FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Marek Behún

Post-quantum alternative to secure sockets

Department of Software Engineering

Supervisor of the bachelor thesis:	Miroslav Kratochvíl, M.Sc.
Study programme:	Computer Science
Study branch:	General Computer Science

Prague 2017

Title: Post-quantum alternative to secure sockets

Author: Marek Behún

Department: Department of Software Engineering

Supervisor: Miroslav Kratochvíl, M.Sc., Department of Software Engineering

Abstract: The goal of this thesis is to implement a software library that provides a wrapping of real-time socket-like communication into an cryptographic protocol with purpose similar to SSL or TLS, that is secure against an adversary in possession of a quantum computer. Resulting software utilizes the Supersingular Isogeny Diffie Hellman (SIDH) key-exchange algorithm for achieving this level of security, and is simple, portable and independent on system-specific primitives. The thesis gives a concise introduction to the theory on which SIDH is built, targeting the audience of undergraduate students of Computer Science.

Keywords: encryption security post-quantum cryptography diffie-hellman supersingular isogeny elliptic curves

"It is not paranoia if they really are out to get you."
— an unknown paranoid

This thesis is dedicated to all those who systematically try to overcome cognitive biases that make them underestimate the need for proper caution and security, the biases that make them *not paranoid enough*.

I would also like to thank my supervisor Mirek Kratochvíl, who invested his time to introduce me into this interesting problem, which also led me to study the beautiful theory behind it.

Contents

Introduction

The security of trapdoor functions used by current cryptographic key-exchange schemes is usually based on some number-theoretic problem related either to integer factorization or the discrete logarithm problem. It is now more than twenty years since the algorithm that solves these problems in polynomial time on a quantum computer was published, and more than thirty since the dawn of the development of cryptographic schemes that are considered safe against an attacker possessing such a quantum computer—these schemes are usually referred to as *post-quantum safe schemes*.

In spite of this, there are few software packages implementing these quantum-secure schemes, at least when compared to those which focus on classical, *pre-quantum* cryptography.

This thesis aims to contribute to this development by implementing a secure communication library that employs a key-exchange scheme constructed from a post-quantum trapdoor function, namely the difficulty of finding *isogenies between supersingular elliptic curves*.

Motivation

In 1994, Peter Shor presented a quantum algorithm that performs integer factorization in polynomial time [Sho95], that was later also shown to be able to efficiently perform discrete logarithm on elliptic curves [PZ03]. As a result, at the moment somebody succeeds in building a large quantum computer, practically all asymmetric ciphers used today (RSA, DSA, elliptic curves, etc.) are instantly rendered insecure. Although the record for largest integer to be successfully factored using Shor's algorithm, which was achieved in 2012, does not really seem to show any possible harm to the security of classical cryptographic schemes (the integer factored was 21), there is no reason to believe that fully-functional large quantum computers are far future. Moreover, research in related topic of adiabatic quantum computing was successful in factoring the integer 56153, also in 2012 [DB14].

Furthermore, taking into account that

- there is a risk that the existence of working quantum computers large enough to attack classical cryptographic schemes would be for a certain period of time concealed

- it is expectable that these computers will exist in such a near future when someone could still gain an unfair profit by breaking the security of encrypted messages archived at present time

- there are already proposed post-quantum alternatives to classical cryptographic schemes, they are easy to implement and in certain ways very neat

- in general it is not recommended for the security of the whole system to be built around one problem

it would be appreciable to have a practical, usable software ready as soon as possible.

Goals

The main goal of this thesis is to implement a post-quantum alternative to current SSL and TLS protocols [DR08, TLS]. To achieve that, we implement a software library that is able to wrap real-time socket-like communication in a protocol to provide the required security, is easy to use, has small specification and straightforward implementation, is easily extensible and has minimal dependencies on other software.

It was decided not to implement a public key infrastructure such as X.509 as known from SSL/TLS, since it would have to depend on post-quantum signatures, which, although they do exist, are not quite practical, usually because of the size of the key or signatures (the most promising signature scheme is called SPHINCS-256 [BHH$^+$15] and its signatures are 41 KiB long). Instead, as a substitution, we utilize post-quantum asymmetric encryption in an SSH-like authentication model.

Although a general post-quantum alternative to a forward-secure Diffie-Hellman like key-exchange scheme did not exist for a long time, recent development [FJP11] has shown a brilliant new method for a quantum-secure key-exchange algorithm based on computing isogenies on supersingular elliptic curves. Our library is expected to use the Supersingular Isogeny Diffie-Hellman (SIDH) key-exchange algorithm and shall be implemented in the C++ programming language (with a possibility of implementing wrapper functions around the C++ API so that it can also be accessible from programs written in C).

The programming interface should be inspired by that of the `TLS` class of Qt Cryptographic Architecture [QCA], since we want to have the library only do the actual protocol- and cryptography-related computations, and leave the system-dependent part of the code on the user—QCA does this in a sensible way.

The second goal is to document the inner workings of the resulting software, to explain the communication protocol and to present several example programs that use the library to achieve security.

The third goal of this thesis is to give an introduction to the theory behind the key exchange method used (SIDH), since most readers ever only learn at most about RSA—there is not much introductory material into elliptic curve isogenies. The reader is expected to have been introduced to some basics of number theory and group theory. We try to simplify the more complex parts of the theory, giving informal definitions of some concepts, and avoiding some of them completely (for example, we do not talk about algebraic closure of a field at all). This should not break the soundness of the theory nor give incorrect notions to those readers who would then like to continue studying this beautiful theory in more depth.

Related Work

Related Research

Note that there are other possible post-quantum secure key-exchange methods this software could have focused on, like the Ring Learning With Errors key-exchange [Sin15], based on lattice-based cryptography. We have chosen SIDH because it uses smallest key sizes of all currently available post-quantum key-exchange algorithms, and provides perfect forward secrecy.

Similar Software

To the best of our knowledge and apart from the basic implementation by the authors of the SIDH algorithm [SSi], the only open-source SIDH implementations currently available to general public are:

- SIDH Library from Microsoft Research [MSR] is a MIT licensed C implementation of the SIDH algorithm focused on efficiency which also implements an ECDH+SIDH hybrid called BigMont.

- openssl-sidh [Oss] is an OpenSSL plugin which uses the SIDH Library.

- liboqs from Open Quantum Safe [OQSa] is a C library for quantum-safe cryptographic algorithms released under the MIT License and supports multiple key-exchange methods as cryptographic primitives (it uses the SIDH Library for SIDH implementation). Open Quantum Safe also integrates liboqs into their fork of OpenSSL [OQSb].

An API for secure sockets is only provided by the openssl-sidh plugin and the OpenSSL fork from OQS, but it seems that the openssl-sidh project has virtually no documentation nor does seem to be actively developed, and the OpenSSL fork from OQS does not yet provide key-exchange method based on SIDH. Also, these packages do not provide protection from active attacks using countermeasures such as signature schemes.

Furthermore, there is another significant problem we wanted to avoid—OpenSSL has a troublesome vulnerability history, which may, among other things, be due to the size of the library and a not-quite-exemplary history of resolving bug reports [Lib, Ope]. Note that this fact has led some major projects invest resources into cleaning it and releasing a fork of the library—there is now a patched OpenSSL called BoringSSL used in Google Chrome [Bor], and a massively cleaned OpenSSL fork from the OpenBSD project called LibreSSL [Luc14, Ram].

Layout of this Thesis

In Chapter 1 we give an introduction into Galois fields, and show how to perform fast arithmetic operations on a certain type of them.

We explain the algebraic concept of an elliptic curve over a finite field in Chapter 2, together with how the group operations can be constructed on an elliptic curve. We also show how the operation called elliptic curve point multiplication is performed and how it is used for cryptographic purposes.

In Chapter 3 we define isogenies of elliptic curves and explain how an isogeny can be computed given its kernel. We briefly explain what supersingular elliptic curves are, and then describe the key-exchange protocol and the public-key encryption scheme based on isogenies between supersingular elliptic curves.

Chapter 4 describes the communication protocol that carries data encrypted by our library, and also the inner workings of the library, which includes a brief overview of its sub-modules and their interaction.

We conclude in Conclusion, giving an overview of accomplished goals and a summary of complications encountered while developing the software. Some ideas for future development are also presented.

1. Galois Fields

Given an algebraic structure, such as the real numbers, with a set of axioms defining that structure and a set of proved properties of that structure, mathematicians have often tried to see how many of the structure defining axioms they can remove and still have a valid proof of a given property—this is called generalization. For example, the definition of a convergent sequence is not limited to real numbers, because the notion of distance is all it takes to define convergence—and thus the results of the theory of metric spaces is valid not only for real numbers, but all structures that can properly define the notion of distance.

A field is one such generalized structure, taking only some of the axioms of real numbers, but still having enough common with their structure that it is possible to think about polynomials over a field, or elliptic curves over it.

When we limit the cardinality of that structure to only contain a finite number of elements, we are talking about finite fields, or Galois[1] fields.

In this Chapter we introduce finite fields, stating for which orders finite fields do exist and we show how to compute efficiently in one particular kind of them.

We present most of the theorems in this chapter without a proof. If required, proofs can be found in any literature concerning abstract algebra, for example in that from Judson [Jud13].

1.1 Galois Fields in General

Definition 1. *For $d \in \mathbb{N}$, a finite (Galois) field of d elements is the tuple $\mathbb{F}_d = (F, +, \cdot, 0_F, 1_F)$, where:*

- *F is a set of d elements,*
- *$0_F, 1_F \in F$ and $0_F \neq 1_F$,*
- *$F(+, 0_F)$ and $F \setminus \{0_F\}\,(\cdot, 1_F)$ are Abelian groups,*
- *for every $a, b, c \in F$, $a \cdot (b + c) = a \cdot b + a \cdot c$.*

The number $d = |F|$ is called the order of the field.

