(Fp2 fp2){
        x0.add(fp2.getX0());
        x1.add(fp2.getX1());
    }

    public void sub(Fp2 fp2){
        x0.sub(fp2.getX0());
        x1.sub(fp2.getX1());
    }

    public void negate(){
        x0.negate();
        x1.negate();
    }

    public void square(){
        mult(this);
    }
```

The class also provides the add-, sub-, negate- and square-function to provide the needed mathematical operations for complex numbers in SIKE. The first three functions are straightforward implementations with the $\mathbb{F}_p$ optimized additions and subtractions. The negation is a multiplication by -1 followed with a modulo operation. The square function is utilizing the $\mathbb{F}_{p^2}$ multiplication function and the $\mathbb{F}_p$ multiplication function on the second layer of operations. Both are optimized with the Karatsuba multiplication algorithm.

```
        @Override
    public String toString() {
        return ""+ x0.toString() + " + " + x1.toString() + "*i";
    }

    public void print(){
        System.out.println(toString());
    }
```

The *toString* and *print* functions provide debugging possibilities and give the option to print the current status, values and general output to the user or developer.

```
    public Fp2 copy(){
        return new Fp2(x0.copy(), x1.copy());
    }

    public void assign(Fp2 fp2){
        x0.assign(fp2.getX0());
        x1.assign(fp2.getX1());
    }
```

The *copy* and *assign* function are important parts in a program with mutable objects. Since the $\mathbb{F}_{p^2}$ objects are reused as often as possible, there is the need to copy the whole objects or easily assign values to it. This is mostly done by utilizing the underlying copy functions from $\mathbb{F}_p$ and the BigInt library. A *copy* operation copies the whole value, which is allocated in the RAM(Random Access Memory) to a new and free space. The *assign* function overwrites the space in the memory. This removes the value that was stored there beforehand. Depending on the size of the new value it automatically increases or decreases the allocated space, thus optimizing the general memory usage.

```
    public boolean isZero(){
        return x0.isZero()  x1.isZero();
    }
    public boolean equals(Fp2 fp2) {
        return fp2.getX0().equals(x0)  fp2.getX1().equals(x1);
    }
}
```

The last part of this class consists of comparative functions. The *isZero* method returns a true Boolean value when both, x_0 and x_1 are equal to zero in $\mathbb{F}_p$. The *equals* method returns a true Boolean value, when both, the real and the imaginary part of a complex number in $\mathbb{F}_{p^2}$ are equal.

7.4 Montgomery and isogeny math class implementation

7.4.1 Implementation of the Montgomery point doubling method

The affine point doubling on a Montgomery curve(4.5.2) is a special function for a point P to calculate $[2]P$ faster than with a coordinate addition in the complex plane.

```
    public Point xDBL(Point p, Fp2 a, Fp2 b){

        if(p.isInfinite()){
            return p;
        }
```

In the first parth of the algorithm, the points need to be checked in order to work correctly. In the if-section the point itself is returned to the caller, if the point P is the the point at infinity($\mathcal{O}$). This would lead to an error in further calculations. The point is not being checked for its order since the caller needs to make sure to only send the correct points, whose order do not divide two, into the function.The parameters for this algorithm are the point P, that needs to be doubled and the coefficients A and B of the Montgomery curve.[39]

```
        Fp2 t0 = p.x.copy();
        t0.square();
        Fp2 t1 = t0.copy();
        t1.add(t0);
        t0.add(t1);
        t1.assign(a.copy());
        t1.mult(p.x);
        t1.add(t1);
        t0.add(t1);
        Fp2 t2 = b.copy();
        t0.add(t2);
        t1.assign(b.copy());
        t1.mult(p.y);
        t1.add(t1);
        t1.multiplicativeInverse();
```

Since this algorithm follows the principle of the affine coordinate doubling, the B coefficient needs to be used within the algorithm. Although there is a faster algorithm, which utilizes the projective plane, this implementation of the SIDH-Algorithm tries to optimize the existing reference algorithms from the NIST submission.[39]

The multiplicative inverse is besides the multiplications, the most resource and performance hungry operation in the affine coordination doubling.

```
        t0.mult(t1);
        t1.assign(t0.copy());
        t1.square();
        t2.assign(b.copy());
        t2.mult(t1);
        t2.sub(a);
        t2.sub(p.x);
        t2.sub(p.x);
        t1.mult(t0);
        t1.mult(b);
        t1.add(p.y);
        Fp2 y = p.x.copy();
        y.add(p.x);
        y.add(p.x);
        y.add(a);
        y.mult(t0);
        y.sub(t1);

        return new Point(t2, y);
    }
```

In the final part of the algorithm, the variable $t2$ holds the new x-coordinate and the variable y holds the y-coordinate. It is important to note, that both coordinates are numbers on the complex plane in the finite field $\mathbb{F}_{p^2}$.

7.4.2 Repeated Montgomery point doubling method

```
public Point xDBLe(Point p, Fp2 a, Fp2 b, BigInt e){

    Point p2 = p.copy();

    for (BigInt i = Engine.ZERO.copy(); !i.equals(e); i.add(Engine.ONE
        )) {
        p2 = xDBL(p2, a, b);
    }
    return p2;
}
```

The repeated affine Montgomery point doubling method consecutively executes the *xDBL* function until the function parameter e reaches zero. The result of the *xDBL* function is used as the starting value for the next iteration. When the parameter e is big enough and one iteration reaches a point with a small order, than there is the possibility, that the new point reaches infinity. This is almost non-existent in the SIDH-Algorithm, because the finite field in the complex plane is accordingly big.

7.4.3 Implementation of the Montgomery point addition method

The following code describes the Montgomery point addition(4.5.1) as explained in the math section. The focus lies on a fast addition whilst keeping slim on the resources.

```
public Point xADD(Point p, Point q, Fp2 a, Fp2 b){
    Fp2 minusQY = q.y.copy();
    minusQY.negate();

    if(p.isInfinite()){
        return q;
    }
    else if(q.isInfinite()){
        return p;
    }
    else if(p.equals(q)){
        return xDBL(p, a, b);
    }
    else if(p.y.equals(minusQY)){
        return Engine.INFINITE;
    }
```

Too keep the error rate down and optimize the execution time, some efficency checks need to be done. In the first part of the addition algorithm the function returns P if Q is equal to the point at infinity($\mathcal{O}$) or returns Q if P is the equal to the point at infinity. When both points P and Q are equal, it is better to execute the Montgomery doubling method(7.4.1) for increased performance and better execution times. If Q is the additive complex inverse of P in $\mathbb{F}_{p^2}$ the functions returns the point at infinity($\mathcal{O}$).

```
        Fp2 t0 = q.y.copy();
        t0.sub(p.y);
        Fp2 t1 = q.x.copy();
        t1.sub(p.x);
        t1.multiplicativeInverse();
        t0.mult(t1);
        t1.assign(t0.copy());
        t1.square();
        Fp2 t2 = p.x.copy();
        t2.add(p.x);
        t2.add(q.x);
        t2.add(a);
        t2.mult(t0);
        t0.mult(t1);
        t0.mult(b);
        t0.add(p.y);
        t0.negate();
        t0.add(t2);
        t1.mult(b);
        t1.sub(a);
        t1.sub(p.x);
        t1.sub(q.x);

        return new Point(t1,t0);
    }
```

The *xADD* function returns a new point with the added x and y values, which are themselves complex finite field numbers.

The speed of point addition is mainly affected by the speed of the multiplicative inverse and multiplication in $\mathbb{F}_{p^2}$, which is based on the speed of all underlying multiplication functions. The BigInt-Library tries to keep all assignments to a minimum to increase speed and performance.[39]

7.4.4 Implementation of the Montgomery point tripling method

```
    public Point xTPL(Point p, Fp2 a, Fp2 b){

        Point p2 = xDBL(p,a,b);
        p2 = xADD(p2, p, a, b);

        return p2;
    }
```

The tripling method first doubles a point with the Montgomery point doubling method and then adds the base point to the resulting value. This method is very comparable to the *double-and-add* function, but does not serve the Montgomery ladder functionality.

7.4.5 Repeated Montgomery point tripling method

The following function repeatedly executes Montgomery point tripling method.

```
    public Point xTPLe(Point p, Fp2 a, Fp2 b, BigInt e){

        Point p2 = p.copy();

        for (BigInt i = Engine.ZERO.copy(); !i.equals(e); i.add(Engine.ONE
            )) {
            p2 = xTPL(p2, a, b);
        }
        return p2;
    }
```

7.4.6 Implementation of the Montgomery double-and-add method

The Montgomery double-and-add method compares relatively closely to the Montgomery ladder[72], sine it also computes a scalar multiplication for a given bit length of an Integer value. The double-and-add algorithm, which is also comparable to a square-and-multiply operation, provides a similar problem like the discrete logarithm is in the multiplicative group. This problem can be achieved via the double-and-add method or the Montgomery ladder. Although they are similar, there is a difference in security and the possibility of performance optimizations.[72][73]

```
    public Point double_and_add(BigInt mc, Point p, Fp2 a, Fp2 b){
        BigInt m = mc.copy();
        Point q = Engine.INFINITE.copy();
        String binary = m.toString();
        BigInteger b1 = new BigInteger(binary);
        String binary2 = b1.toString(2);
        int length = binary2.length();
```

In the first part of the algorithm, the scalar is being transformed into binary representation to account for the loop in the second part of the algorithm, because an addition only happens if a bit in the scalar is set to true. For the transformation the Java class BigInteger is used. This is needed because the third party Huldra-Library does not support this direct conversion yet. For a real world application this needs to be exchanged with the correct BigInt format to remove dependencies to internal Java libraries. This is absolutely needed for an application in Java smart cards or any other comparable cryptographic key system.

```
        for (int i = length-1; i>=0; i--) {
            q = xDBL(q, a, b);
            if(m.testBit(i)){
                q = xADD(q, p, a, b);
            }
        }
        return q;
    }
```

The last part is the classical double-and-add method. The loop is dependent on the bit value of the variable m, which means, that each iteration does not complete in constant time. This is a security consideration and it brings up the possibility to utilize a modified Montgomery ladder to tackle this problem and calculate the algorithm in constant time with slightly less performance. This seems like a good trade-off, but also the constant Montgomery ladder is vulnerable to low level hardware attacks that read the calculation timings.

7.4.7 Implementation of the j-Invariant of a Montgomery curve

The j-invariant, is besides the isogenic calculations, the most important function of the SIDH-Algorithm. It is not overly complex, but it calculates a specific curve descriptor, which is used as the secret value in the key exchange from SIDH and the key encapsulation from SIKE.[39]
The Montgomery curve $\mathcal{E}$ is defined by the curve coefficients A and B. The most important value for the Montgomery curve is A which completely determines its geometry. The following equation shows the j-invariant for the Montgomery curve. An important fact is the absence of B in the equation. B can be described as twisting factor and gives certain freedom to construct important points for different cryptographic applications.[74]

$$j(\mathcal{E}_{A,B}) = \frac{256(A^2-3)^3}{A^2-4}$$

The coded algorithm uses speed and memory optimization, thus the operations are different than the mathematical notation.

```
public Fp2 j_inv(Fp2 a){

     Fp2 t0 = a.copy();
     t0.square();

     Fp2 j = new Fp2("3", "0");
     j.negate();
     j.add(t0);

     Fp2 t1 = j.copy();
     t1.square();
     j.mult(t1);

     j.add(j);
     j.add(j);
     j.add(j);
     j.add(j);

     j.add(j);
     j.add(j);
     j.add(j);
     j.add(j);
```

It is more efficient to use additions instead of multiplications for a small multiplicand. Since j is dependent on the size of the curve, its size varies for different finite fields in which the curves reside. For a small curve a multiplication would be fine, but with a finite field hundreds of bits long, this situation is very different.

```
        t1 = new Fp2("4", "0");
        t0.sub(t1);
        t0.multiplicativeInverse();
        j.mult(t0);

        return j;
    }
```

The last part of the j-invariant function is implemented with the modified algorithm by utilizing a multiplicative inverse for faster computational speed. Although the general idea is to minimize the usage of multiplicative inverses, because its computation in $\mathbb{F}_{p^2}$ is quite resource intensive. This is not the same with the j-invariant function because $t0$, which is used in the multiplicative inverse algorithm is very small and only holds the curve coefficient A^2. This value is always the same on each curve and that makes the value predictably small. In the end the function returns the calculated j-invariant value.

7.4.8 Implementation of the 4-isogeny curve coefficient algorithm

The 4-isogeny function takes a point P and the curve twisting coefficient B as parameters and calculates new A' and B' corresponding to $\mathcal{E}_{A',B'} = \mathcal{E}_{A,B}/\langle P_4 \rangle$. The input point has to have the exact order(4.5.1) of 4 on the base curve $\mathcal{E}$. This is why the algorithm is called 4-isogeny.
The isogeny $\phi_4 = \mathcal{E}_{A,B} \to \mathcal{E}_{A',B'}$ is the unique 4-isogeny up to isomorphism with kernel $\langle (x_4, y_4) \rangle$. The new coefficients A' and B' can be computed as follows:

$$(A', B') = (4x_4^4 - 2, -x_4(x_4^2 + 1) * \frac{B}{2})$$

```
public Fp2[] curve_4_iso(Point p, Fp2 b){

    Fp2 t1 = p.x.copy();
    t1.square();

    Fp2 a2 = t1.copy();
    a2.square();
    a2.add(a2);
    a2.add(a2);
    Fp2 t2 = new Fp2("2", "0");
    a2.sub(t2);
    t1.mult(p.x);
    t1.add(p.x);
    t1.mult(b);
    t2.multiplicativeInverse();
    t2.negate();
    t2.mult(t1);

    return new Fp2[]{a2,t2};
}
```

7.4.9 Implementation of the 4-isogeny curve point algorithm

The second 4-isogeny function is the most complex and computation heavy algorithm implementation in the whole SIKE cryptosystem. It takes two points P and Q as parameters, where P is any point on the base curve $\mathcal{E}_{A,B}$ that is not in kernel ϕ_4. This means that the isogeny $\phi_4 = (x_P, y_P) \to (x_{\phi_4(P)}, y_{\phi_4(P)})$.
The algorithms for $x_{\phi_4(P)}$ and $y_{\phi_4(P)}$ is the following:

$$x_{\phi_4(P)} = \frac{-x_P(x_Px_4^2 + x_P - 2x_4)(x_Px_4 - 1)^2}{(2x_Px_4 - x_4^2 - 1)(x_P - x_4)^2}$$

$$y_{\phi_4(P)} = y_P * \frac{-2x_4^2(x_Px_4 - 1)(x_P^4(x_4^2 + 1) - (4x_P^3(x_4^3 + x_4) + 2x_P^2(x_4^4 + 5x_4^2) - 4x_P(x_4^3 + x_4) + x_4^2 + 1)}{(x_P - x_4)^3(2x_Px_4 - x_4^2 - 1)}$$

The equations for $x_{\phi_4(P)}$ and $y_{\phi_4(P)}$ can be optimized programmatically with the use of an multiplicative inverse and allocation optimizations.

```
    public Point eval_4_iso(Point p, Point q){

        Fp2 t1 = q.x.copy();
        t1.square();
        Fp2 t2 = t1.copy();
        t2.square();
        Fp2 t3 = p.x.copy();
        t3.square();
        Fp2 t4 = t2.copy();
        t4.mult(t3);
        t2.add(t4);
        t4.assign(t1.copy());
        t4.mult(t3);
        t4.add(t4);
        Fp2 t5 = t4.copy();
        t5.add(t5);
        t4.add(t5);
        t2.add(t4);
        t4.assign(t3.copy());
        t4.mult(t3);
        t5.assign(t1);
        t5.mult(t4);
        t5.add(t5);
        t2.add(t5);
        t1.mult(q.x);

        t4.assign(p.x.copy());
        t4.mult(t3);
        t5.assign(t1.copy());
        t5.mult(t4);
        t5.add(t5);
        t5.add(t5);
        t2.sub(t5);
        t1.mult(p.x);
        t1.add(t1);
        t1.add(t1);
        t1.negate();
        t1.add(t2);
        t2.assign(q.x.copy());
        t2.mult(t4);
        t2.add(t2);
        t2.add(t2);
        t1.sub(t2);
        t1.add(t3);
        t1.add(Engine.FP2ONE);
        t2.assign(q.x.copy());
        t2.mult(p.x);

        t4.assign(t2.copy());
        t4.sub(Engine.FP2ONE);
        t2.add(t2);
        t5.assign(t2.copy());
        t5.add(t5);
        t1.sub(t5);
        t1.mult(t4);
        t1.mult(t3);
        t1.mult(q.y);
        t1.add(t1);
        Fp2 ynew = t1.copy();
        ynew.negate();
        t2.sub(t3);
        t1.assign(t2.copy());
        t1.sub(Engine.FP2ONE);
        t2.assign(q.x.copy());
        t2.sub(p.x);
        t1.mult(t2);
        t5.assign(t1.copy());
        t5.square();
        t5.mult(t2);
        t5.multiplicativeInverse();
        ynew.mult(t5);
        t1.mult(t2);
        t1.multiplicativeInverse();
        t4.square();
        t1.mult(t4);
        t1.mult(q.x);
        t2.assign(q.x.copy());
        t2.mult(t3);
        t2.add(q.x);
        t3.assign(p.x.copy());
        t3.add(p.x);
        t2.sub(t3);
        t2.negate();
        t1.mult(t2);
        return new Point(t1, ynew);
    }
```

The function returns a new point that lies on the curve $\mathcal{E}_{A',B'}$. The morphism from $\mathcal{E}_{A,B}$ to $\mathcal{E}_{A',B'}$ is described by $\phi_4(P)$.

7.4.10 Implementation of the 3-isogeny curve coefficient algorithm

The 3-isogeny curve coefficient algorithm also calculates a new A and B corresponding to an input point. But unlike the 4-isogeny algorithm it takes both parameters A and B from a curve. When $P \in \mathcal{E}_{A,B}$ with exact order of 3, then $\phi_3 = (x_P, y_P) \to (x_{\phi_3(P)}, y_{\phi_3(P)})$ describes the isogeny from $\mathcal{E}_{A,B}$ to $\mathcal{E}_{A',B'}$. The function can be computed as follow:

$$(A', B') = ((Ax_3 - 6x_3^2 + 6)x_3, Bx_3^2)$$

It is important to note, that the new coefficient A' only depends on A and x_P

```
public Fp2[] curve_3_iso(Point p, Fp2 a, Fp2 b){

    Fp2 t1 = p.x.copy();
    t1.square();
    Fp2 b2 = t1.copy();
    b2.mult(b.copy());

    t1.add(t1);
    Fp2 t2 = t1.copy();
    t2.add(t2);
    t1.add(t2);
    t2.assign(new Fp2("6","0"));
    t1.sub(t2);
    t2.assign(a.copy());
    t2.mult(p.x);
    t1.negate();
    t1.add(t2);
    t1.mult(p.x);
    return new Fp2[]{t1, b2};
}
```

7.4.11 Implementation of the 3-isogeny curve point algorithm

The 3-isogeny curve point algorithm takes two points P and Q and constructs an isogeny to a new point on a new isogenous elliptic curve. The point Q needs to be of exact order of 3 on $\mathcal{E}_{A,B}$. P is any point and not in kernel ϕ_3. If that is the case, then $\phi_3 = (x_P, y_P) \to (x_{\phi_3(P)}, y_{\phi_3(P)})$. The new 3-isogenous point can be calculated as follows:

$$x_{\phi_3(P)} = \frac{x_P(x_P x_3 - 1)^2}{(x_P - x_3)^2}$$

$$y_{\phi_3(P)} = y_P * \frac{(x_P x_3 - 1)^2(x_P^2 x_3 - 3x_P x_3^2 + x_P + x_3)}{(x_P - x_3)^2}$$

```java
public Point eval_3_iso(Point p, Point q){
    Fp2 t1 = q.x.copy();
    t1.square();
    t1.mult(p.x);
    Fp2 t2 = p.x.copy();
    t2.square();
    t2.mult(q.x);
    Fp2 t3 = t2.copy();
    t3.add(t3);
    t2.add(t3);
    t1.sub(t2);
    t1.add(q.x);
    t2.assign(q.x.copy());
    t2.sub(p.x);
    t2.multiplicativeInverse();
    t3.assign(t2.copy());
    t3.square();
    t2.mult(t3);
    Fp2 t4 = q.x.copy();
    t4.mult(p.x);
    t4.sub(Engine.FP2ONE);
    t1.mult(t4);
    t1.mult(t2);
    t2.assign(t4.copy());
    t2.square();
    t2.mult(t3);
    t2.mult(q.x);
    t1.mult(q.y);
    return new Point(t2, t1);
}
```

7.4.12 Computation and evaluation of a 2^e-isogeny

The 2^e-isogeny function serves a special function to generate isogenies from points with a certain kernel. This function takes both A and B from the curve $\mathcal{E}_{A,B}$ and a point S with exact order of 2^e on $\mathcal{E}_{A,B}$. The parameter e is equal to e_2 from the public parameters of the SIKE algorithm and is used to describe the primary base field.
This function automatically traverses multiple supersingular elliptic curves, until it reaches the correct curve within the same kernel.

```
public IsoReturn iso_2_e(Point s, Fp2 a, Fp2 b, Point[] p){

    Fp2 a2 = a.copy();
    Fp2 b2 = b.copy();
    BigInt e = Engine.E2.copy();
    e.sub(Engine.TWO);
    Point r;

    for (; !e.equals(Engine.MINUSTWO); e.sub(Engine.TWO)) {
        r = xDBLe(s, a2, b2, e);
        Fp2 ab[] = curve_4_iso(r, b2);
        a2.assign(ab[0]);
        b2.assign(ab[1]);
        s = eval_4_iso(s, r);
        for (int i = 0; i < p.length; i++) {
            p[i] = eval_4_iso(p[i], r);
        }
    }
    return new IsoReturn(a, b, p);
}
```

The function returns an isogeny with the new coefficients A' and B' on the supersingular elliptic curve $\mathcal{E}_{A',B'} = \mathcal{E} \setminus \langle S \rangle$ and a new isogenic point on that curve.

7.4.13 Computation and evaluation of a 3^e-isogeny

The 3^e-isogeny computation uses the public parameter e_3 as the exponential and the point S with the order of 3^e_3 on the base curve $\mathcal{E}_{A,B}$. Besides this difference and the usage of the 4-isogeny functions, this algorithm does the same as 7.4.12.

```
    public IsoReturn iso_3_e(Point s, Fp2 a, Fp2 b, Point[] p){

        Fp2 a2 = a.copy();
        Fp2 b2 = b.copy();
        BigInt e = Engine.E3.copy();
        e.sub(Engine.ONE);
        Point r;

        for (; !e.equals(Engine.MINUSONE); e.sub(Engine.ONE)) {
            r = xDBLe(s, a2, b2, e);
            Fp2 ab[] = curve_3_iso(r, a2, b2);
            a2.assign(ab[0]);
            b2.assign(ab[1]);
            s = eval_3_iso(s, r);
            for (int i = 0; i < p.length; i++) {
                p[i] = eval_3_iso(p[i], r);
            }
        }
        return new IsoReturn(a, b, p);
    }
```

7.4.14 Implementation of the x-Coordinate recovery algorithm

The public generator points are only defined for P and Q. The x_R coordinate of the third point R can be recovered from P and Q with the help of the two coefficients A and B. This can be used to recover different x_R-coordinates for the point R depending on the curve and isogeny the point is needed in. x_R is calculated so that $R = P - Q$.