As we can see, the structure has enough properties to be able to express constructions like integer powers or multiplicative inverses.

[1]Évariste Galois (25 October 1811 – 31 May 1832) was a French mathematician whose work led to the foundation of Galois theory and group theory.

Definition 2. *For $n \in \mathbb{N}$ and $x \in \mathbb{F}_d$, where $\mathbb{F}_d$ is a Galois field*

- *the expression $n \cdot x$ denotes $\overbrace{x + x + \cdots + x}^{n \text{ times}}$ and x^n denotes $\overbrace{x \cdot x \cdot \cdots \cdot x}^{n \text{ times}}$,*
- *$-x$ denotes the additive inverse of x (such that $x + (-x) = 0_F$),*
- *$-n \cdot x$ is $n \cdot (-x)$ and $0 \cdot x$ is 0_F,*
- *x^{-1} or $\frac{1}{x}$ is the multiplicative inverse of x (such that $x \cdot x^{-1} = 1_F$),*
- *x^{-n} is $(x^{-1})^n$ and if $x \neq 0_F$, then x^0 is 1_F.*

Our aim is to construct a Galois field of a given number of elements. The following gives characterization of the allowed cardinalities.

Definition 3. *The characteristic of a Galois field $\mathbb{F}_d$ is the smallest $p \in \mathbb{N}$ such that $p \cdot 1_F = 0_F$, or 0 if no such p exists.*

Theorem 1. *If $\mathbb{F}_d$ is a Galois field, then $d = p^n$, where p is a prime and $n \in \mathbb{N}$. Two Galois fields of the same order are isomorphic and the characteristic of $\mathbb{F}_{p^n}$ is p.*

Corollary. If $\mathbb{F}_p$ is a Galois field of prime order p, then $\mathbb{F}_p \simeq \mathbb{Z}_p$, where $\mathbb{Z}_p$ is the ring of integers modulo p.

The elements of a finite field can be represented as *polynomials over a base field.* We therefore define the following:

Definition 4. *A polynomial over a finite field $\mathbb{F}_d$ is a polynomial with coefficients from $\mathbb{F}_d$. The set of all polynomials over the field $\mathbb{F}_d$ is denoted $\mathbb{F}_d(x)$ and it forms a commutative ring.*

Definition 5. *An irreducible polynomial over a Galois field $\mathbb{F}_d$ is a non-constant polynomial $p(x)$ over $\mathbb{F}_d$ such that $p(x)$ cannot be factored into two non-constant polynomials over $\mathbb{F}_d$. That is, there are no non-constant polynomials $q(x)$, $r(x)$ over $\mathbb{F}_d$ such that $p(x) = q(x) \cdot r(x)$.*

Now we can use these notions to finally construct finite field of given order:

Theorem 2 (Galois field construction)**.** *A Galois field $\mathbb{F}_{p^n}$ of order p^n, where p is prime and $n > 1$, is isomorphic to the quotient ring $\mathbb{F}_p(x)/q(x)$, where $q(x)$ is an irreducible polynomial of degree n over $\mathbb{F}_p$. Such a polynomial always exists.*

Because of Theorem 2, we can represent the elements of a finite field as polynomials over $\mathbb{Z}_p$ for some prime p. Therefore, in the following text, we will denote some of this elements with the symbols usually reserved for integers (such as 0, -1, 1). They map into the finite field naturally—an integer is also a constant polynomial.

A simple way to find an irreducible polynomial for a field of order p^2 for a prime $p \equiv 3 \pmod 4$ is shown in the next section.

1.2 Arithmetic in $\mathbb{F}_{p^2}$

Now that we can construct Galois fields, we need algorithms for performing basic arithmetical operations on their elements. Most of them can be computed "from definition", but because the atomic operations on large integers are costly, we want to speed them up as much as we can.

We shall describe explicit algorithms for arithmetic in $\mathbb{F}_{p^2}$ for a prime p such that $p \equiv 3 \pmod 4$. This specific p is used because of the following lemmas.

Definition 6. *A non-zero element $r \in \mathbb{F}_d$ is called a quadratic residue over $\mathbb{F}_d$ if there exists an element $q \in \mathbb{F}_d$ such that $q^2 = r$, or, equivalently, $q^2 - r = 0$. Otherwise, it is called a quadratic non-residue.*

Theorem 3 (Euler's criterion)**.** *A non-zero element $r \in \mathbb{F}_d$ is a quadratic residue if and only if $r^{\frac{d-1}{2}} = 1$ and a quadratic non-residue if and only if $r^{\frac{d-1}{2}} = -1$.*

Lemma 4. *If p is a prime such that $p \equiv 3 \pmod 4$, then the polynomial (x^2+1) is irreducible over $\mathbb{F}_p$.*

Proof. We have

$$(-1)^{\frac{p-1}{2}} = (-1)^{\frac{3-1 \bmod 4}{2}} = (-1)^1 = -1$$

therefore -1 is a quadratic non-residue in $\mathbb{F}_p$, thus $x^2 = -1$ has no solutions, which is equivalent to (x^2+1) being irreducible. $\square$

Thus, by selecting a prime p such that $p \equiv 3 \pmod 4$, we can always use the polynomial (x^2+1) as the irreducible polynomial for construction of $\mathbb{F}_{p^2}$.

Moreover, because $x^2 \equiv -1 \pmod{x^2+1}$, the field also has the arithmetical structure of complex numbers, and so we will write the elements of $\mathbb{F}_{p^2}$ in the form $a + ib$, where $a, b \in \mathbb{F}_p$.

1.2.1 Addition, Substraction and Negation

The operations of addition and substraction in $\mathbb{F}_{p^2}$ are done by components. We present the pseudocode for addition in Algorithm 1, substraction is done correspondingly.

To save some computations, we can avoid using the modulo operation in negation (shown in Algorithm 2).

1.2.2 Multiplication and Squaring

The naive multiplication algorithm based on the formula

$$(a + ib) \cdot (c + id) = (ac - bd) + i(ad + bc)$$

uses four multiplications. We can save one multiplication if we use the identity

$$(a - b) \cdot (c + d) = ac + ad - bc - bd.$$

The pseudocode is shown in Algorithm 3. Note that if the multiplication algorithm gets specialized for squaring, we can save one more multiplication.

1.2.3 Multiplicative Inverse

We follow the method also used for inversing complex numbers to find multiplicative inverse in $\mathbb{F}_{p^2}$:

$$\frac{1}{a + ib} = \frac{a - ib}{(a + ib) \cdot (a - ib)} = \frac{a}{a^2 + b^2} + i\frac{-b}{a^2 + b^2}$$

This is shown in Algorithm 4.

To find the multiplicative inverse in $\mathbb{F}_{p^2}$ one has to find the multiplicative inverse in $\mathbb{F}_p$, which is also called modular multiplicative inverse. Extended Euclidean algorithm [Euc] can be used for this task. Both the modular multiplicative inverse and Extended Euclidean algorithm operations are implemented in the GMP library (as `mpz_invert(o, i, m)`, which stores the inverse of `i` modulo `m` into `o` and `mpz_gcdext(g, s, t, a, b)`, which stores the greatest common divisor of `s` and `t` into `g` and also into `a` and `b` such coefficients that $\mathtt{a} \cdot \mathtt{s} + \mathtt{b} \cdot \mathtt{t} = \mathtt{g}$). Because of this, in the text below, we shall consider these operations atomic.

1.2.4 Square Root

Finding square roots in $\mathbb{F}_{p^2}$ again requires the ability to find square roots in $\mathbb{F}_p$.

Algorithm 1 Addition in $\mathbb{F}_{p^2}$

```
function AddGFp2($a + ib, c + id$)
    $a \leftarrow (a + c) \mod p$
    $b \leftarrow (b + d) \mod p$
    return $a + ib$
end function
```

Algorithm 2 Negation in $\mathbb{F}_{p^2}$

```
function NegateGFp2($a + ib$)
    $a \leftarrow p - a$
    $b \leftarrow p - b$
    return $a + ib$
end function
```

Algorithm 3 Multiplication in $\mathbb{F}_{p^2}$

```
function MultiplyGFp2($a + ib, c + id$)
    $t_1 \leftarrow (a - b) \cdot (c + d)$
    $t_2 \leftarrow ad$
    $t_3 \leftarrow bc$
    $a \leftarrow (t_1 - t_2 + t_3) \mod p$
    $b \leftarrow (t_2 + t_3) \mod p$
    return $a + ib$
end function
```

Algorithm 4 Multiplicative inverse in $\mathbb{F}_{p^2}$

```
function InverseGFp2($a + ib \neq 0$)
    $t \leftarrow$ ModularInverseGFp($a^2 + b^2$)
    $a \leftarrow a \cdot t \mod p$
    $b \leftarrow -b \cdot t \mod p$
    return $a + ib$
end function
```

Lemma 5. *If p is a prime number and x is a quadratic residue in $\mathbb{F}_p$, then $\sqrt{x} = x^{\frac{p+1}{4}}$.*

Proof. As x is a quadratic residue, Theorem 3 says that $x^{\frac{p-1}{2}} = 1$. Now

$$\sqrt{x}^2 = \left(x^{\frac{p+1}{4}}\right)^2 = x^{\frac{p+1}{2}} = x^{\frac{p-1}{2}} \cdot x = 1 \cdot x = x$$

$\square$

Lemma 6. *If p is a prime number and x a quadratic residue in $\mathbb{F}_p$, it has exactly two square roots: if d is one of them, $-d$ is the other.*

Proof. Suppose that there are d, e such that $d^2 = e^2$. Then $d^2 - e^2 = 0$ which can be factored into $(d+e)\cdot(d-e) = 0$, and because in a field it holds that if $ab = 0$, then either $a = 0$ or $b = 0$, we have $d = e$ or $d = -e$. $\square$

Now, given an element $c + id$ that is a quadratic residue in $\mathbb{F}_{p^2}$, we want to find $a + ib$ such that $(a + ib)^2 = c + id$. After multiplicatoin of the left side we can compare the coefficients:

$$\begin{aligned} a^2 - b^2 &= c \\ 2ab &= d. \end{aligned}$$

Substituting $b = \frac{d}{2a}$ into the first equation yields

$$a^2 - \frac{d^2}{4a^2} = c,$$

which is a quadratic equation for $A = a^2$ with the solution

$$A_{1,2} = \frac{c \pm \sqrt{c^2 + d^2}}{2}$$

Now only one $A \in \{A_1, A_2\}$ is a quadratic residue over $\mathbb{F}_p$ if $c + id \neq 0$. Were it not so, we would have four different square roots, which is not possible according to lemma 6.