```
    public Point get_xR(Fp2 a, Fp2 b, Point p, Point q){

        Point q2 = q.copy();
        q2.y.negate();

        Point r = xADD(p, q2, a, b);

        return r;
    }
```

Q is negated and then added to P which is equal to $P - Q$. Although both, x_R and y_R are existing in R only x_R is needed in SIKE.

7.4.15 Implementation of the y-Coordinate recovery algorithm

A public key essentially holds the x-coordinates of the three points P, Q and R. With the following algorithm it is possible to recover A and B from an elliptic curve $\mathcal{E}$. This public key is based upon and the y-coordinates from P and Q. After that, the program knows the full parameters of the curve $\mathcal{E}_{A,B}$ and the points P and Q. This is an essential function and utilized for further calculations when the Public-Key is being transferred between participants.[39]

```
    public Fp2[] get_yP_yQ_A_B(PublicKey pk){
        Fp2 xp = pk.getXp().copy();
        Fp2 xq = pk.getXq().copy();
        Fp2 xr = pk.getXr().copy();

        Fp2 a = xp.copy();

        a.mult(xq);
        a.mult(xr);
        a.add(a);
        a.add(a);
        a.multiplicativeInverse();

        Fp2 t1 = xp.copy();
        Fp2 t2 = Engine.FP2ONE.copy();
        t1.mult(xr);
        t2.sub(t1);

        t1 = xq.copy();
        t1.mult(xr);
        t2.sub(t1);
        t2.square();

        a.mult(t2);
        a.sub(xp);
        a.sub(xq);
        a.sub(xr);

        t1 = xp.copy();
        t1.square();
        t2 = t1.copy();
        t2.mult(xp);
        t1.mult(a);
        t1.add(t2);
        t1.add(xp);
        t1.sqrt();
        Fp2 yp = t1.copy();
        t1 = xq.copy();
        t1.square();
        t2 = t1.copy();
        t2.mult(xq);
        t1.mult(a);
        t1.add(t2);
        t1.add(xq);
        t1.sqrt();
        Fp2 yq = t1.copy();
        t1.negate();

        Point p = new Point(xp, yp);
        Point q = new Point(xq, t1);
        p = xADD(p, q, a, Engine.B);
```

Both P and Q are being populated with the values from the y-coordinate recovery algorithm.[39]

```
        if(!p.x.equals(xr)){
            return new Fp2[]{yp, t1, a, Engine.B.copy()};
        }
        return new Fp2[]{yp, yq, a, Engine.B.copy()};
    }
```

The function returns an array consisting of y_P, y_Q, A and B from $\mathcal{E}$.

7.4.16 Implementation of the 2^e-isogeny secret key generation

```
    public Fp sk_keygen_2(){
        BigInteger pow = new BigInteger("2").pow(Integer.parseInt(Engine.
           E2.toString()));
        pow = pow.subtract(BigInteger.ONE);
        SecureRandom rnd = new SecureRandom();
        int numBits = pow.bitLength();
        BigInteger randval;
        do {
            randval = new BigInteger(numBits, rnd);
        }while(randval.bitLength() != numBits);

        return new Fp(randval.toString());
    }
```

The 2^e-isogeny secret key generation outputs a random value according to the bit-length of the boundaries $[0, 2^{e_2} - 1]$.[39]

7.4.17 Implementation of the 3^e-isogeny secret key generation

```
    public Fp sk_keygen_3(){

        BigInteger pow = new BigInteger("3").pow(Integer.parseInt(Engine.
           E3.toString()));
        int bitLength = pow.bitLength();
        pow = new BigInteger("2").pow(bitLength);
        pow = pow.subtract(BigInteger.ONE);
        SecureRandom rnd = new SecureRandom();
        int numBits = pow.bitLength();

        BigInteger randval;
        do {
            randval =  new BigInteger(numBits, rnd);
        }while(randval.bitLength() != numBits);

        return new Fp(randval.toString());
    }
}
```

The 3^e-isogeny secret key generation outputs a random value according to the bit-length of the boundaries $[0, 2^{(log_2(3_3^e)-1)}]$.

7.4.18 Implementation of the 2^e-isogeny public key generation

The 2^e-isogeny public key from the SIKE algorithm[39] is calculated with the following parameters:

- The prime $p = 2^{e_2}3^{e_3} - 1$
- The curve $\mathcal{E}_0 \in \mathbb{F}_{p^2}$
- Public generator points P_2 and Q_2 with the coordinates x_{P_2} and x_{Q_2}
- Recovered x-coordinate of R with x_{R_2}
- The secret key sk_2

With these parameters set in place, it is possible to calculate the isogenic key pair consisting of the secret key sk_2, which is a BigInt value and the public key pk_2 consisting of the x-coordinates $x_{P_2}, x_{Q_2}, x_{R_2}$

The computation follows this algorithm:

- Set a new point x_S to x_{P_2}
- Traverse the Montgomery ladder with Q_2 until the secret key sk_2 is satisfied.
- Add ladder result to x_S so that $x_S = x_{P_2} + [sk_2]Q_2$
- Define i and iterate from 0 to $e_2 - 1$ for the next step
- Compute x-coordinate of the 2-isogeny so that $\phi_i = \mathcal{E}_i \rightarrow \mathcal{E}'$ and $(x) \rightarrow (f_i(x))$
- x_S-coordinate from the point S on the curve $\mathcal{E}_i$ should have kernel $\langle [2^{e_2-i-1}]S \rangle$ isogeny ϕ_i
- Set x_S so that $x_S = f_i(x_S)$ using the isogeny ϕ_i and curve $\mathcal{E}'$
- Recover x_R with A and B from $\mathcal{E}'$ and two points from the isogeny ϕ_i
- Set (x_P, x_Q, x_R) to $(f_i(x_{P_2}), f_i(x_{Q_2}), f_i(x_{R_2})$
- Return the public key $pk_2(x_P, x_Q, x_R)$

```java
public PublicKey isogen_2(BigInt sk){
    Fp2 a = Engine.FP2ZERO.copy();
    Fp2 b = Engine.FP2ONE.copy();
    Point s = double_and_add(sk, Engine.Q2.copy(), a, b);
    s = xADD(Engine.P2.copy(), s, a, b);
    IsoReturn ir = iso_2_e(s, a, b, new Point[]{Engine.P3.copy(),
        Engine.Q3.copy()});
    Point[] points = ir.getP();
    Point r = get_xR(a, b, points[0], points[1]);
    return new PublicKey(points[0].x, points[1].x, r.x);
}
```

7.4.19 Implementation of the 3^e-isogeny public key generation

The 3^e-isogeny public key follows the same principle as the 3^e-isogeny public key(7.4.18) with different parameters and exponents[39]:

- The prime $p = 2^{e_2}3^{e_3} - 1$
- The curve $\mathcal{E}_0 \in \mathbb{F}_{p^2}$
- Public generator points P_3 and Q_3 with the coordinates x_{P_3} and x_{Q_3}
- Recovered x-coordinate of R with x_{R_3}
- The secret key sk_3

With these parameters set in place, it is possible to calculate the isogenic key pair consisting of the secret key sk_3 which is a BigInt value and the public key pk_3 consisting of the x-coordinates $x_{P_3}, x_{Q_3}, x_{R_3}$

The computation follows this algorithm:

- Set a new point x_S to x_{P_3}
- Traverse the Montgomery ladder with Q_3 until the secret key sk_3 is satisfied.
- Add ladder result to x_S so that $x_S = x_{P_3} + [sk_3]Q_3$
- Define i and iterate from 0 to $e_3 - 1$ for the next step
- Compute x-coordinate of the 3-isogeny so that $\phi_i = \mathcal{E}_i \to \mathcal{E}'$ and $(x) \to (f_i(x))$
- x_S-coordinate from the point S on the curve $\mathcal{E}_i$ should have kernel $\langle[3^{e_3-i-1}]S\rangle$ isogeny ϕ_i
- Set x_S so that $x_S = f_i(x_S)$ using the isogeny ϕ_i and curve $\mathcal{E}'$
- Recover x_R with A and B from $\mathcal{E}'$ and two points from the isogeny ϕ_i
- Set (x_P, x_Q, x_R) to $(f_i(x_{P_3}), f_i(x_{Q_3}), f_i(x_{R_3})$
- Return the public key $pk_3(x_P, x_Q, x_R)$

```java
public PublicKey isogen_3(BigInt sk){
    Fp2 a = Engine.FP2ZERO.copy();
    Fp2 b = Engine.FP2ONE.copy();
    Point s = double_and_add(sk, Engine.Q3.copy(), a, b);
    s = xADD(Engine.P3.copy(), s, a, b);
    IsoReturn ir = iso_3_e(s, a, b, new Point[]{Engine.P2.copy(),
       Engine.Q2.copy()});
    Point[] points = ir.getP();
    Point r = get_xR(a, b, points[0], points[1]);
    return new PublicKey(points[0].x, points[1].x, r.x);
}
```

7.4.20 Implementation of the 2^e-isogeny key exchange

The 2^e-isogeny key exchange takes the prime number $p = 2^{e_2}3^{e_3} - 1$, the secret key sk_2 and the public key pk_3 and calculates a shared secret j with the j-invariant function as described in 7.4.7. The shared secret should be the same for both SIKE participants, given that no errors happened during the transmission or the calculations.[39]

The following algorithm handles the key exchange and computes the j-invariant:

- Calculate the base curve E'_0 from the public key pk_3
- Define a new point S and set x_s to x_{P_2}
- Traverse the Montgomery ladder with Q_2 until the secret key sk_2 is satisfied.
- Add the result of the ladder to S so that $x_S = x_{P_2} + [sk_2]Q_2$
- Define i and iterate from 0 to $e_2 - 1$ for the next step
- Compute x-coordinate of the 2-isogeny so that $\phi_i = \mathcal{E}'_i \to \mathcal{E}'$ and $(x) \to (f_i(x))$
- x_S-coordinate from the point S on the curve $\mathcal{E}'_i$ should have kernel $\langle[2^{e_2-i-1}]S\rangle$ isogeny ϕ_i
- Set x_S so that $x_S = f_i(x_S)$ using the isogeny ϕ_i and curve $\mathcal{E}'_i$
- Return the j-invariant of the curve $\mathcal{E}'_{e_2}$

```java
public Fp2 isoex_2(BigInt sk, PublicKey pk){
    Fp2[] t = get_yP_yQ_A_B(pk);
    Point q = new Point(pk.getXq().copy(), t[1]);
    Point p = new Point(pk.getXp(), t[0]);
    Fp2 a = t[2];
    Fp2 b = t[3];
    Point s = double_and_add(sk, q, a, b);
    s = xADD(p, s, a, b);
    IsoReturn is = iso_2_e(s, a, b, new Point[]{});
    Fp2 j = j_inv(is.getA());
    return j;
}
```

7.4.21 Implementation of the 3^e-isogeny key exchange

The 3^e-isogeny key exchange is very similar to the 2^e-isogeny version. The main difference is, that the 3^e-isogeny is faster to calculate and is overall quicker performance wise. Despite this differences, both versions are needed to compute SIKE.[39]

This key exchange takes the prime number $p = 2^{e_2}3^{e_3} - 1$, the secret key sk_3 and the public key pk_2 and calculates a shared secret j with the j-invariant function as described in 7.4.7. The shared secret should be the same for both SIKE participants, given that no errors happened during the transmission or the calculations. This means that both, the j-invariant of the 2^e-isogeny and the j-invariant of this function is the same after a correct key exchange. This is the case when the 2^e-isogeny key exchange calculates its j-invariant with the secret key sk_2 and and the public key pk_3. The other participant needs to calculate the 3^e-isogeny key exchange with the secret key sk_3 and and the public key pk_2.[39]

The following algorithm handles the key exchange and computes the j-invariant:

- Calculate the base curve E_0' from the public key pk_2
- Define a new point S and set x_s to x_{P_3}
- Traverse the Montgomery ladder with Q_3 until the secret key sk_3 is satisfied.
- Add the result of the ladder to S so that $x_S = x_{P_3} + [sk_3]Q_3$
- Define i and iterate from 0 to $e_3 - 1$ for the next step
- Compute x-coordinate of the 3-isogeny so that $\phi_i = \mathcal{E}_i' \rightarrow \mathcal{E}'$ and $(x) \rightarrow (f_i(x))$
- x_S-coordinate from the point S on the curve $\mathcal{E}_i'$ should have kernel $\langle [3^{e_3-i-1}]S \rangle$ isogeny ϕ_i
- Set x_S so that $x_S = f_i(x_S)$ using the isogeny ϕ_i and curve $\mathcal{E}_i'$
- Return the j-invariant of the curve $\mathcal{E}_{e_3}'$

```
        public Fp2 isoex_3(BigInt sk, PublicKey pk){
        Fp2[] t = get_yP_yQ_A_B(pk);
        Point q = new Point(pk.getXq().copy(), t[1]);
        Point p = new Point(pk.getXp(), t[0]);
        Fp2 a = t[2];
        Fp2 b = t[3];
        Point s = double_and_add(sk, q, a, b);
        s = xADD(p, s, a, b);
        IsoReturn is = iso_3_e(s, a, b, new Point[]{});
        Fp2 j = j_inv(is.getA());
        return j;
    }
```

7.5 Montgomery $\mathbb{F}_{p^2}$ Point class implementation

The Montgomery $\mathbb{F}_{p^2}$ point is as a data-holding class to store complex curve points. This could be inline since the isogenic calculations don't require a lot of point transfer between values, but to keep the coding style according to the Java object orientated workflow, this class is being used.