We can find out which one is the residue using Euler's criterion 3, receiving $a = \sqrt{A}$ and $b = \frac{d}{2a}$.

The pseudocode for square root is shown in Algorithm 5.

Algorithm 5 Square root in $\mathbb{F}_{p^2}$

```
function SqrtGFp2(a quadratic residue $c + id \neq 0$)
    $t \leftarrow$ SqrtGFp($c^2 + d^2$)
    $A \leftarrow (c + t)/2$
    if not IsResidueGFp(A) then
        $A \leftarrow (c - t)/2$
    end if
    $a \leftarrow$ SqrtGFp($A$)
    $b \leftarrow d/(2a)$
    return $a + ib$
end function
```

2. Elliptic Curves over a Field

The study of structure of finite groups is one of the main building blocks of modern cryptography. The theory of finite groups is a rich and nontrivial field of mathematics which has led to many useful applications.

To make use of the structure of finite groups of required cryptographic properties we need a way to compactly represent the groups and their elements. The theory of elliptic curves gives one such way to represent certain types of finite Abelian groups.

To avoid misunderstanding, we give a forward informal definition of an elliptic curve: an elliptic curve over a field $\mathbb{F}$ is the set of points $E = \{O\} \cup \{(x, y) \in \mathbb{F}^2 \mid y^2 = x^3 + ax + b\}$ for some $a, b \in \mathbb{F}$ together with a group operation on this set.

2.1 Short Weierstrass Form

We begin with the definition of the short Weierstrass form of an elliptic curve equation.

Definition 7. *Given a field $\mathbb{F}$ of characteristic neither 2 nor 3, the* short Weierstrass form of an elliptic curve *E is given by the equation*

$$E : y^2 = x^3 + ax + b, \tag{2.1}$$

where $a, b \in \mathbb{F}$ and $4a^3 + 27b^2 \neq 0$.

The condition $(4a^3 + 27b^2) \neq 0$ is required for the property of the curve being smooth, that is, it has to contain no singular points (geometrically, these are points where the tangent cannot be properly defined: intersections or cusps). This happens when the polynomial on the right side of the Equation 2.1 does not have a multiple root.

Lemma 7. *The polynomial $p(x) = x^3 + ax + b$ defined over a field $\mathbb{F}$ has a multiple root if and only if $4a^3 + 27b^2 = 0$.*

Proof. If $p(x)$ has a multiple root α, it is either a double root or a triple root. If it is a double root, then by dividing $p(x)$ by $(x - \alpha)^2$ we get $(x - \beta)$, where β is the other root. In either case we can write

$$p(x) = (x - \alpha)^2(x - \beta) = x^3 + (-2\alpha - \beta)x^2 + \left(\alpha^2 + 2\alpha\beta\right)x - \alpha^2\beta.$$

Comparison of coefficients leads to $-2\alpha - \beta = 0$, $a = \alpha^2 + 2\alpha\beta$, $b = \alpha^2\beta$. From this $a = -3\alpha^2$ and $b = 2\alpha^3$, and thus

$$4a^3 + 27b^2 = 4\left(-3\alpha^2\right)^3 + 27\left(2\alpha^3\right)^2 = -108\alpha^6 + 108\alpha^6 = 0.$$

On the other hand, if $4a^3 + 27b^2 = 0$, consider that α such that $a = -3\alpha^2$ is a root of $p(x)$. Then $27b^2 = -4a^3 = 108\alpha^6$, thus $b = \pm 2\alpha^3$. Since α can be replaced by $-\alpha$, assume that $a = -3\alpha^2$ and $b = 2\alpha^3$. This holds for $p(x) = (x - \alpha)^2(x + 2\alpha)$. □

Lemma 8. *An elliptic curve E over a field $\mathbb{F}$ with a short Weierstrass form defined by Equation 2.1 is smooth if and only if the Equation 2.1 does not have a multiple root.*

Proof. Since we have not defined derivatives of polynomials over a field nor proven that they have properties similar enough to those over $\mathbb{R}$, we prove this lemma only for the base field $\mathbb{R}$.

Let the polynomial on the right side of the Equation 2.1 have three roots, α, β and γ, so that the equation can be written as $y^2 = (x-\alpha)(x-\beta)(x-\gamma)$. Differentiating this implicitly defined function we get the tangent at point (x, y):

$$\begin{aligned}\frac{\mathrm{d}y}{\mathrm{d}x} &= -\frac{(x-\beta)(x-\gamma) + (x-\alpha)(x-\gamma) + (x-\alpha)(x-\beta)}{-2y} \\ &= \pm\frac{(x-\beta)(x-\gamma) + (x-\alpha)(x-\gamma) + (x-\alpha)(x-\beta)}{2\sqrt{(x-\alpha)(x-\beta)(x-\gamma)}}\end{aligned}$$

This is properly defined[1] for all $(x, y) \in E$ if and only if α, β, γ are pairwise different. □

The Weierstrass form can be defined for fields of characteristic 2 and 3 as well, but in these cases the equation is more complicated:

$$y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6.$$

When the characteristic of the field $\mathbb{F}$ is neither 2 nor 3, one can transform the coordinates with $\eta = y + (a_1x + a_3)/2$ and $\xi = x + (a_1^2 + 4a_2)/12$ to get the short Weierstrass form.

[1]we consider the improper derivative, which arises when $x \in \{\alpha, \beta, \gamma\}$, as properly defined, since it properly defines the tangent

2.2 The Group Law

We will now show how to perform group operations on a curve with a short Weierstrass form, thus showing that points on an elliptic curve form a group. Consider a curve E defined over the real numbers $\mathbb{R}$ by the Equation 2.1.

We want to construct an operation $+_E$ such that the operation with the points on curve E yields an Abelian group. Since the curve is symmetrical about the x-axis, given a point P, we define $-_E P$ as the point opposite to it. Given two points $P, Q \in E$, we can uniquely describe a third point $R = P +_E Q$ by first drawing a line between P and Q, and then taking as $-_E R$ the point where the line intersects E the third time. Lemma 9 shows that there are no more intersections.

Lemma 9. *Given a field $\mathbb{F}$ of characteristic 0 or greater than 2 and points $a, b, c, d \in \mathbb{F}$, a curve defined by $y^2 = x^3 + ax + b$ has at most three intersections with a line defined by $y = cx + d$.*

Proof. We have

$$\begin{aligned}
(cx+d)^2 &= x^3 + ax + b \\
c^2x^2 + 2cdx + d^2 &= x^3 + ax + b \\
0 &= x^3 - c^2x^2 + (a - 2cd)x + (b - d^2)
\end{aligned}$$

which is a cubic equation. Since the standard algebraic derivation of roots of a cubic equation is correct also in $\mathbb{F}$ (as shown for example on Wikipedia [Cub]), we conclude that it has at most three solutions. $\square$

For two distinct, non-opposite points P, Q, Figure 2.1 shows how $P +_E Q = R$ can be found. In the case that $P = Q$, the tangent at P is used as the line for finding the intersection, which is illustrated in Figure 2.2. In the case of opposite points, $P = -_E Q$, we define $P +_E Q = O$, where O is the neutral point of the group operation, the so called *point at infinity.* There is one special case: if P is an inflection point, we take $P +_E P = -_E P$.

To obtain explicit formulas for the coordinates of point $R = P +_E Q = (x_R, y_R)$ given coordinates $P = (x_P, y_P)$, $Q = (x_Q, y_Q)$, one has first to find the equation for the line crossing P and Q, which is $y = \lambda(x - x_P) + y_P$, where the slope λ is computed as

$$\lambda = \begin{cases} \dfrac{y_P - y_Q}{x_P - x_Q}, & \text{if } x_P \neq x_Q \\ \dfrac{3x_P^2 + a}{2y_P}, & \text{if } x_P = x_Q, \end{cases} \tag{2.2}$$

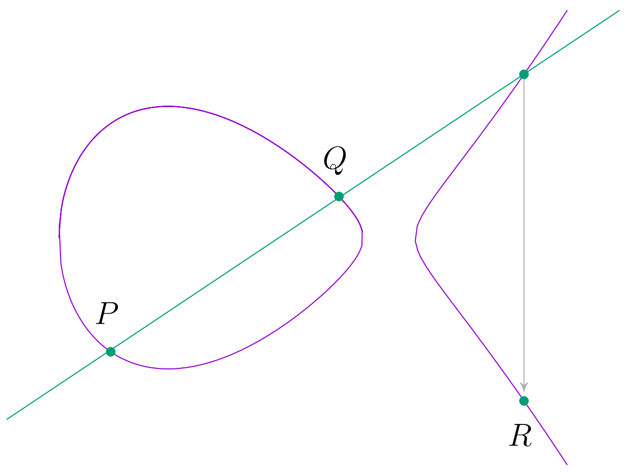

Figure 2.1: The group law when P and Q are distinct, non-opposite, non-neutral points

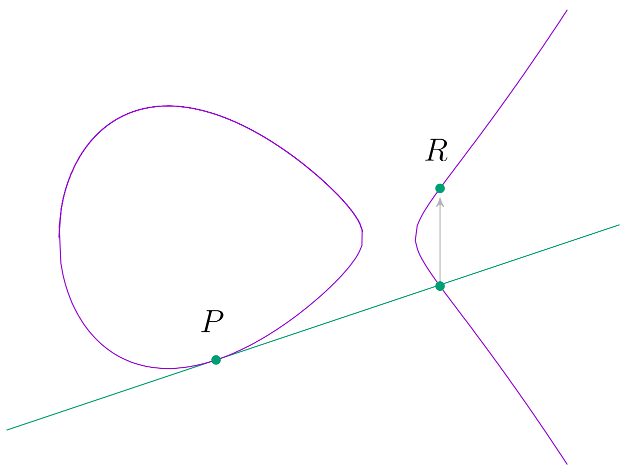

Figure 2.2: The group law when doubling a point P, i.e. when $P = Q$

and then substitutes the right side of this equation as y into Equation 2.1. This reduces into a cubic equation in the x variable. Taking into account that Equation 2.1 holds also for the pairs (x_P, y_P) and (x_Q, y_Q), one can then divide by the polynomial $(x - x_P) \cdot (x - x_Q)$ and finally obtain:

$$x_R = \lambda^2 - x_P - x_Q \tag{2.3}$$

$$y_R = \lambda \cdot (x_P - x_R) - y_P \tag{2.4}$$

The operation we have just defined, usually called *elliptic curve point addition*, is clearly commutative. It is an exercise in manipulation with algebraic expressions to check that associativity also holds.

Observation. Linear transformations of the coordinates do not change the group structure of the curve—the points of intersection transform linearly as well.

Note that the Equations 2.2, 2.3 and 2.4 are properly defined for any field $\mathbb{F}$ with characteristic different from 2 and 3, not just for real numbers. Given Equation 2.1 defining curve E with $a, b \in \mathbb{F}$ and points $P, Q \in \mathbb{F}^2$ such that $P, Q \in E$, the elliptic curve point addition yields a point $R = P +_E Q \in E$.

Remark. The group law also holds when working with a generic field $\mathbb{F}$ of characteristic 0 or greater than 2, not just on $\mathbb{R}$.

Lemma 10. *Given a short Weierstrass form $E_{a,b} : y^2 = x^3 + ax + b$ where $a, b \in \mathbb{F}$, there is a short Weierstrass form $E_{c,d} : y^2 = x^3 + cx + d$ with $c, d \in \mathbb{F}$ and $(c, d) \neq (a, b)$ having the same group structure as $E_{a,b}$.*

Proof. As observed above, the group structure does not change when transforming the coordinates linearly. Thus a change of variables $x' = 2x$ and $y' = y/\sqrt{8}$ does not change the group structure and yields equation $y'^2 = x'^3 + \frac{1}{4}ax' + \frac{1}{8}b$. $\square$

We can now properly define an elliptic curve as understood in abstract algebra:

Definition 8. *By an elliptic curve E over a field $\mathbb{F}$ with a short Weierstrass form $y^2 = x^3 + ax + b$, where $a, b \in \mathbb{F}$, we understand a set of points of the form $P = (x, y)$ for $x, y \in \mathbb{F}$ and $y^2 = x^3 + ax + b$ together with a special* point at infinity *O, equipped with a group operation $+_E$ as defined by the group law.*

2.3 Point Multiplication

Elliptic curve point multiplication is the trapdoor function used in classical elliptic curve cryptography (ECC).

Definition 9. *Given an elliptic curve E, integer n and point $P \in E$, we define the product of n with P as $[n]P = \underbrace{P +_E P +_E \cdots +_E P}_{n\ times}$ for $n > 0$, $[n]P = [-n] -_E P$ for $n < 0$ and $[n]P = O$ for $n = 0$.*

The problem of finding the value of n as the solution to $P = [n]Q$ given points $P, Q \in E$ is known as the elliptic curve discrete logarithm problem, and is used in classical ECC in the same way as the integer factorization problem is used in RSA. As we have mentioned in Introduction, there is a known polynomial time quantum algorithm that solves this problem, and so classical ECC schemes are not quantum-safe. Although not as a trapdoor function, elliptic curve point multiplication is also used in the SIDH algorithm.

A naive algorithm for elliptic curve point multiplication needs $\mathcal{O}(n)$ elliptic curve point additions. This is, naturally, inefficient, and therefore an algorithm similar to exponentiation by squaring is used. There are several different algorithms for efficient multiplication, but for the need to avoid side-channel attacks which arise from timing, the Montgomery ladder is used most frequently (shown in Algorithm 6).

2.4 Elliptic Curve Diffie-Hellman

As we have said, the classical elliptic curve based protocols assume that finding the elliptic curve discrete logarithm of a random point on an elliptic curve is infeasible. We will now describe how a key-exchange scheme using elliptic curve point multiplication as a trapdoor function works—the Elliptic Curve Diffie Hellman algorithm.

First, the public parameters are chosen: a prime field $\mathbb{F}_p$ or a binary field $\mathbb{F}_{2^m}$ for prime m together with an irreducible polynomial f of order m over $\mathbb{F}_2$, an elliptic curve $E : y^2 = x^3 + ax + b$ on that field, and a base-point $P \in E$ with a prime order n, such that the cyclic subgroup generated by P is not too much smaller from E, preferably $n = |E|$, but at least $n \geq \frac{|E|}{4}$.

Now suppose that Alice and Bob want to establish a shared secret key over an insecure channel. Alice generates a random number $a \in_R \mathbb{Z}_n \setminus \{0\}$, computes $A = [a]P$ and sends A to Bob. Bob, similarly, generates a random $b \in_R \mathbb{Z}_n \setminus \{0\}$, computes $B = [b]P$ and sends B to Alice. Upon receiving B, Alice computes $[a]B$ and upon receiving A, Bob computes $[b]A$. Since

Algorithm 6 Montgomery ladder elliptic curve point multiplication algorithm

```
function MontgomeryLadder($n \in \mathbb{Z}, P \in E$)
    if $n < 0$ then
        return MontgomeryLadder($-n, -_E P$)
    end if
    $Q \leftarrow O$
    $R \leftarrow P$
    for $i$-th bit $b$ of $n$ from most significat bit to the least do
        if $b = 1$ then
            $Q \leftarrow R +_E Q$
            $R \leftarrow R +_E R$
        else
            $R \leftarrow R +_E Q$
            $Q \leftarrow Q +_E Q$
        end if
    end for
    return $Q$
end function
```

$[a]B = [a][b]P = [ab]P = [b][a]P = [b]A$, they have both computed the same result, which is used as the shared secret key.

3. Supersingular Isogeny Diffie-Hellman

In this Chapter we give a basic introduction to the theory of isogenies between elliptic curves, on which the Supersingular Isogeny Diffie-Hellman key-exchange algorithm is built.

We explain what is an isogeny between elliptic curves and we outline the notion of supersingular elliptic curves. We then show how an isogeny can be represented, how can certain kind of isogenies be computed efficiently and how they can be used by the parties of a key-exchange scheme to generate two isomorphic elliptic curves. We then show how the *j-invariant* of this isomorphism class can be computed and used as a shared secret.

In the last Section we describe the SIDH key-exchange algorithm as proposed by Jao, Feo and Plût [FJP11].

We do not go into much details in this Chapter, since the theory is rather complex. For interested readers we recommend the book of Silverman [Sil92] and Connell [Con96] and the article of Shumow [Shu09].

3.1 Isogenies Between Elliptic Curves

Definition 10. *An* isogeny *from elliptic curve E_1 to elliptic curve E_2, both over a field $\mathbb{F}$, is a map which preserves the neutral element and maps every other point $(P_x, P_y) \in E_1$ to a point $(g_x(P_x), g_y(P_x, P_y)) \in E_2$, where g_x, g_y are rational functions over $\mathbb{F}$. If the denominator of either g_x or g_y is evaluated to 0 for (P_x, P_y), the result is taken to be the neutral element.*

Two elliptic curves are isogenous if there exists an isogeny between them.

Note that this definition of isogeny prohibits constant isogenies (those which map every point to the neutral element), which is different from some literature.

Example. For $m \in \mathbb{N}$, the multiplication-by-m map $[m]$, which takes $P \in E$ to $[m]P \in E$, is an isogeny from E to E.