```java
public class Point{

    public final Fp2 x, y;
    public Fp degree;

    public Point(Fp2 x, Fp2 y) {
        this.x = x;
        this.y = y;
    }

    public Point(String x0, String x1, String y0, String y1) {
        x = new Fp2(new BigInteger(x0, 16).toString(), new BigInteger(x1,
            16).toString());
        y = new Fp2(new BigInteger(y0, 16).toString(), new BigInteger(y1,
            16).toString());
    }
```

Each point consists of an x and y value, which are both values in the complex number field $\mathbb{F}_{p^2}$. This means that each value consists of an real part $\Re$ and an imaginary part $\Im$.
The class constructors serves for $\mathbb{F}_{p^2}$ compatibility and String compatibility to provide easier allocations from within the program.

```java
    @Override
    public String toString() {
       return String.format("(%s, %s)", x.toString(), y.toString());
    }
    public String toDegreeString() {
       if(degree != null){
           return String.format("(%s, %s) --> %s", x.toString(), y.
               toString(), degree.toString());
       }
       return String.format("(%s, %s)", x.toString(), y.toString());
    }
    public void print(){
        System.out.println(toDegreeString());
    }
```

The first output function overrides the basic *toString()* method to provide basic debug and user output. The second function provides information about the order(4.5.1) of the point. The knowledge of the order is especially important to check or prove the correct operation of certain function. For example the isogeny graph operations need points with exact order of 3 or 4 on the elliptic curve. The third function is used to directly print to the user or developer.

```
    public Fp getDegree() {
        return degree;
    }
    public void setDegree(Fp degree) {
        this.degree = degree;
    }
    public Point copy(){
        return new Point(x.copy(), y.copy());
    }
```

The complex point class also has the possibility to save and read the degree value, as well as copy the point into a new variable. The *copy* function is utilized very often, since every calculation on points is based upon mutable mathematical objects. This means that when a mathematical operation needs to be done and the value needs to be saved for concurrent calculations, the point is going to be copied into a new object.

```
    public boolean isInfinite(){
       return x.isZero()  y.isZero();
    }
    public boolean equals(Point p) {
        return p.x.equals(x)  p.y.equals(y);
    }
}
```

The last part of this class focuses on comparative function. The first one checks if a certain point is the point at infinity. This function returns true when both complex coordinates return zero. The second function returns a Boolean value if the location is the same for both points. This means that both, the x and the y coordinate needs to be compared.

$$P.\text{equals}(Q) = \begin{cases} 1, & \text{if } x \text{ and } y \text{ of } P \text{ is greater than } x \text{ and } y \text{ of } Q. \\ 0, & \text{if } x \text{ or } y \text{ of } P \text{ is smaller than } x \text{ or } y \text{ of } Q. \end{cases}$$

7.6 Public-key class implementation

The public-key class holds the Montgomery values that represent a public key. These include of the x-values of P and Q and the x-value of R, which is a special point that has been recovered from the two aforementioned Montgomery points.

```
package org.fh.sidh_new;

public class PublicKey {

    private Fp2 xp, xq, xr;

    public PublicKey(Fp2 xp, Fp2 xq, Fp2 xr) {
        this.xp = xp;
        this.xq = xq;
        this.xr = xr;
    }
```

This class is mainly used for data storage and transfers between certain isogeny functions. Usually all values are immediately set on creation of this object, but there is the possibility that some values need to be exchange while going trough the SIKE algorithm for key encapsulation. This is also dependant on the usage of the keys and the isogenic algorithm used.

```
    public Fp2 getXp() {
        return xp;
    }

    public void setXp(Fp2 xp) {
        this.xp = xp;
    }

    public Fp2 getXq() {
        return xq;
    }

    public void setXq(Fp2 xq) {
        this.xq = xq;
    }

    public Fp2 getXr() {
        return xr;
    }

    public void setXr(Fp2 xr) {
        this.xr = xr;
    }
}
```

The second and final part of this class consists of get- and set-functions for the three Public-key points.

7.7 IsoReturn data-hoalding class implementation

The IsoReturn class holds isogenic values that need to be exchanged with certain functions. This class holds the coefficients of an isogenic curve and 1 to n Montgomery points that lie on this isogenic curve.

```
public class IsoReturn {

    private Fp2 a, b;
    private Point p[];

    public IsoReturn(Fp2 a, Fp2 b, Point[] p) {
        this.a = a;
        this.b = b;
        this.p = p;
    }
```

In the first part of the IsoReturn class, the values a and b, which describe the new isogenic Montogmery curve and a Point array is being allocated. The size of this array depends of the input on the 3-isogeny and 4-isogeny functions.

```
    public Fp2 getA() {
        return a;
    }

    public void setA(Fp2 a) {
        this.a = a;
    }

    public Fp2 getB() {
        return b;
    }

    public void setB(Fp2 b) {
        this.b = b;
    }

    public Point[] getP() {
        return p;
    }

    public void setP(Point[] p) {
        this.p = p;
    }
}
```

The rest of this class are classical get- and set-functions to provide a way of data access and compatibility with different functions. This is primarily utilized to read the Montgomery points stored in the array, because these points are used to advance further in the isogeny graph.

7.8 Main engine implementation of the SIDH/SIKE algorithm

The Engine class implements all previously explained classes and functions and calls everything according to the SIKE algorithm in order to perform isogeny based cryptography. This class initializes and holds all important values that are needed for to calculate the isogenic algorithms correctly.

```
import java.math.BigInteger;
import java.security.SecureRandom;
import com.github.aelstad.keccakj.cipher.CipherInterface;
import com.github.aelstad.keccakj.provider.KeccakjProvider;
import com.github.aelstad.keccakj.spi.Shake256Key;
import com.github.aelstad.keccakj.spi.Shake256StreamCipher;
import java.security.InvalidAlgorithmParameterException;
import java.security.InvalidKeyException;
import java.security.Security;
import java.util.logging.Level;
import java.util.logging.Logger;
import javax.crypto.BadPaddingException;
import javax.crypto.Cipher;
import javax.crypto.IllegalBlockSizeException;
import javax.crypto.spec.IvParameterSpec;

public class Engine {

    public static final BigInt E2 =
    new BigInt(new BigInteger("FA", 16).toString());
    public static final BigInt E3 =
    new BigInt(new BigInteger("9F", 16).toString());
```

e_2 and e_3 describe the finite field $\mathbb{F}_{p^2}$, so that $p = 2^{e_2} * 3^{e_3} - 1$. The values of e_2 and e_3 are given and depend on the bit size and security that is needed in the isogenic graph based algorithm the field $\mathbb{F}_{p^2}$ is used in.

```
    public static final BigInt PRIME =
    new BigInt(new BigInteger("4066F541811E1E6045C6BDDA77A4D01B9BF6C"+
    "87B7E7DAF13085BDA2211E7A0ABFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF"+
    "FFFFFFFFFFFFFFFFFFFFFFFFFF", 16).toString());
```

The next value is the prime number that was described by e_2 and e_3. This number sets the boundaries for the finite field $\mathbb{F}_p$ and $\mathbb{F}_{p^2}$. In the SIKE submission there are three security levels with different bit sizes listed. The first one is 503-bits long which, in terms of quantum cryptography, has the lowest quantum security but also the lowest calculation time. The other two quantum-secure possible values are 751 and 964 bits. This means, that there is a trade-off between security and performance and it heavily depends on the security need and the type of application to correctly decide the bit size. In this implementation of the SIKE algorithm, the 503-bit prime number is used. The algorithm works the same way with every other prime base field, but the calculation time differs. Implementations in smaller devices or even smart cards require a longer thought process and maybe need completely different starting values or a different amount of iterations. This heavily depends on the use case and depends if confidential information needs to be stored short- or long-term.

```
    public static final BigInt MINUSTWO = new BigInt("-2");
    public static final BigInt MINUSONE = new BigInt("-1");
    public static final BigInt ZERO = new BigInt("0");
    public static final BigInt ONE = new BigInt("1");
    public static final BigInt TWO = new BigInt("2");
    public static final BigInt THREE = new BigInt("3");
```

The above static parameters are helper values to calculate the SIKE algorithm with the key exchange and encapsulation. They provide lower calculations time and performance improvements, because the program does not need to reallocate the same values over and over. Instead they are copied or red from already existing memory space depending on the usage.

```
    public static final Fp2 A = new Fp2(ZERO, ZERO);
    public static final Fp2 B = new Fp2(ONE, ZERO);
    public static final Fp2 FP2ONE = new Fp2(ONE, ZERO);
    public static final Fp2 FP2ZERO = new Fp2(ZERO, ZERO);
    public static final Fp FPZERO = new Fp(ZERO);
    public static final Fp FPONE = new Fp(ONE);
    public static final Point INFINITE = new Point(A.copy(), A.copy());
```

The static point allocations above reuse the static parameters and link them via pointers. This saves memory space and allocation time, because the BigInt value-arrays already exist in memory.

```
    public static final Point P2 = new Point(
        "1F6D52A7563BB9356B98A116A0CA9775DBB7382EB29E24E45299D8939959EAE"+
        "EB47FF3113F60882D12103E4B8B8CD2B97DA14657AE8C128BE82209D2DDFCA9",
        "2D44C3FAD24E4CBDDC8A2D9DE336A92A9912EE6D09E2DD5C33AB26D60A268AC"+
        "91F38E1AF4C2D5BFA2B87DD55C8CA6019C6B0C08ED92B5AEB6C65A8E06E53E9",
        "3C9F7C397283C0871F78D9F74ECC0A8F89579CCBEF8FE60D07338AF0A0322E3"+
        "F0C66CA826AA5BF85EB53666C272C8EAEC9B808B3B78E6422330617AC23D6F2",
        "38222AE95DA234ABD1B90FD897C2E2E7995B2C0006DC92CC079B7C60C94DCAE"+
        "9961CC7A4BAEAC9D294F6D5760D4D654821193AE92AD42AC0047ADE55C343FC"
    );
```

The first generator point P_2, which is one of the four starting points is calculated the following way[39]:

- Define boundaries $P_2 \in E_0(\mathbb{F}_{p^2}))$
- Set $x_0 = i + c \in \mathbb{F}_{p^2}$
- Increment c until x_0 is the smallest non-negative integer in $\mathbb{F}_{p^2}$ so that $[2^{e_2-1}]P_2 = (-3 \pm 2\sqrt{2}, 0)$
- Calculate $y_0 = \sqrt{f(i+c)}$
- Calculate $(x_{P_2}, y_{P_2}) = [3^{e_3}](x_0, y_0)$
- The point P_2 has exact order of 2^{e_2} and forms a basis for $E_0(\mathbb{F}_{p^2}[2^{e_2}])$

```
    public static final Point Q2 = new Point(
        "21B7098B640A01D88708B729837E870CFF9DF6D4DF86D86A7409F41156CB5F7"+
        "B8514822730940C9B51E0D9821B0A67DD7ED98B9793685FA2E22D6D89D66A4E",
        "2F37F575BEBBC33851F75B7AB5D89FC3F07E4DF3CC52349804B8D17A17000A4"+
        "2FC6C5734B9FCFDE669730F3E8569CEB53821D3E8012F7F391F57364F402909",
        "78F8A30AB36B301BDF672D9E3518AF741F8227CC95A9F351B99623A826DE3F"+
        "8D90DD6ED42FF298E394E77B7AEFEE6010CDF34A7DE9F9E239B103E7B3EEE",
        "37F3C600488EBB6B11462C4CAFC41CD5DC611A9B0C804E3BF50D6D8F75C4E7A"+
        "136E29E00D80EB8653CA830F2AED61D04F9F3A8317F7916E016F2733B828AC0"
    );
```

The second generator point Q_2 is calculated the following way[39]:

- Define boundaries $Q_2 \in E_0(\mathbb{F}_{p^2})$
- Set $x_0 = i + c \in \mathbb{F}_{p^2}$
- Increment c until x_0 is the smallest non-negative integer in $\mathbb{F}_{p^2}$ so that $[2^{e_2-1}]Q_2 = (0, 0)$
- Calculate $y_0 = \sqrt{f(i+c)}$
- Calculate $(x_{Q_2}, y_{Q_2}) = [3^{e_3}](x_0, y_0)$
- The point Q_2 has exact order of 2^{e_2} and forms a basis for $E_0(\mathbb{F}_{p^2}[2^{e_2}])$