Lemma 11. *The relation of being isogenous is an equivalence relation.*

Theorem 12. *Every isogeny is a homomorphism. That is, if $\phi : E_1 \to E_2$ is an isogeny, then for all $P, Q \in E_1$*

$$\phi(P +_{E_1} Q) = \phi(P) +_{E_2} \phi(Q).$$

Theorem 13. *If $\phi : E_1 \to E_2$ is an isogeny, then $\ker \phi = \phi^{-1}(O)$ is a finite group.*

Theorem 14 (Tate's theorem)**.** *Two elliptic curves E_1 and E_2 defined over a finite field $\mathbb{F}$ are isogenous if and only if $|E_1| = |E_2|$.*

Theorem 15. *Let E be an elliptic curve over $\mathbb{F}$ and Φ be a finite subgroup of E. Then there is a unique (up to isomorphism) elliptic curve E' and isogeny ϕ such that $\phi : E \to E'$ and $\ker \phi = \Phi$.*

Definition 11. *Let E be an elliptic curve over $\mathbb{F}$ and Φ a finite subgroup of E. The unique elliptic curve which is the image of isogeny ϕ with $\ker \phi = \Phi$, and which exists thanks to Theorem 15, is denoted by the quotient E/Φ or E/ϕ. The degree of isogeny ϕ, $\deg \phi$, is equal to the cardinality of Φ and we shall call ϕ the* isogeny induced by Φ on E.

Hence we can identify an isogeny by specifying its kernel, and given a kernel subgroup, we can compute the corresponding isogeny using Vélu's formulas.

Theorem 16 (Vélu's formulas)**.** *Let E be an elliptic curve over $\mathbb{F}$ defined with the equation $y^2 = x^3 + ax + b$, Φ a finite subgroup of E, ϕ an isogeny induced by Φ on E and $l = \deg \phi$. If $O \neq P = (P_x, P_y) \in E$, then $\phi(P) = (\phi_x(P), \phi_y(P))$, where*

$$\begin{aligned}\phi_x(P) &= P_x + \sum_{Q \in \Phi \setminus \{O\}} \left((P +_E Q)_x - Q_x\right) \\ \phi_y(P) &= P_y + \sum_{Q \in \Phi \setminus \{O\}} \left((P +_E Q)_y - Q_y\right).\end{aligned}$$

Furthermore, if we define the polynomial D such that

$$D(z) = \prod_{Q \in \Phi \setminus \{O\}} (z - Q_x) = z^{l-1} - \sigma_1 z^{l-2} + \sigma_2 z^{l-3} - \sigma_3 z^{l-4} + \cdots$$

and take σ_1, σ_2 and σ_3 from $D(z)$ to define

$$\begin{aligned}t &= a(l-1) + 3\left(\sigma_1^2 - 2\sigma_2\right) \\ w &= 3a\sigma_1 + 2b(l-1) + 5\left(\sigma_1^3 - 3\sigma_1\sigma_2 + 3\sigma_3\right),\end{aligned}$$

then the target curve of isogeny ϕ, E/ϕ, is given by the equation $y^2 = x^3 + a'x + b'$, where

$$\begin{aligned}a' &= a - 5t \\ b' &= b - 7w.\end{aligned}$$

Theorem 17. *Let E be an elliptic curve over $\mathbb{F}$ and Φ a finite subgroup of E. If the degree of isogeny $\phi : E \to E/\Phi$ induced by Φ on E is greater than 1, then the isogeny ϕ can be factored into a composition of isogenies of prime degree.*

If Ψ is a subgroup of Φ and $\psi : E \to E/\Psi$ the isogeny induced by Ψ on E, then $\psi(\Phi)$ is a finite subgroup of E/Ψ and for the isogeny $\chi : E/\Psi \to E/\psi(\Phi)$ induced by $\psi(\Phi)$ on E/Ψ holds:

$$\phi = \chi \circ \psi$$

Thanks to this theorem we can efficiently compute isogenies of smooth degree (a smooth number is informally defined as a number which can be factored completely into small primes). We describe the iterative method in corollary.

Corollary. Let l be a small prime and $e \in \mathbb{N}$. Let E_0 be an elliptic curve and $\langle R_0 \rangle$ a cyclic subgroup of E_0 of order l^e generated by the point R. For $0 \le i < e$ put

$$\begin{aligned} E_{i+1} &= E_i / \left\langle \left[l^{e-i-1} \right] R_i \right\rangle \\ \phi_i &: E_i \to E_{i+1} \\ R_{i+1} &= \phi_i(R_i) \end{aligned}$$

Then $\phi = \phi_{e-1} \circ \phi_{e-2} \circ \cdots \circ \phi_1 \circ \phi_0$ is the isogeny generated by $\langle R_0 \rangle$ on E_0.

3.1.1 Supersingular Elliptic Curves

We showed how isogenies can be computed efficiently if they are of smooth order.

If an elliptic curve E has smooth order ($|E|$ is a smooth number), then there are many isogenies of smooth order on E—taking a random point on E with a large smooth order as a generator of the inducing subgroup would lead to such isogeny.

There is a certain kind of curves which are easy to construct to have smooth number of elements [Brö09]. We give the definition and state one property of this kind of elliptic curves, and we leave it at that, since the theory behind the construction is not important for our implementation.

Definition 12. *Let $\mathbb{F}$ be a field of characteristic $p > 0$. An elliptic curve E defined over $\mathbb{F}$ is* supersingular *if and only if the kernel of the multiplicative map $[p] : E \to E$ is trivial, that is*

$$\{R \in E \mid [p]R = O\} = \{O\}$$

Theorem 18. *If elliptic curves E_1 and E_2 defined over a finite field $\mathbb{F}$ are isogenous, then either both E_1 and E_2 are supersingular, or none is.*

3.2 The j-invariant of an Elliptic Curve

We will now extend the observation about linear transformations from Section 2.2 to describe a method by which it can be checked if two elliptic curves defined by their short Weierstrass form are isomorphic.

Observation. Two elliptic curves E_1 and E_2 over a field $\mathbb{F}$ of characteristic not 2 nor 3 defined by their short Weierstrass forms are isomorphic if and only if there is a linear transformation of the x-coordinate and corresponding linear transformation of the y-coordinate in their short Weierstrass forms that can bring one curve onto the other.

Lemma 19. *Two elliptic curves over a field $\mathbb{F}$ of characteristic not 2 nor 3 given by their short Weierstrass forms $E_1 : y^2 = x^3 + ax + b$ and $E_2 : y^2 = x^3 + a'x + b'$ are isomorphic if and only if there is a non-zero $\lambda \in \mathbb{F}$ such that $a' = \lambda^2 a$ and $b' = \lambda^3 b$.*

Proof. Apply to the short Weierstrass form of E_1 the linear transformation $x \mapsto \alpha x + \beta$ for $\alpha, \beta \in \mathbb{F}$, $\alpha \neq 0$ to get

$$\begin{aligned} y^2 &= (\alpha x + \beta)^3 + a(\alpha x + \beta) + b \\ \frac{y^2}{\alpha^3} &= x^3 + \frac{3\beta}{\alpha} x^2 + \frac{3\beta^2 + a}{\alpha^2} x + \frac{\beta^3 + a\beta + b}{\alpha^3}. \end{aligned}$$

Now apply linear transformations $x \mapsto x - \frac{\beta}{\alpha}$ so that the coefficient of x^2 vanishes and $y^2 \mapsto y^2\alpha^3$ to normalize the equation:

$$y^2 = x^3 + \frac{a}{\alpha^2} x + \frac{b}{\alpha^3}.$$

Finally take λ as $\frac{1}{\alpha}$.

The other implication is straightforward. □

Definition 13. *The j-invariant of an elliptic curve E over field $\mathbb{F}$ with characteristic not 2 nor 3 with short Weierstrass form $E : y^2 = x^3 + ax + b$ is defined as*

$$\mathrm{j}(E) = 1728 \frac{4a^3}{4a^3 + 27b^2}.$$

Now we can finally express how the j-invariant identifies the isomorphism class of an elliptic curve.

Lemma 20. *Let $\mathbb{F}$ be a field of characteristic not 2 nor 3. Elliptic curves E_1, E_2 over $\mathbb{F}$ are isomorphic if and only if $\mathrm{j}(E_1) = \mathrm{j}(E_2)$.*

Proof. By lemma 19 there is a $\lambda \in \mathbb{F}$ such that $E_1 : y^2 = x^3 + ax + b$ and $E_2 : y^2 = x^3 + \lambda^2 ax + \lambda^3 b$. Now

$$\begin{aligned}
\mathrm{j}(E_2) &= 1728 \frac{4\left(\lambda^2 a\right)^3}{4\left(\lambda^2 a\right)^3 + 27\left(\lambda^3 b\right)^2} \\
&= 1728 \frac{4\lambda^6 a^3}{4\lambda^6 a^3 + 27\lambda^6 b^2} \\
&= 1728 \frac{4a^3}{4a^3 + 27b^2} \\
&= \mathrm{j}(E_1).
\end{aligned}$$

□

3.3 The SIDH Key-Exchange Algorithm

Now that we have introduced the required theory, we can explain the SIDH key-exchange algorithm and the corresponding public-key encryption algorithm.

3.3.1 Parameter Generation

First, a prime number of the form $p = l_A^{e_A} l_B^{e_B} f \pm 1$ is generated, where l_A, l_B are small, different prime numbers, e_A, e_B are exponents defining the number of security-bits and f is a (preferentially small) cofactor so that p is prime. Then a supersingular elliptic curve E_0 is constructed over F_{p^2} with cardinality $\left(l_A^{e_A} l_B^{e_B} f\right)^2$. This curve has group structure $\left(\mathbb{Z}/\left(l_A^{e_A} l_B^{e_B} f\right)\mathbb{Z}\right)^2$. Finally a basis $\{P_A, Q_A\}$ of $E_0[l_A^{e_A}] = \{R \mid R \in E_0,\ [l_A^{e_A}]R = O\}$ and a basis $\{P_B, Q_B\}$ of $E_0[l_B^{e_B}] = \{R \mid R \in E_0,\ [l_B^{e_B}]R = O\}$ are found using the method described in the article of Jao, Feo and Plût [FJP11].

3.3.2 Key-Exchange

Alice and Bob want to establish a shared secret key over an insecure channel. They do this by taking random walks on the isogeny graph: Alice will walk the graph consisting of isogenies of degrees l_A and Bob of degrees l_B.

Alice generates random integers $m_A, n_A \in_R \mathbb{Z}_{l_A^{e_A}}$, not both divisible by l_A, and computes the isogeny $\phi_A : E_0 \to E_A$ with kernel

$$\langle [m_A]P_A +_{E_0} [n_A]Q_A \rangle .$$

Since the kernel is of smooth order, this can be done efficiently as described in the corollary in Section 3.1. She also computes the images of points P_B and Q_B through the isogeny ϕ_A and sends the triplet $\{E_A, \phi_A(P_B), \phi_A(Q_B)\}$ to Bob. Bob does the corresponding with his parameters.

Upon receipt of $\{E_B, \phi_B(P_A) \in E_B, \phi_B(Q_A) \in E_B\}$ from Bob, Alice computes the image curve of the isogeny $\phi'_A : E_B \to E_{AB}$ with kernel

$$\langle [m_A]\phi_B(P_A) +_{E_B} [n_A]\phi_B(Q_A)\rangle \,.$$

Bob, again, does the corresponding with his parameters.

Alice and Bob can then use the common j-invariant of

$$\begin{aligned} E_{AB} &= \phi'_A(\phi_B(E_0)) \cong \phi'_B(\phi_A(E_0)) = E_{BA} \cong \\ &\cong E_0/\left\langle [m_A]P_A +_{E_0} [n_A]Q_A, [m_B]P_B +_{E_0} [n_B]Q_B \right\rangle \end{aligned}$$

as the shared secret key.

Figure 3.1 illustrates this.

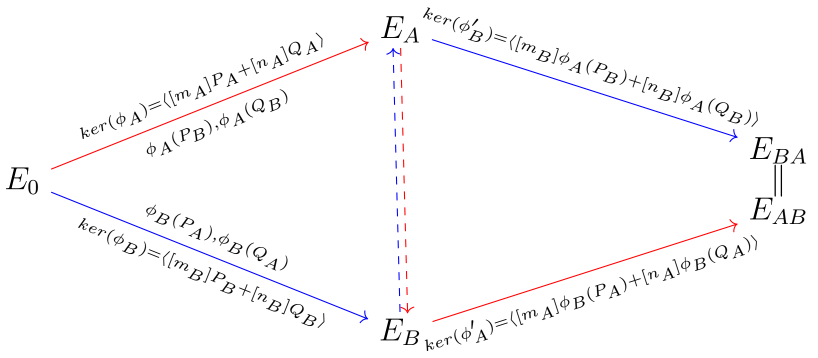
E_0
E_A
E_B
E_{BA}
E_{AB}
$ker(\phi_A)=\langle [m_A]P_A+[n_A]Q_A\rangle$
$\phi_A(P_B),\phi_A(Q_B)$
$\phi_B(P_A),\phi_B(Q_A)$
$ker(\phi_B)=\langle [m_B]P_B+[n_B]Q_B\rangle$
$ker(\phi'_B)=\langle [m_B]\phi_A(P_B)+[n_B]\phi_A(Q_B)\rangle$
$ker(\phi'_A)=\langle [m_A]\phi_B(P_A)+[n_A]\phi_B(Q_A)\rangle$

Figure 3.1: SIDH key-exchange protocol (figure taken from the article of Jao, Feo and Plût [FJP11])

3.3.3 Public-Key Encryption

The key-exchange method can be adapted into an ElGamal-like encryption scheme. Let $\mathcal{H} = \{H_k \,|\, k \in K\}$ be a set of hash functions indexed by a finite set K, where each $H_k : \mathbb{F}_{p^2} \to \{0,1\}^w$.

Key generation Alice generates random integers $m_A, n_A \in_R \mathbb{Z}_{l_A^{e_A}}$, not both divisible by l_A, and a random index $k \in_R K$ and computes the isogeny

ϕ_A, curve E_A and points $\phi_A(P_B)$ and $\phi_A(Q_B)$ as in the key-exchange scheme. The public key is the tuple $(E_A, \phi_A(P_B), \phi_A(Q_B), k)$ and the private key is the tuple (m_A, n_A, k).

Encryption Bob wants to encrypt a message $m \in \{0,1\}^w$ using public key $(E_A, \phi_A(P_B), \phi_A(Q_B), k)$. Bob generates random integers $m_B, n_B \in_R \mathbb{Z}_{l_B^{e_B}}$, not both divisible by l_B, and computes the isogeny ϕ_B, curve E_B and points $\phi_B(P_A), \phi_B(Q_A)$ as in the key-exchange scheme. He also computes the isogeny ϕ'_B and the resulting curve E_{BA} and its j-invariant $\mathrm{j}(E_{BA})$. He then computes

$$c = H_k(\mathrm{j}(E_{BA})) \oplus m$$

and sends $(E_B, \phi_B(P_A), \phi_B(Q_A), c)$ to Alice.

Decryption Upon receiving ciphertext $(E_B, \phi_B(P_A), \phi_B(Q_A), c)$, Alice computes the isogeny ϕ'_A and the resulting curve E_{AB} and its j-invariant. The decrypted plaintext is

$$m = H_k(\mathrm{j}(E_{AB})) \oplus c.$$

3.3.4 Current State of Security

The authors of SIDH conjectured that the problem of finding the resulting j-invariant of the curve E_{AB} as described in the key-exchange protocol when given only $(E_A, \phi_A(P_B), \phi_A(Q_B), E_B, \phi_B(P_A), \phi_B(Q_A))$ is computationally infeasible, in the sense that for any polynomial-time solver algorithm, the advantage of the algorithm is negligible of the security parameter $\log p$.

The fastest known algorithm for finding isogenies between supersingular elliptic curves is a quantum algorithm with a running time $\mathcal{O}(p^{1/4})$ [BJS14], where p is the characteristic of the base field. Since SIDH is a special case of this problem, where the base field is $\mathbb{F}_{p^2}$ and some parameters are known, the problem can be viewed as an instance of the *claw problem*, which has optimal asymptotic quantum complexity $\mathcal{O}(p^{1/6})$ [Tan07, Zha05].

However, a very powerful attack was found on the public-key encryption scheme [GPST16], which also attacks the key-exchange scheme where one side uses static keys. This attack can be prevented with a countermeasure of checking if the sender generated his parameters honestly, for which the protocol has to be changed.

4. Implementation

We shall now describe the structure of the implemented library, explain how the implemented protocol works and how the key-exchange and authentication sub-protocols encode data, explain the interfaces of mathematical primitives and algorithms, and show how to install the library.

As stated in Introduction, the main interface of the library is similar to that of the `TLS` class of the Qt Cryptographic Architecture.

We have chosen C++ as the programming language for the implementation because of several reasons:

- Programs written in C and C++ have very few requirements on the underlying system, they require almost no runtime support and can be easily ported to embedded devices.
- In contrast to C, C++ does greatly simplify bothersome things like memory allocation and deallocation, and with operator overloading the usage of mathematical primitives becomes more readable.
- There are fast implementations available for C and C++ of some required algorithms that are out of scope of this thesis, such as the SHA-2 family of hash functions or the ChaCha20 cipher. Although the library does currently depend on the GNU Nettle library [Net] because of these primitives, it is possible to change or remove that dependency in the future, if such a need arises.

Note that the library is implemented in a way that makes it easy to create wrapper function calls for programs written in C, and it makes little use of those parts of the C++ STL library that are implemented in a shared binary (usually called `libstdc++.so`, at least on GNU systems). Therefore, if such a need comes, it is possible to remove even the dependency on the STL shared library, which can be a reasonable requirement for embedded programs.

4.1 Description of The Protocol

4.1.1 Handshake

We have decided to make the handshake protocol plaintext, because it seemed that debugging would be easier this way, and parsing it does not cost almost any extra computing power (at least compared to the actual cryptographic operations).

From software engineering point of view, the usage of text-based protocol is rationalized by the complexity of transferred data—a complicated binary protocol that would embrace all possible future changes therefore seems infeasible. On the contrary, the very simple raw-data-transfer protocol that is used ever since handshake is finished is not expected to grow, and is designed as binary for efficiency reasons. This decision can be also viewed as a subtle instance of the popularized Rule of Transparency, as defined e.g. in The Art of Unix Programming [Ray03].

The handshake protocol uses Unix-style newline as line separator.

At the beginning, the client sends this message to the server:

```
Post-quantum hello v1, [server name].
Key-exchange: [KEX type]
Supported-ciphers: [space delimited list of supported ciphers]
Supported-MACs: [space delimited list of supported MACs]
Secret: [KEX data in base64]
Auth-type: [authentication type, optional]
Server-auth: [ID of server authentication key, optional]
Auth-request: [auth request data in base64, optional]
[empty line]
```

The fields `Server-auth` and `Auth-request` must be present if only if the field `Auth-type` is present.