```
public static final Point P3 = new Point(
    "97453912E12F3DAF32EEFFD618BD93D3BBBF399137BD39858CADEFAE382E42D"+
    "6E60A62FD62417AD61A14B60DB26125273EC980981325D86E55C45E3BB46B1",
    "0",
    "9B66640A4CC79F82B68D72609233812DF76E8B0422EF3527A1F2A9915EFF16E"+
    "0940040DF4A15A84A5ACF024FC2ED8A50102A731E8D20D033B48035B63DD62",
    "0"
);
```

The third generator point P_3 is the first point to have a zero value in the imaginary axis and is calculated the following way[39]:

- Define boundaries $P_3 \in E_0(\mathbb{F}_{p^2}) \setminus E_0(\mathbb{F}_p)$
- Set $x_0 = c \in \mathbb{F}_p$
- Increment c until $f(c)$ is the smallest non-negative square in $\mathbb{F}_p$
- Calculate $y_0 = \sqrt{f(c)}$
- Calculate $(x_{P_3}, y_{P_3}) = [2^{e_2 - 1}](x_0, y_0)$
- The point P_3 has exact order of 3^{e_3} and forms a basis for $E_0(\mathbb{F}_{p^2}[3^{e_3}])$

```
public static final Point Q3 = new Point(
    "1E7D6EBCEEC9CFC47779AFFD696A88A971CDF3EC61E009DF55CAF4B6E01903B"+
    "2CD1A12089C2ECE106BDF745894C14D7E39B6997F70023E0A23B4B3787EF08F",
    "0",
    "2EC0AAEF9FBBDD75FBDA11DA19725F79E842FBC355071FD631C1CDF90E08E60"+
    "1929FAEC5DAEB0D96BBB4AD50FC7C8AD47064F05C06DC5D4AAE61CCCEFF1F26",
    "0"
);
```

The fourth and last generator point Q_3 is the second point to have a zero value in the imaginary axis and is calculated the following way[39]:

- Define boundaries $Q_3 \in E_0(\mathbb{F}_p)$
- Set $x_0 = c \in \mathbb{F}_p$
- Increment c until $f(c)$ is the smallest non-negative non-square in $\mathbb{F}_p$
- Calculate $y_0 = \sqrt{f(c)}$
- Calculate $(x_{Q_3}, y_{Q_3}) = [2^{e_2 - 1}](x_0, y_0)$
- The point Q_3 has exact order of 3^{e_3} and forms a basis for $E_0(\mathbb{F}_{p^2}[3^{e_3}])$

The *genRandom* function is a pseudo-random number generator that uses the Java BigInteger class to generate a random value. This random value is used to randomize the key exchange and generate random values for the quadratic non-residue check. The main problematic with this function is, that it utilizes the BigInteger class and this is an unwanted dependency. This is why this function needs to be exchanged with random value function existing on the platform this SIKE algorithm is used on. For example a Java smart card needs to utilize random functions, that are available on the hardware itself and remove this dependency. For legacy and testing reasons this function is included here since it serves an important functionality for the whole program.
The function itself generates random values up to the same bit length as the boundary, which is the prime number, until the value generated is strictly less than the boundary value. An average expected number of calls to the BigInteger class, which always allocates a new object is less than two. This will always be faster with hardware implementations of random number generators.

```java
public static BigInt genRandom() {
    BigInteger bound = new BigInteger(PRIME.toString());
    SecureRandom rnd = new SecureRandom();
    int numBits = bound.bitLength();

    BigInteger randval = new BigInteger(numBits, rnd);

    while (randval.compareTo(bound) >= 0) {
        randval = new BigInteger(numBits, rnd);
    }
    return new BigInt(randval.toString());
}
```

7.8.1 Main SIDH-Algorithm

The next part of the main Engine class consists of server function that execute all functions to fully compute the SIDH-Algorithm, a public key encryption with isogenies and the key exchange mechanism.

```
    public Engine(){
        MathR mr = new MathR();

        Fp sk2 = mr.sk_keygen_2();

        Fp sk3 = mr.sk_keygen_3();


        PublicKey pk2 = mr.isogen_2(sk2.getValue());
        PublicKey pk3 = mr.isogen_3(sk3.getValue());

        Fp2 j_invariant2 = mr.isoex_2(sk2.getValue(), pk3);
        Fp2 j_invariant3 = mr.isoex_3(sk3.getValue(), pk2);

        if(j_invariant2.equals(j_invariant3)){
            System.out.println("SIDH working correctly");
        }else{
            System.err.println("SIDH not working correctly");
        }
    }
}
```

The main SIDH-Algorithm executes the keygen functions *sk_keygen2* and *sk_keygen3* to generate two secret keys (*sk2*,*sk3*) which represent the secrets for two SIDH participants. These secret keys are used by 2^e- and 3^e-isogenies. Afterwards these secret keys are utilized to generate the public keys *pk2* and *pk3* with the respective isogenic functions *isogen_2* and *isogen_3*. After these calculations, two isogenic key pairs exist and are used to calculate the key exchange where one party uses *isoex_2* with the values *sk2* and *pk3* and the other party uses *isoex_3* with the values *sk3* and *pk3*. Both calculations should yield the same j-invariant which means, that the key exchange has been calculated successfully.

7.8.2 Public-Key-Encryption algorithm implementation

The next part of the main engine focuses on the Public-Key-Encryption(PKE) and the Key-Encapsulation-Mechanism(KEM). The KEM works very closely with the PKE and utilizes some of its parts to encrypt data in the Encapsulation.

```
public Transfer gen_pke() {
      Fp sk3 = mr.sk_keygen_3();
      PublicKey pk3 = mr.isogen_3(sk3.getValue());

      return new Transfer(sk3, pk3);
  }
```

The *gen_pke* function generates a new keypair with a secret and a public key that are used in the Public-Key-Encryption.

```
    public Transfer enc_pke(PublicKey pk3, byte[] data) {
        Fp sk2 = mr.sk_keygen_2();
        PublicKey c0 = mr.isogen_3(sk2.getValue());
        Fp2 j = mr.isoex_2(sk2.getValue(), pk3);
        Security.addProvider(new KeccakjProvider());
        CipherInterface ci = new Shake256StreamCipher();
        Shake256Key key = new Shake256Key(j.toByteStream());
        byte[] nonce = new byte[256];
        IvParameterSpec ivParameterSpec = new IvParameterSpec(nonce);
```

The first part of the Public-Key-Encryption generates a 2-isogeny key pair and a shared secret which is a j-invariant. This j-invariant serves as the key in the stream cipher that encrypts the data. The encryption happens with Shake256[75]. The NIST implementation does not force developers to use Shake256 and leaves the encryption-, hash- and digest-method up to the developer to decide. Because NIST uses Shake256 this implementation tries to implement this as close to the guideline as possible.

```
        try {
            ci.init(Cipher.ENCRYPT_MODE, key, ivParameterSpec);
            byte[] c1 = ci.doFinal(data);
            return new Transfer(c0, c1);
        } catch (InvalidKeyException ex) {
            Logger.getLogger(MathTester.class.getName()).log(Level.SEVERE,
                 null, ex);
        } catch (InvalidAlgorithmParameterException ex) {
            Logger.getLogger(MathTester.class.getName()).log(Level.SEVERE,
                null, ex);
        } catch (IllegalBlockSizeException ex) {
            Logger.getLogger(MathTester.class.getName()).log(Level.SEVERE,
                null, ex);
        } catch (BadPaddingException ex) {
            Logger.getLogger(MathTester.class.getName()).log(Level.SEVERE,
                null, ex);
        }
        return null;
    }
```

The last part of the function calls the correct stream cipher and encrypts the data. The values *c0*, which is the public key and the value *c1*, which is the encrypted data, is returned to the caller via the Transfer class.

```
    public byte[] dec_pke(Fp sk3, PublicKey c0, byte[] c1) {
        Fp2 j = mr.isoex_2(sk3.getValue(), c0);
        Security.addProvider(new KeccakjProvider());
        CipherInterface ci = new Shake256StreamCipher();
        Shake256Key key = new Shake256Key(j.toByteStream());
        byte[] nonce = new byte[256];
        IvParameterSpec ivParameterSpec = new IvParameterSpec(nonce);

        try {
            ci.init(Cipher.DECRYPT_MODE, key, ivParameterSpec);
        } catch (InvalidKeyException ex) {
            Logger.getLogger(MathTester.class.getName()).log(Level.SEVERE,
                null, ex);
        }

        byte[] data = ci.doFinal(j.toByteStream());

        return data;
    }
```

The *decrypt* function does the exact reverse operation compared to *encrypt* function. It takes the secret key *sk3* from the other party and the public key *c0* along with the encrypted data *c1* as input. With the secret key *sk3* and the public key *c0* a new j-invariant is being calculated. This j-invariant is the same as the one that was used for encrypting the data. It then calculates the original data by utilizing Shake256 with the new j-invariant. The function returns the original data to the caller.[39]

7.8.3 Key-Encapuslation-Mechanism algorithm implementation

```
    public Transfer gen_kem() {
        Fp sk3 = mr.sk_keygen_3();
        PublicKey pk3 = mr.isogen_3(sk3.getValue());

        return new Transfer(sk3, pk3);
    }
```

The *gen_kem* function generates a new keypair with a secret and a public key that are used in the Key-Encapsulation-Mechanism.

```
    public Transfer enc_kem(PublicKey pk3) {
        BigInt m = Engine.genRandom();
        Security.addProvider(new KeccakjProvider());
        CipherInterface ci = new Shake256StreamCipher();
        Shake256Key key = new Shake256Key(m.toByteStream());
        byte[] nonce = new byte[256];
        IvParameterSpec ivParameterSpec = new IvParameterSpec(nonce);

        try {
            ci.init(Cipher.ENCRYPT_MODE, key, ivParameterSpec);
            byte[] r = ci.doFinal(pk3.getBytes());
        } catch (InvalidKeyException ex) {
            Logger.getLogger(MathTester.class.getName()).log(Level.SEVERE,
                 null, ex);
        } catch (InvalidAlgorithmParameterException ex) {
            Logger.getLogger(MathTester.class.getName()).log(Level.SEVERE,
                 null, ex);
        }

        Transfer t = enc_pke(pk3, r)
        PublicKey c0 = t.getC0();
        byte[] shared_secret = ci.doFinal(c0.toByteStream());
        t.setSs(shared_secret);

        return t;
    }
```

The Key-Encapsulation-Mechanism performs a transformation according to Hofheinz, Hövelmanns and Kiltz[76] with some variations. The algorithm uses Shake256[75] to encapsulate and decapsulate the key. The function starts by generating a random value *m* that is then used as the encapsulation key for the public key *pk3*. The existing public key is then sent into the Public-Key-Encryption method to receive an encrypted public key where the encapsulated bits serve as the key. After that step, the algorithm applies a second layer of security by running the stream cipher again with the encrypted and encapsulated public key. The output of this function serves as a shared secret. The public key *c0*, the encrypted data *c1* and the shared secret *shared_secret* are returned to the caller.