If the server supports requested key-exchange type, authentication type (if present) and at least one of the ciphers and message authentication codes the client allows, it should respond with this message:

```
Post-quantum hello v1.
Key-exchange: [KEX type]
Supported-ciphers: [space delimited list of supported ciphers]
Supported-MACs: [space delimited list of supported MACs]
Secret: [KEX data, potentially in base64]
[empty line]
```

After this, both sides should perform the key-exchange scheme, and (if requested) the authentication scheme, and select one of the ciphers and MACs supported by other side, generate a random nonce which they will use as the key for the MAC selected and respond:

```
KEX: OK
Cipher: [cipher]
MAC: [mac]
Nonce: [base64 encoded nonce]
```

```
Auth-reply: [authentication reply data, optional]
[empty line]
```

At the end of this procedure, the key-exchange has produced secret key SK which is same for both sides. The peers have chosen MAC functions $M_s(d, n)$ and $M_c(d, n)$ and stream cipher functions $E_s(d, k, o)$, $E_c(d, k, o)$ for encryption and $D_s(d, k, o)$, $D_c(d, k, o)$ for decryption, where variables d, n, k and o represent data, nonce, key and offset, respectively.

Ephemeral Key Derivation Given secret key SK, ephemeral key of the client side is $EK_c = M_c(SK, n_c)$, where n_c is nonce generated by client and send to server, and ephemeral key of the server side is computed correspondingly, $EK_s = M_s(SK, n_s)$.

This is done so that the secret key can be forgotten immediately, thus ensuring weak perfect forward secrecy.

4.1.2 After Handshake

After a successful handshake, proper packets are transmitted. This part of the protocol is binary: before encryption, the first byte of each packet represents the packet type. Each packet m shall be authenticated by corresponding MAC to create an authenticated packet m_{auth}: $m_{\text{auth}} = m \mid M_p(m, n_p)$, where peer p is either s for server or c for client, and the symbol $\mid$ means concatenation. The authenticated packet is then encrypted into m_{enc}: $m_{\text{enc}} = E_p(m_{\text{auth}}, EK_p, o_p)$, where o_p is the stream cipher offset of peer p, which is 0 at beginning and incremented by the length of m_{auth} after this operation.

Close Packets Sent when one side wants to close its end of the connection. After this, the side cannot send packets anymore. The packet is one byte long:

$$m = \text{byte}(0).$$

Data Packets Each data packet can hold up to 64 KiB of data. The length is stored in a 32 bit unsigned integer in network byte order after the first byte which identifies the packet type:

$$m = \text{byte}(1) \mid \text{uint32}(\text{length}(\mathit{data})) \mid \mathit{data}.$$

Rekey Packets When a side wants to change the ephemeral key it uses for encryption, it shall generate new random nonce and send this information to the other side in a rekey packet. Length of the nonce is stored in an 8 bit unsigned integer after the packet identifying byte (thus nonce can be up to 255 bytes long):

$$m = \text{byte}(2) \mid \text{uint8}(\text{length}(\textit{nonce})) \mid \textit{nonce}.$$

4.1.3 Error Handling

If any error occurs during the handshake, or when parsing or verifying the authenticity of the packets, the library calls report an error status, corresponding error code can be retrieved by other API calls. Since the reported errors are not recoverable, the connection is then no longer considered to provide security, and the user is required to terminate it.

4.1.4 The `SIDHex` Key-Exchange Type

The `SIDHex` key-exchange type utilizes the SIDH key-exchange scheme as described in Section 3.3.2. The prime of the base field $p = 2^{372}3^{239} - 1$ is chosen as in the article of Costello et al. [CLN], so that $l_A^{e_A} \approx l_B^{e_B}$ and p is approximately 768 bits long, thus ensuring 128 bits of security (since, as mentioned in Section 3.3.4, a quantum algorithm can break SIDH in $\mathcal{O}(p^{1/6})$). The other public parameters can be viewed in the file `pqc_sidh_params.cpp`.

First we describe how data are encoded in `SIDHex`:

Encoding of Galois Field Elements An element $a + ib \in \mathbb{F}_{p^2}$ is encoded by its coefficients (a first), which are in turn encoded in network byte order, with each coefficient taking as many bytes as would be required for encoding the prime p. If $\overline{a}$ denotes the network byte order encoding of $a \in \mathbb{Z}_p$ and $\overline{a + ib}$ the encoding of $a + ib \in \mathbb{F}_{p^2}$, then

$$\overline{a + ib} = \overline{a} \mid \overline{b}.$$

Encoding of Elliptic Curves An elliptic curve $E_{a,b}$ with a short Weierstrass form $y^2 = x^3 + ax + b$, where $a, b \in \mathbb{F}_{p^2}$, is encoded as

$$\overline{E_{a,b}} = \overline{a} \mid \overline{b}.$$

Encoding of Points on a Curve A point $P \in E_{a,b}$ is encoded as

$$\overline{P} = \text{byte}(1) \mid \overline{P_x} \mid \overline{P_y}$$

if $O \neq P = (P_x, P_y)$, where $P_x, P_y \in \mathbb{F}_{p^2}$, and as a sequence of zero bytes of corresponding length if $P = O$.

The message which is sent in the `Secret` field of the handshake is

$$\text{base64enc}\left(\overline{E_X} \mid \overline{\phi_X(P_Y)} \mid \overline{\phi_X(Q_Y)}\right),$$

where $(X, Y) \in \{(A, B), (B, A)\}$.

4.1.5 The `SIDHex-sha512` Authentication Type

The general idea of authentication is based on public-key encryption: the peer S that wants to be authenticated generates a public/private key-pair. When another peer C wants to authenticate S, it simply encrypts his key-exchange commitment with S's public key, which enforces that only S can compute the shared secret key successfully. This is visualized in Figure 4.2. Thus a man-in-the-middle attack (shown in Figure 4.1) is prevented.

Because the general idea would require the implementation of algorithms able to encrypt messages of any length that also provide ciphertext indistinguishability, and our case is rather special in that we only need to enforce the key-exchange commitment to pass through the channel unchanged, we have developed a simpler protocol for authentication only, that does not encrypt the key-exchange commitment, only generates a challenge which only a receiver in possession of the private key is able to respond to correctly.

The SIDH parameters for the `SIDHex-sha512` authentication type are same as for the `SIDHex` key-exchange type. The randomly generated private parameters m, n are always of the form $(1, m)$ or $(m, 1)$, for $m \in \{0, \cdots, l^e\}$. The keys also always contain a 32 byte hash seed as the index k of the hash function H_k.

Private Key The first byte of the private key is byte(0) when the generated private parameters are of the form $(1, m)$ and byte(1) when of the form $(m, 1)$. The number m is then encoded in network byte order in as many bytes as would take to encode the number l^e. The 32 byte hash function index k follows.

$$\text{PrivKey} = \begin{cases} \text{byte}(0) \mid \overline{m} \mid \text{uint256}(k) & \text{if } (m, n) = (1, m) \\ \text{byte}(1) \mid \overline{m} \mid \text{uint256}(k) & \text{if } (m, n) = (m, 1). \end{cases}$$

Public Key The public key stores the curve E_A together with the points $\phi_A(P_B)$, $\phi_A(Q_B)$ and hash function index k.

$$\text{PubKey} = \overline{E_A} \mid \overline{\phi_A(P_B)} \mid \overline{\phi_A(Q_B)} \mid \text{uint256}(k).$$

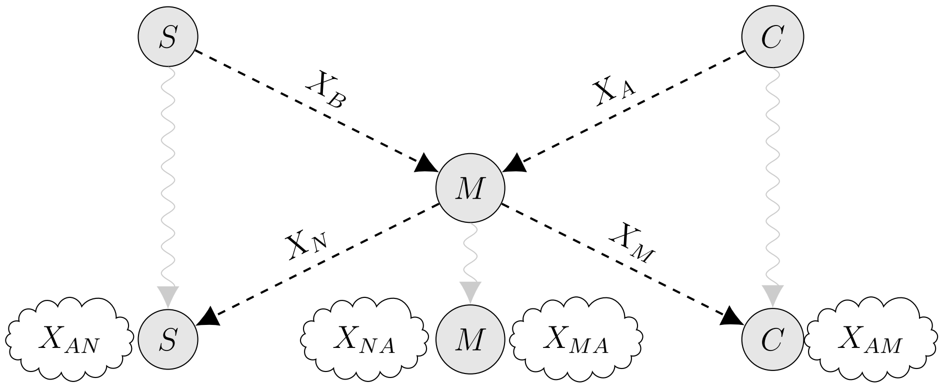

Figure 4.1: Man-in-the-middle attack, applicable to any Diffie-Hellman-style key-exchange.

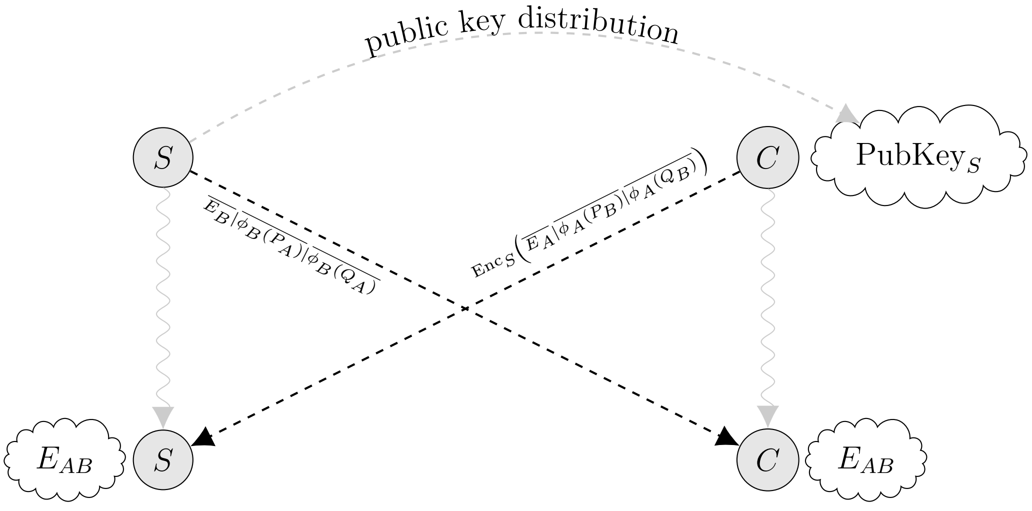
public key distribution
S
C
PubKey_S
$\overline{E_B|\phi_B(P_A)|\phi_B(Q_A)}$
$\text{Enc}_S\left(\overline{E_A|\phi_A(P_B)|\phi_A(Q_B)}\right)$
E_{AB}
S
C
E_{AB}

Figure 4.2: The general idea of authentication of SIDH key-exchange, based on public-key encryption. A potential man-in-the-middle cannot decrypt the commitment from C, therefore he is unable to compute the shared secret that must be used to communicate with C. The same scheme can also be used symmetrically, possibly on both sides.