```
public byte[] dec_kem(byte[] s, Fp2 sk3, PublicKey pk3, PublicKey c0,
    byte[] c1) {
    byte[] m = dec_pke(sk3, c0, c1);
    Security.addProvider(new KeccakjProvider());
    CipherInterface ci = new Shake256SteamCipher();
    Shake256Key key = new Shake256Key(m);
    byte[] nonce = new byte[256];
    IvParameterSpec ivParameterSpec = new IvParameterSpec(nonce);

    try {
        ci.init(Cipher.ENCRYPT_MODE, key, ivParameterSpec);
        byte[] r = ci.doFinal(pk3.getBytes());
    } catch (InvalidKeyException ex) {
        Logger.getLogger(MathTester.class.getName()).log(Level.SEVERE,
             null, ex);
    } catch (InvalidAlgorithmParameterException ex) {
        Logger.getLogger(MathTester.class.getName()).log(Level.SEVERE,
             null, ex);
    } catch (IllegalBlockSizeException ex) {
        Logger.getLogger(MathTester.class.getName()).log(Level.SEVERE,
             null, ex);
    } catch (BadPaddingException ex) {
        Logger.getLogger(MathTester.class.getName()).log(Level.SEVERE,
             null, ex);
    }

    PublicKey c0_1 = mr.isogen_2(new FP(r))

    byte[] ss;
    if (c0_1.equals(c0)) {
        ss = ci.doFinal(c0_1.getBytes());

    } else {
        Shake256Key key = new Shake256Key(s);
        ci.init(Cipher.ENCRYPT_MODE, key, ivParameterSpec);
```

```
            ss = ci.doFinal(c0_1.getBytes());
        }

        return byte[] ss;
    }
```

The final function *dec_kem* takes care of the decapsulation and decryption of the key. It takes the encrypted secret *s*, the secret key of one party *sk3*, the public keys *pk3* and *c0* and encrypted data *c1* as input. In the first step, the encrypted data is decrypted to receive back the random value *m*, that was generated in *enc_kem*. The decrypted value *m* is then used in the stream cipher as the capsulation value for the public key *c_0'* to generate the same *r* that is used in the isogenic calculation. The last part can go two different ways. When the 2-isogeny public key *c_0'* is the same as the 2-isogeny public key parameter *c_0*, the final shared secret is received by either decrypting the the value *m* or the second isogeny byte stream *c_0'*.[39]

7.9 Transfer class implementation

The transfer class does dataholding operations for the interactions between the Key-Encapsulation-Mechanism and the Public-Key-Encryption.

```
public class Transfer {
    private Fp sk3;
0    private PublicKey pk3, c0;
    private byte[] c1, ss;
```

The class can hold secret keys in the form of an $\mathbb{F}_p$ object, two public keys and a shared secret in the form of a byte array.

```
    public Transfer(Fp sk3, PublicKey pk3) {
        this.sk3 = sk3;
        this.pk3 = pk3;
    }

    public Transfer(PublicKey c0, byte[] c1) {
        this.c0 = c0;
        this.c1 = c1;
    }
    public byte[] getSs() {
        return ss;
    }

    public void setSs(byte[] ss) {
        this.ss = ss;
    }

    public PublicKey getC0() {
        return c0;
    }

    public void setC0(PublicKey c0) {
        this.c0 = c0;
    }

    public byte[] getC1() {
        return c1;
    }

    public void setC1(byte[] c1) {
        this.c1 = c1;
    }

    public Fp getSk3() {
        return sk3;
    }

    public void setSk3(Fp sk3) {
        this.sk3 = sk3;
    }

    public PublicKey getPk3() {
        return pk3;
    }

    public void setPk3(PublicKey pk3) {
        this.pk3 = pk3;
    }
}
```

The rest of the class are generic setter- and getter-methods for data input and ouptut and constructors to allocate data with the object creation.

8 Measurements

Good performance and fast computational speed is an important property of any cryptographic system. This is also a downside of post-quantum cryptography, since most of the mathematical operations, that are used in post-quantum cryptography generates a much higher load than conventional cryptographic systems. This is why it is of utmost importance to find a system that is able to fulfill the following properties:

- High quantum security
- Small key-sizes
- Small signature sizes
- Low memory demand
- Fast calculation speed

Finding a system that fits all properties while being secure is a hard task and that's why NIST and comparable organisations have a call for papers to motivate researchers and to further advance the field of quantum cryptography.
The Java implementation has the advantage that it is very versatile and mobile and can run on any system that has the JRE(Java-Runtime-Environement) installed. This comes with a performance hit in calculation speed and memory usage. That's why this implementation takes advantage of optimized algorithms and a fast big number allocation library. The main performance hit takes the generation of random numbers, because the PRNG(Pseudo-Random-Number-Generator) needs to do multiple iterations to generate correct sizes according to $\mathbb{F}_p$ and $\mathbb{F}_{p^2}$. The following measurements have been taken on an i7 6700HQ 2.6 GHz processor with four processing cores. It is important to note, that no JVM(Java-Virtual-Machine) optimizations and no code parallelisation or multithreading is being used.

- SIKE-Keygen: 200ms for the prime base SIKEp503
- SIKE-Key-Exchange: 40ms in the best case

9 Authentication schemes with isogenies

Authentication with isogenies is a topic that researchers have rarely considered, as the signatures were too large and the speed too slow compared to other post-quantum authentication systems. However, there was a big improvement with SeaSign[31], a new authentication scheme based on CSIDH, the group-based version of SIDH. SeaSign and authentication systems in general should be more focused on research. Therefore, many scientific papers on isogenic authentication are currently being written, as SeaSign, among others, seems to be really promising. In 2017, most people in this research field said that authentication systems will remain impractical. With SeaSign this could change quickly. While it is still slow and the security of CSIDH and SeaSign is unclear, more and more progress is being made. Both authentication schemes, CSIDH and SeaSign, will be explained more detailed in the next chaopters.

9.1 Authentication scheme based on SIDH

The first authentication scheme is a proposal from Galbraith, Petit and Silva from the year 2016 [23]. It describes two different authentication schemes, the older one from De Feo, Jao and Plût [17], and their new scheme based on an algorithm of Kohel et al. [77]. For the second scheme they use a quaternions, which are numbers that extend the complex numbers on the 4-dimensional plane. They proposed, that both authentication schemes are unforgeable in the classic random oracle model and in the quantum random oracle model. The comparison of both schemes has proven, that the De Feo scheme is more efficient although the new scheme from Galbraith, relies on more difficult computational problem.

9.2 Authentication scheme based on CSIDH

This second type of authentication scheme is quite new and bases its algorithm on CSIDH, the group based version of SIDH. The first interesting proposal came from De Feo and Galbraith[31]. They called it SeaSign, because CSIDH is spoken as sea-side. It combines the class group actions with the notion of Fiat-Shamir. This allows to have signature size of less than a kilobyte, while still having 128-bit security level. That comes with a slow down in speed and increases the signing and verification times to around 2-3 minutes. They state, that they didn't try to speed it up and they just wanted to make the signature as small as possible. So a speed up in the near future is expected.

10 Security

The last part of this thesis focues on the security and the related main mathematic principles that underlies SIDH. This chapter summarizes the works of [17] and [24].

Public Parameters:

- A prime P
- A supersingular elliptic curve E over Fp^2
- Four points on $E \Rightarrow P_a, P_b, Q_a, Q_b$

Private Parameters:

- Four random generated integers x_a, y_a, x_b, y_b
- Isogenies ϕA and ϕB

Shared Parameter:

- j-invariant of E_{ba}

Definition 1. *Given a field K and two supersingular elliptic curves over K such that* $|E_1| = |E_2|$, *compute an isogeny* $\phi : E_1 \mapsto E_2$

This isogeny is not unique, so the best representation is a pair of rational maps or a kernel which takes exponential time for both parties to calculate. One party can also represent an isogeny as a composition of multiple low degree isogenies, which can be calculated in polynomial time. This problem was stated in many papers beforehand. In the De Feo-Jao scheme it requires computing isogenies of degree ℓ for a small fixed prime ℓ.

Definition 2. *Given an elliptic curve* E *over a finite field K, compute its endomorphism ring.*

Another problem is the computation of the endomorphism ring of an elliptic curve. This was first shown by

Kohel [78] in 1996. He proved that in the supersingular case, a probabilistic algorithm is running in time $O(p)$. Three years later, Galbraith [79] showed that given the birthday paradox it can even run in $O(\sqrt{p})$. It still depends on the implementation, because for some curves it is still easy to calculate, but on average it is considered a serious problem. The problem that Rostovtsev and Stolbunov [18] had with their proposal in 2006 was the use of a commutative structure, where one can find a subexponential-time quantum algorithm [20]. However, the Jao-De Feo scheme is not vulnerable to that, because supersingular curves are non-commutative. That's the reason why there is a need for the auxiliary points, to get around the problems of non-commutativity. These could lead to attacks on the exchange scheme if one party uses a static key. This is why SIKE [39] and CSIDH [30] were invented. Some could perform a "small subgroup" or "invalid curve" attack which were proposed for DLP(Discrete-Logarithm-Problem) cryptosystems [80].

Active attacks and validation methods. Standard attacks on cryptosystems are active attacks that abuse static private keys. They are also known to attack methods based on the logarithm problem, where a user can be used as an oracle. The idea behind this is to learn something about the secret key of Alice $(a1, a2)$ using a known E'. The concept of "validation" is to prevent active attacks. In the context of supersingular-isogeny crypto-systems, the validation of (E, P, Q) should test that E is a supersingular curve and P and Q are on the curve in the correct order and P and Q are independent. One of this methods is the Kirkwood-Validation-Method [81]. It is built around the idea of using the randomness in the protocol to check that the protocol has performed correctly.

Importance of correct isogenies. As stated before, there are infinite isogenies from E to E_A. However, the Jao-De Feo system proposes the correct isogeny to calculate. This follows from the fact that $E/\langle G_A, G_B\rangle = E_A/\langle\phi_A(G_B)\rangle = E_B/\langle\phi_B(G_A)\rangle$. So $\phi_A/G_B()$ and $\phi_B(G_A)$ can be computed. An approach for an attacker to get to E_{BA}, is to compute $\phi_B(ker(\phi'))$ hence an isogeny from E_B with this kernel. But the attacker has no knowledge about these points, so he can only calculate $\phi_B(ker(\phi'))$ if $ker(\phi') \subseteq \langle P_A, Q_A\rangle$. A random isogeny ϕ' is unlikely to have this property. It is a main aspect, for computing the key, to reduce the computing of the endomorphism ring. All known algorithms to compute an isogeny from E to E_A, given end(E) and end(E_A), are not likely to result in the correct degree isogeny.

11 Conclusion and future prospects

11.1 Conclusion

In this thesis we have dealt with the problem of quantum computers and that they pose a great threat to modern communication and cryptography. One of the main contributions of our work is to provide a guide for the implementation of the SIKE protocol and a basis for understanding the mathematics based on it. The literature analysis provides a good overview of the current state of the art and historical facts required to understand SIKE. We have also added a short chapter on quantum computers and public-key cryptography to make it easier to understand. The most important part was the implementation, in which several different adjustments and improvements to the SIKE algorithm were made. These changes are described in great detail with a comprehensive explanation. Although Java may not be the go-to language for implementing SIKE, it shows that the basic problems of Java can be solved by adding more functions and libraries, writing different algorithms, and removing some dependencies. The paper mentions how Java can be used more efficiently and how computing times can be shortened with these enhancements. This gives readers a better grasp of the limitations Java has in dealing with large numbers and provides alternative ways to deal with them.

We have shown that the Java version is not as fast as implementations in C or Assembler, but the results were surprisingly good. There remains enough optimization space to make further improvements in calculation speed, execution times and memory optimizations. The implementation of several math sections as independent classes also allows them to be used as a library for future projects with isogenic cryptography. This could be especially useful for developers who want to integrate post-quantum cryptography into their programs. Many of these classes are very complex to understand, but the thesis gives guidance on their use and meaning in the algorithm. The mathematical part shows different calculations and gives visual and textual examples to the reader.

In the last part of the thesis there is a summary about the security of SIDH/SIKE and the most important mathematical operations and structures on which the algorithm is based on. It presents the most important security facts that will be important in the future. Until the final publication of this work in June 2019, there was no known algorithm that would have contributed to the acceleration of the isogeny calculations. SIKE also made it into the second round of the NIST standardization competition. To reach the second round, safety was proven and the key variables proved to be the shortest of all KEM candidates. This also explains the difference in authentication systems, which have greatly improved over the last 6 months. The older systems had very large signature sizes and the calculation was very slow. Newer signature techniques have shown that it is possible to have small signatures, although this has made the speed of signing and verification even slower. After 5 years of intensive research, cryptography with isogenies has advanced a lot and it is one of the most promising candidates to be used in the near future to ensure quantum computer security.

11.2 Future prospects

In this thesis we investigated cryptography on the basis of the isogeny problem. Implemented versions of SIKE already exist in real-world applications like TLS 1.3, but it is not foreseeable what kind of post-quantum cryptography will be state-of-the-art in the future. SIKE's Key-Enapsulation-Mechanism is a really promising candidate and recent work has brought a huge improvement in performance and reduced computing time to 6.3ms. The signatures of isogenies are very large compared to other post-quantum algorithms and have a long computation time. Therefore, Isogenies were not submitted for the authentication competition. The last years have shown that the possibility to improve isogeny based cryptography still has a high chance and there are further improvements possible. These improvements will result from code optimizations and mathematical accelerations. However, the other post-quantum candidates should not be underestimated. There will probably come a phase of realistic research where the bad ones are sorted out and many researchers focus on improvements for the rest. Quantum IT security is a large area that will increase in importance as larger quantum computers are built. This is the reason why there is still a great need for research and improvement in the future to keep the digital world safe in a quantum computing era.

List of Figures

4.1 Morphism from $X \mapsto Z$ 24

4.2 Points on elliptic curve over $\mathbb{F}_{13}$[38] 29

4.3 Isogenic calculation over $\mathbb{F}_{11^2}$[41] 33

4.4 Isogenic graph with degree 2[41] 34

4.5 Isogenic graph with degree 3[41] 34

7.1 Rotation of a coordinate on the complex plane 64

List of Tables

6.1 Performance benchmark BigInteger vs. BigInt 43

Listings

5.1 SIDH parameter setup[43] 37
5.2 Random public points generation 38
5.3 Alice isogeny generation[44][45] 38
5.4 Bob isogeny generation[44][45] 39
5.5 Alice secret point calculation[44][43] 40
5.6 Bob secret point calculation[44][43] 40
5.7 Same j-invariant values for both parties 40

Glossary

AES	Advanced Encryption Standard
DES	Data Encryption Standard
DH	Diffie-Hellman
IDE	Integrated Development Environment
IT	Information Technology
JDK	Java Development Kit
JRE	Java Runtime Environment
KEM	Key encapsulation Mechanism
OS	Operating System
RAM	Random Access Memory
SHA	Secure Hash Algorithm
SIDH	Supersingular Isogeny Diffie-Hellman
SIKE	Supersingular Isogeny Key Exchange

Bibliography

[1] W. Diffie and M. Hellman, "New directions in cryptography," *IEEE Transactions on Information Theory*, vol. 22, no. 6, pp. 644–654, Nov. 1976, ISSN: 0018-9448. DOI: `10.1109/TIT.1976.1055638`. [Online]. Available: `http://ieeexplore.ieee.org/document/1055638/` (visited on 05/22/2018).

[2] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Commun. ACM*, vol. 21, no. 2, pp. 120–126, Feb. 1978, ISSN: 0001-0782. DOI: `10.1145/359340.359342`. [Online]. Available: `http://doi.acm.org/10.1145/359340.359342` (visited on 05/22/2018).

[3] Richard P. Feynman, "Simulating physics with computers," *International Journal of Theoretical Physics*, vol. 21, no. 6, pp. 467–488, Jun. 1982, ISSN: 0020-7748, 1572-9575. DOI: `10.1007/BF02650179`. [Online]. Available: `http://link.springer.com/10.1007/BF02650179` (visited on 05/22/2018).

[4] David Deutsch, "Quantum theory, the church-turing principle and the universal quantum computer," in *Proceedings of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, vol. 400, The Royal Society, 1985, pp. 97–117.

[5] Stephen Cole Kleene, *Introduction to Metamathematics*. North Holland, 1952.

[6] David Hilbert and Wilhelm Ackermann, *Grundzüge der theoretischen Logik*, 2nd ed., ser. Grundlehren der mathematischen Wissenschaften. Berlin Heidelberg: Springer-Verlag, 1938, ISBN: 978-3-662-41928-1. [Online]. Available: `//www.springer.com/la/book/9783662419281` (visited on 05/22/2018).

[7] Alan Turing, "On computable numbers, with an application to the entscheidungsproblem," *Proceedings of the London Mathematical Society*, vol. 42, no. 1, pp. 230–265, 1936.

[8] Alonzo Church, "A note on the entscheidungsproblem," *The Journal of Symbolic Logic*, vol. 1, no. 1, pp. 40–41, Mar. 1936, ISSN: 0022-4812, 1943-5886. DOI: 10.2307/2269326. [Online]. Available: https://www.cambridge.org/core/journals/journal-of-symbolic-logic/article/note-on-the-entscheidungsproblem/ (visited on 05/22/2018).

[9] Kurt Gödel, "Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I," *Monatshefte für Mathematik und Physik*, vol. 38, no. 1, pp. 173–198, Dec. 1, 1931, ISSN: 0026-9255, 1436-5081. DOI: 10.1007/BF01700692. [Online]. Available: https://link.springer.com/article/10.1007/BF01700692 (visited on 05/22/2018).

[10] Peter W. Shor, "Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer," *SIAM review*, vol. 41, no. 2, pp. 303–332, 1999.

[11] (). A preview of bristlecone, google's new quantum processor, Google AI Blog, [Online]. Available: http://ai.googleblog.com/2018/03/a-preview-of-bristlecone-googles-new.html (visited on 05/22/2018).

[12] E. Berlekamp, R. McEliece, and H. van Tilborg, "On the inherent intractability of certain coding problems (corresp.)," *IEEE Transactions on Information Theory*, vol. 24, no. 3, pp. 384–386, May 1978, ISSN: 0018-9448. DOI: 10.1109/TIT.1978.1055873.

[13] O. M. Guillen, T. Pöppelmann, J. M. Bermudo Mera, E. F. Bongenaar, G. Sigl, and J. Sepulveda, "Towards post-quantum security for IoT endpoints with NTRU," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*, Mar. 2017, pp. 698–703. DOI: 10.23919/DATE.2017.7927079.

[14] Ralph Charles Merkle, Ralph Charles erkle, Ralph Charles Yerkle, Ate Students, Steve Pohlig, Raynold Kahn, and Dov Andleman, "Secrecy, authentication, and public key systems," 1979.

[15] Johannes Buchmann Dahmen Erik and Andreas Hülsing, "XMSS - a practical forward secure signature scheme based on minimal security assumptions," 484, 2011. [Online]. Available: http://eprint.iacr.org/2011/484 (visited on 01/07/2018).

[16] Daniel J. Bernstein, Daira Hopwood, Andreas Hülsing, Tanja Lange, Ruben Niederhagen, Louiza Papachristodoulou, Michael Schneider, Peter Schwabe, and Zooko Wilcox-O'Hearn, "SPHINCS: Practical stateless hash-based signatures," in *Advances in Cryptology – EUROCRYPT 2015*, ser. Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, Apr. 26, 2015, pp. 368–397. DOI: 10.1007/978-3-662-46800-5_15. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-662-46800-5_15 (visited on 01/08/2018).

[17] Luca De Feo, David Jao, and Jérôme Plût, "Towards quantum-resistant cryptosystems from supersingular elliptic curve isogenies," *Journal of Mathematical Cryptology*, vol. 8, no. 3, pp. 209–247, 2014.

[18] Alexander Rostovtsev and Anton Stolbunov, "Public-key cryptosystem based on isogenies.," *IACR Cryptology ePrint Archive*, vol. 2006, p. 145, 2006.

[19] Steven Galbraith and Anton Stolbunov, "Improved algorithm for the isogeny problem for ordinary elliptic curves," *arXiv:1105.6331 [cs, math]*, May 31, 2011. arXiv: `1105.6331`. [Online]. Available: `http://arxiv.org/abs/1105.6331` (visited on 05/22/2018).

[20] Andrew M. Childs, David Jao, and Vladimir Soukharev, "Constructing elliptic curve isogenies in quantum subexponential time," *Journal of Mathematical Cryptology*, vol. 8, no. 1, pp. 1–29, Jan. 1, 2014, ISSN: 1862-2976, 1862-2984. DOI: `10.1515/jmc-2012-0016`. arXiv: `1012.4019`. [Online]. Available: `http://arxiv.org/abs/1012.4019` (visited on 05/23/2018).

[21] David Jao and Vladimir Soukharev, "Isogeny-based quantum-resistant undeniable signatures," in *International Workshop on Post-Quantum Cryptography*, Springer, 2014, pp. 160–179.

[22] Craig Costello, Patrick Longa, and Michael Naehrig, "Efficient algorithms for supersingular isogeny diffie-hellman," in *Annual Cryptology Conference*, Springer, 2016, pp. 572–601.

[23] Steven D. Galbraith, Christophe Petit, and Javier Silva, "Identification protocols and signature schemes based on supersingular isogeny problems," 1154, 2016. [Online]. Available: `https://eprint.iacr.org/2016/1154` (visited on 12/11/2017).

[24] Steven D. Galbraith, Christophe Petit, Barak Shani, and Yan Bo Ti, "On the security of supersingular isogeny cryptosystems," in *Advances in Cryptology–ASIACRYPT 2016: 22nd International Conference on the Theory and Application of Cryptology and Information Security, Hanoi, Vietnam, December 4-8, 2016, Proceedings, Part I 22*, Springer, 2016, pp. 63–91.

[25] Craig Costello, David Jao, Patrick Longa, Michael Naehrig, Joost Renes, and David Urbanik, "Efficient compression of SIDH public keys," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, Springer, 2017, pp. 679–706.

[26] Amir Jalali, Reza Azarderakhsh, Mehran Mozaffari Kermani, and Daivd Jao, "Supersingular isogeny diffie-hellman key exchange on 64-bit ARM," *IEEE Transactions on Dependable and Secure Computing*, pp. 1–1, 2017, ISSN: 1545-5971. DOI: `10.1109/TDSC.2017.2723891`. [Online]. Available: `http://ieeexplore.ieee.org/document/7970184/` (visited on 12/11/2017).

[27] (May 18, 2018). PQCrypto-SIDH: SIDH library is a fast and portable software library that implements state-of-the-art supersingular isogeny cryptographic schemes. the chosen parameters aim to provide security agai.. original-date: 2017-04-17T23:16:37Z, [Online]. Available: `https://github.com/Microsoft/PQCrypto-SIDH` (visited on 05/23/2018).

[28] (May 16, 2018). PQCrypto-SIKE: This software is part of "supersingular isogeny key encapsulation", a submission to the NIST post-quantum standardization project. original-date: 2017-12-01T22:33:31Z, [Online]. Available: `https://github.com/Microsoft/PQCrypto-SIKE` (visited on 05/23/2018).

[29] Real World Crypto. (). Supersingular isogeny based cryptography gets practical | patrick longa (microsoft r.) | RWC 2018, [Online]. Available: `https://www.youtube.com/watch?v=31NyfrHSAco` (visited on 05/23/2018).

[30] Wouter Castryck, Tanja Lange, Chloe Martindale, Lorenz Panny, and Joost Renes, "CSIDH: An efficient post-quantum commutative group action," 383, 2018. [Online]. Available: `https://eprint.iacr.org/2018/383` (visited on 05/23/2018).

[31] Luca De Feo and Steven D. Galbraith, "SeaSign: Compact isogeny signatures from class group actions," in *Advances in Cryptology – EUROCRYPT 2019*, Yuval Ishai and Vincent Rijmen, Eds., vol. 11476, Cham: Springer International Publishing, 2019, pp. 759–789. DOI: `10.1007/978-3-030-17659-4_26`. [Online]. Available: `http://link.springer.com/10.1007/978-3-030-17659-4_26` (visited on 05/17/2019).

[32] Joseph H. Silverman, *The Arithmetic of Elliptic Curves*, 2nd ed., ser. Graduate Texts in Mathematics. New York: Springer-Verlag, 2009, ISBN: 978-0-387-09493-9. [Online]. Available: `https://www.springer.com/de/book/9780387094939` (visited on 06/03/2019).

[33] Luca De Feo, "Mathematics of isogeny based cryptography," *arXiv preprint arXiv:1711.04062*, 2017.

[34] Patrick Morandi, *Field and Galois theory*, ser. Graduate texts in mathematics 167. New York Berlin: Springer, 1996, 281 pp., OCLC: 246650112, ISBN: 978-0-387-94753-2.

[35] Rudolf Lidl and 1944- Niederreiter Harald, *Finite fields*, English, 2nd ed. Cambridge ; New York : Cambridge University Press, 1997, ISBN: 0521392314. [Online]. Available: `http://site.ebrary.com/id/10450828`.

[36] Olivia Di Matteo, "A very brief introduction to finite fields," p. 7,

[37] Conrad Keith. (2019). Finite fields, [Online]. Available: `https://kconrad.math.uconn.edu/blurbs/galoistheory/finitefields.pdf` (visited on 06/02/2019).

[38] (2019). Elliptic curves over finite fields, [Online]. Available: `http://www.graui.de/code/elliptic2/` (visited on 06/03/2019).

[39] Costello Craig, De Feo Luca, Jao David, Longa Patrick, Naehrig Michael, and Renes Joost, "Supersingular isogeny key encapsulation," Apr. 17, 2019. [Online]. Available: `https://sike.org/files/SIDH-spec.pdf`.

[40] Peter L. Montgomery, "Speeding the pollard and elliptic curve methods of factorization," *Mathematics of Computation*, vol. 48, no. 177, pp. 243–264, 1987, ISSN: 0025-5718, 1088-6842. DOI: `10.1090/S0025-5718-1987-0866113-7`. [Online]. Available: `https://www.ams.org/mcom/1987-48-177/S0025-5718-1987-0866113-7/` (visited on 06/03/2019).

[41] Craig Costello, "An introduction to supersingular isogeny-based cryptography," p. 78,

[42] Dana Mackenzie, "Hash of the future?" *Science*, vol. 319, no. 5869, pp. 1481–1481, Mar. 14, 2008, ISSN: 0036-8075, 1095-9203. DOI: `10.1126/science.319.5869.1481`. [Online]. Available: `http://science.sciencemag.org/content/319/5869/1481` (visited on 09/07/2018).

[43] William Stein, "SAGE reference manual," p. 2931, [Online]. Available: `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.103.2051%5C&%5Clinebreak%20rep=rep1%5C&type=pdf`.

[44] Daniel Shumow, "Isogenies of elliptic curves: A computational approach," p. 55, [Online]. Available: `https://eprint.iacr.org/2009/522.pdf`.

[45] (). Elliptic curves over a general field — sage reference manual v8.2: Plane curves, [Online]. Available: `https://doc.sagemath.org/html/en/reference/curves/sage%5Clinebreak%20/schemes/elliptic%5Ctextunderscore%20curves/ell%5Ctextunderscore%20field.html` (visited on 05/18/2018).

[46] Raza Ali Kazmi, "ISOGENIES AND CRYPTOGRAPHY," p. 100,

[47] Simon Klein. (Apr. 19, 2019). Outperforming BigInteger since 2015. possibly providing the fastest available arbitrary-precision arithmetic integer classes purely written in java. because performance matters (and we like java)! .. original-date: 2015-03-31T17:37:17Z, [Online]. Available: `https://github.com/bwakell/Huldra` (visited on 06/02/2019).

[48] (2019). BigNum arithmetic in java — let's outperform BigInteger! Codeforces, [Online]. Available: `https://codeforces.com/blog/entry/17235` (visited on 06/02/2019).

[49] (2019). BigInteger (java platform SE 7), [Online]. Available: `https://docs.oracle.com/javase/7/docs/api/java/math/BigInteger.html` (visited on 06/02/2019).

[50] (2019). Java - difference between mutable objects and immutable objects, Stack Overflow, [Online]. Available: `https://stackoverflow.com/questions/4658453/difference-between-mutable-objects-and-immutable-objects` (visited on 06/02/2019).

[51] Nozomu Nishihara, Ryuichi Harasawa, Yutaka Sueyoshi, and Aichi Kudo, "A remark on the computation of cube roots in finite fields," p. 10,

[52] Gora Adj, Francisco Rodriguez-Henriquez, and Universite Claude Bernard Lyon, "Square root computation over even extension fields," p. 33,

[53] Harold N Ward, "Quadratic residue codes and symplectic groups," *Journal of Algebra*, vol. 29, no. 1, pp. 150–171, Apr. 1974, ISSN: 00218693. DOI: `10.1016/0021-8693(74)90120-3`. [Online]. Available: `https://linkinghub.elsevier.com/retrieve/pii/0021869374901203` (visited on 06/02/2019).

[54] (2019). Legendre symbol | brilliant math & science wiki, [Online]. Available: `https://brilliant.org/wiki/legendre-symbol/`.

[55] (2019). Euler's criterion | brilliant math & science wiki, [Online]. Available: `https://brilliant.org/wiki/eulers-criterion/`.

[56] D. A. Burgess, "The distribution of quadratic residues and non-residues," *Mathematika*, vol. 4, no. 2, pp. 106–112, 1957. DOI: `10.1112/S0025579300001157`.

[57] (2019). SecureRandom (java platform SE 8), [Online]. Available: `https://docs.oracle.com/javase/8/docs/api/java/security/SecureRandom.html`.

[58] *Pseudorandom number generator*, in *Wikipedia*, Page Version ID: 895034635, May 1, 2019. [Online]. Available: `https://en.wikipedia.org/w/index.php?title=Pseudorandom%5C_number%5C_generator%5C&oldid=895034635` (visited on 06/02/2019).

[59] (2019). Modular exponentiation, [Online]. Available: `http://gauss.math.luc.edu/greicius/Math201/Fall2012/Lectures/modular-exponentiation.article.pdf`.

[60] *Binary logarithm*, in *Wikipedia*, Page Version ID: 897941866, May 20, 2019. [Online]. Available: `https://en.wikipedia.org/w/index.php?title=Binary%5C_logarithm%5C&oldid=897941866` (visited on 06/02/2019).

[61] E. Savas and C.K. Koc, "The montgomery modular inverse-revisited," *IEEE Transactions on Computers*, vol. 49, no. 7, pp. 763–766, Jul. 2000, ISSN: 00189340. DOI: `10.1109/12.863048`. [Online]. Available: `http://ieeexplore.ieee.org/document/863048/` (visited on 06/02/2019).

[62] Róbert Lórencz, "New algorithm for classical modular inverse," in *Cryptographic Hardware and Embedded Systems - CHES 2002*, Burton S. Kaliski, çetin K. Koç, and Christof Paar, Eds., red. by Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen, vol. 2523, Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 57–70. DOI: `10.1007/3-540-36400-5_6`. [Online]. Available: `http://link.springer.com/10.1007/3-540-36400-5%5C_6` (visited on 06/02/2019).

[63] Ceské Vysoké Utení Technické V Praze and Workshop, Eds., *Montgomery multiplcative inverse 2003*, OCLC: 53271701, Prague: Czech Technical University, 2003, ISBN: 978-80-01-02708-0.

[64] (2015). Algorithm - complex numbers product using only three multiplications, Stack Overflow, [Online]. Available: `https://stackoverflow.com/questions/19621686/complex-numbers-product-using-only-three-multiplications`.

[65] Andre Weimerskirch and Christof Paar, "Generalizations of the karatsuba algorithm for efficient implementations," p. 17,

[66] Winograd S., "On the number of multiplications required to compute certain functions," pp. 1840–1842, 1967. DOI: `https://doi.org/10.1073/pnas.58.5.1840`.

[67] *Modular exponentiation*, in *Wikipedia*, Page Version ID: 897262055, May 15, 2019. [Online]. Available: `https://en.wikipedia.org/w/index.php?title=Modular_exponentiation&oldid=897262055`.

[68] *Exponentiation by squaring*, in *Wikipedia*, Page Version ID: 897990493, May 20, 2019. [Online]. Available: `https://en.wikipedia.org/w/index.php?title=Exponentiation_by_squaring&oldid=897990493`.

[69] Uni-Mainz, "Square roots in finite prime fields," p. 4, [Online]. Available: `https://www.staff.uni-mainz.de/pommeren/Cryptology/Asymmetric/5%5C_NTh/SqRprim.pdf`.

[70] Siguna Müller, "On the computation of square roots in finite fields," *Designs, Codes and Cryptography*, vol. 31, no. 3, pp. 301–312, Mar. 2004, ISSN: 0925-1022. DOI: 10.1023/B:DESI.0000015890.44831.e2. [Online]. Available: http://link.springer.com/10.1023/B:DESI.0000015890.44831.e2 (visited on 06/02/2019).

[71] Gonzalo Tornaría, "Square roots modulo p," in *LATIN 2002: Theoretical Informatics*, Sergio Rajsbaum, Ed., vol. 2286, Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 430–434. DOI: 10.1007/3-540-45995-2_38. [Online]. Available: http://link.springer.com/10.1007/3-540-45995-2%5C_38 (visited on 06/02/2019).

[72] Ivo Kubjas, "Modern elliptic curve cryptography," p. 12,

[73] Sylvain Duquesne, "Montgomery scalar multiplication for genus 2 curves," in *Algorithmic Number Theory*, Duncan Buell, Ed., red. by David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi, and Gerhard Weikum, vol. 3076, Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 153–168. DOI: 10.1007/978-3-540-24847-7_11. [Online]. Available: http://link.springer.com/10.1007/978-3-540-24847-7%5C_11 (visited on 06/02/2019).

[74] Craig Costello and Benjamin Smith, "Montgomery curves and their arithmetic: The case of large characteristic fields," *Journal of Cryptographic Engineering*, vol. 8, no. 3, pp. 227–240, Sep. 2018, ISSN: 2190-8508, 2190-8516. DOI: 10.1007/s13389-017-0157-6. [Online]. Available: http://link.springer.com/10.1007/s13389-017-0157-6 (visited on 06/02/2019).

[75] Amund Elstad. (Apr. 8, 2019). Java implementation of FIPS 202, SHA3-xxxx, SHAKE128, SHAKE256 : Aelstad/keccakj. original-date: 2014-10-22T13:44:46Z, [Online]. Available: https://github.com/aelstad/keccakj (visited on 06/02/2019).

[76] Dennis Hofheinz, Kathrin Hövelmanns, and Eike Kiltz, "A modular analysis of the fujisaki-okamoto transformation," in *Theory of Cryptography*, Yael Kalai and Leonid Reyzin, Eds., vol. 10677, Cham: Springer International Publishing, 2017, pp. 341–371. DOI: 10.1007/978-3-319-70500-2_12. [Online]. Available: http://link.springer.com/10.1007/978-3-319-70500-2%5C_12 (visited on 06/02/2019).

[77] Kristin Tignol, Christoph Lauter, Jean-Pierre Petit, and Kohel David, "On the quaternion ℓ-isogeny path problem," 505, 2014. [Online]. Available: http://eprint.iacr.org/2014/505 (visited on 06/03/2019).

[78] David Russell Kohel, *Endomorphism Rings of Elliptic Curves Over Finite Fields*. University of California, Berkeley, 1996, 260 pp., Google-Books-ID: EBNNAQAAMAAJ.

[79] Steven D. Galbraith, “Constructing isogenies between elliptic curves over finite fields,” *LMS Journal of Computation and Mathematics*, vol. 2, pp. 118–138, 1999, ISSN: 1461-1570. DOI: `10.1112/S1461157000000097`. [Online]. Available: `https://www.cambridge.org/` (visited on 05/23/2018).

[80] Chae Hoon Lim and Pil Joong Lee, “A key recovery attack on discrete log-based schemes using a prime order subgroup,” in *Advances in Cryptology — CRYPTO ’97*, ser. Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, Aug. 17, 1997, pp. 249–263. DOI: `10.1007/BFb0052240`. [Online]. Available: `https://link.springer.com/chapter/10.1007/BFb0052240` (visited on 05/23/2018).

[81] Daniel Kirkwood, Bradley C Lackey, John McVey, Mark Motley, Jerome A Solinas, and David Tuller, “Failure is not an option: Standardization issues for post-quantum key agreement,” presented at the Workshop on Cybersecurity in a Post-Quantum World, 2015, p. 21.

Since the SIDH/SIKE post-quantum cryptosystem is a relatively new area of cryptography and requires a lot of research time and work, this bachelor thesis was done as a joint work of two students. Although the work was done together, some of it was defined as independent work in order to prove that one can write a scientific paper or bachelor thesis:

Stefan Schubert

- Introduction - Chapter: 1
- Historical facts - Chapter: 2
- Literature analysis on SIDH/SIKE - Chapter: 3
- Mathematical preliminaries - Chapter: 4
- Authentication with Isogenies - Chapter: 9
- Security of SIDH/SIKE - Chapter: 10
- Conclusion and future prospects - Chapter: 11

Manuel Ravnik

- Abstract
- Finite fields - Chapter: 4.3
- Graphical display of the SIDH algorithm - Chapter: 5.1
- Mathematics of Montgomery curves - Chapter: 4.5
- Implementation of SIDH in Sagemath - Chapter: 5
- Java Huldra-Library - Chapter: 6
- Full implementation of the SIKE Cryptosystem - Chapter: 7
- Speed and performance measurements - Chapter: 8