The authentication process uses the `hmac_sha512` hash based message authentication code as the hash function H_k, where k is the 32 byte key value to the HMAC. The authentication consists of three parts:

Request When peer C wants to prove the authenticity of peer S, it follows the encryption procedure of the SIDH public-key encryption scheme given the public key of peer S, but ignores the ciphertext c. The request that is sent to S is:

$$\mathrm{Req} = \mathrm{base64enc}(\overline{E_B} \mid \overline{\phi_B(P_A)} \mid \overline{\phi_B(Q_A)}).$$

Sign Upon receiving message m and authentication request for message m from C (the message is the data from the `Secret` field of the handshake), peer S computes the signature of the request

$$\mathrm{Sign} = H_k\left(m \mid \overline{\mathrm{j}(E_{AB})}\right),$$

and then sends base64enc(Sign) back to peer C.

Verification Peer C also computes $H_k(m \mid \overline{\mathrm{j}(E_{AB}})$ and compares this value to the one received from peer S. If they differ, the verification is unsuccessful.

As mentioned in Section 3.3.4, there is a recently developed attack on the SIDH public-key encryption scheme. The authentication type we described in this Section is also vulnerable. Fortunately, a simple countermeasure does also already exist. We leave the implementation of this countermeasure for the future.

4.2 Library Internals

The logical structure of the library is shown on Figure 4.3. Here we describe library internals in more detail.

4.2.1 Mathematical Primitives

Big Integers The GNU GMP library [GMP] is used for computation with big integers. It has bindings for C++ implemented in a header file in such a way that when compiler optimizations are enabled, the emitted assembly is very efficient. We have extended the `mpz_class` representing big integers to class `Z`, adding some useful features such as serialization, modular multiplicative inverse or quadratic residues (most of

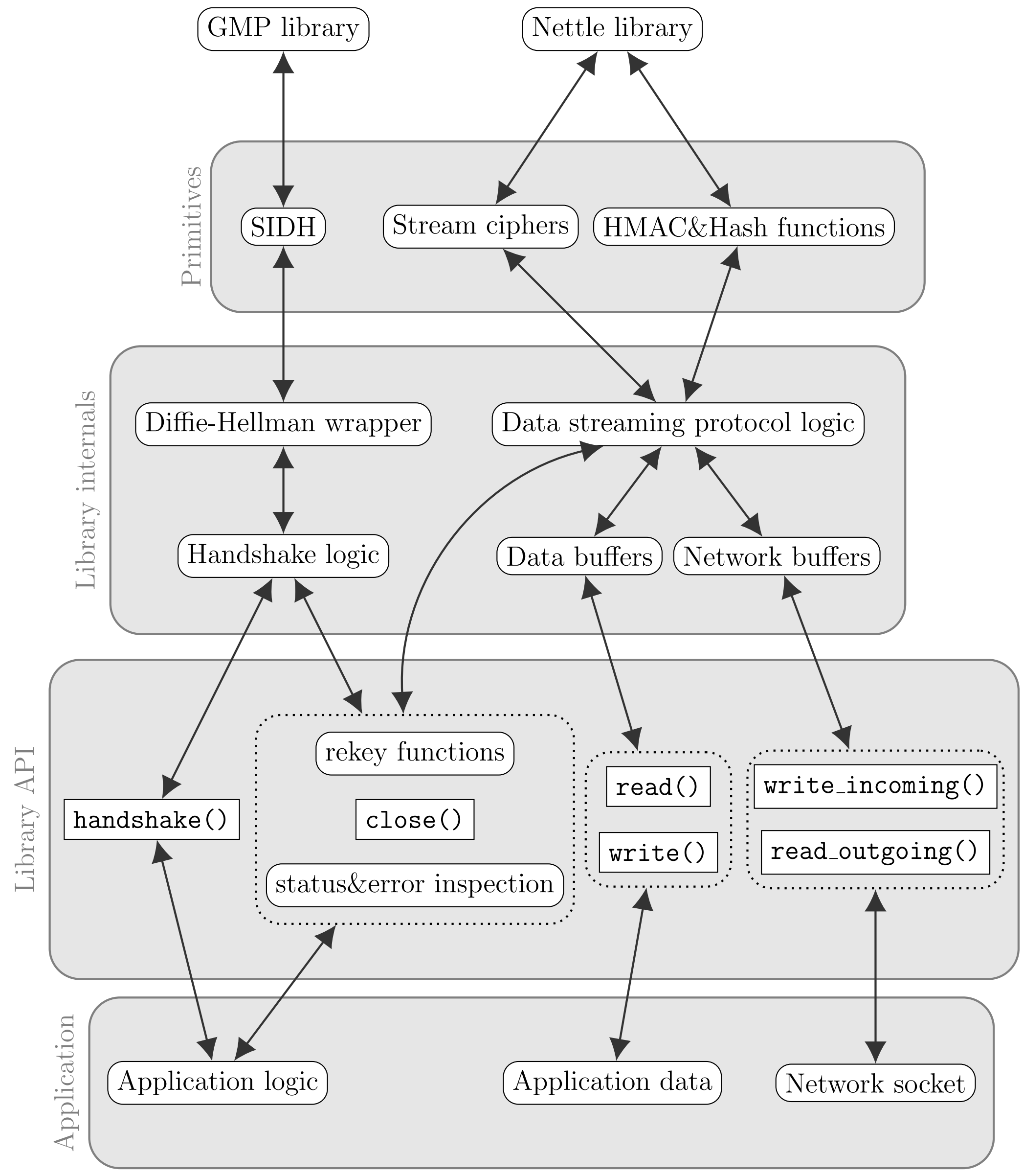

Figure 4.3: Overview of library structure by logical layers.

these are implemented in GMP, but are not accessible in a C++ way from `mpz_class`).

Galois Field Primitives The arithmetic in $\mathbb{F}_{p^2}$ as described in Section 1.2 is implemented in class `GF`.

It is important to note that the way C++ handles temporary results in compound expressions can result in a performance penalty—for example, given variables `x`, `y`, `z` of type `GF`, the expression `x = x*y + x*y*z` needs to construct at least two temporary variables and then destroy them. This can be rewritten to an expression `x *= x, x *= y, x *= z+1`, which only requires the construction of one temporary variable (`z+1`). It is possible to implement similar optimizations as `gmpxx.h` does, using builtin functions of the compiler, so that the need to create temporary variables would be minimized.

But if one wants to avoid the creation of temporary variables at all for such expressions, one would need to create named global temporary variables to store the temporary results and write all expressions in such a way that would avoid creating per-expression temporary variables.

This is done in the reference implementation [SSi], but we wanted to avoid this, for the same reason we preferred C++ over C: it is practically writing in assembly—there is not much difference between `x *= x, x *= y, tmp = z+1, x *= tmp` and

```
gf_mul (x, x, x);
gf_mul (x, x, y);
gf_add (tmp, z, gf_const(1));
gf_mul (x, x, tmp);
```

Elliptic Curve Primitives Elliptic curves in short Weierstrass form and points on them are represented by classes `WeierstrassCurve` and `WeierstrassPoint`, respectively. Operators for addition and negation are overloaded on the class `WeierstrassPoint` so that these operations correspond to the group law on the elliptic curve. The operator for multiplication implements scalar multiplication by the Montgomery ladder algorithm described in Algorithm 6.

Isogenies between elliptic curves are represented by the `WeierstrassSmallIsogeny` and `WeierstrassIsogeny` classes. The `WeierstrassSmallIsogeny` class computes, given isogeny generator and its degree d, the image of a given point in $\mathcal{O}(d)$ operations, while the `WeierstrassIsogeny` class represents isogenies of smooth degree—it computes the composition of small degree isogenies represented by the class `WeierstrassSmallIsogeny`. Because the naive implementation of the algorithm which computes a smooth degree isogeny would require $\mathcal{O}(n^2)$ operations in the base field, we use the optimal strategy algorithm described in Section 4.2.2 of the article by Jao et. al. [FJP11].

Random Number Generator Primitives The random number generator primitive, `random_bytes`, generates random bytes using the ChaCha20 algorithm, with a seed initialized from system entropy. We decided not to give the user multiple choices for the random number generator engine, since there seemed to be little advantage in it.

The function `random_z_below(Z p)` returns a random element of $\mathbb{Z}_p$, utilizing the generator from `random_bytes`.

4.2.2 Cryptographic Algorithm Abstractions

The implementations of stream cipher algorithms, message authentication codes, key-exchanges and authentication mechanisms are descended from abstract classes `cipher`, `mac`, `kex` and `auth`, respectively. These types define the interfaces for each class of cryptographic algorithm, and their descendants must define, for example, the key size needed for them to work.

The cryptographic algorithms defined in this way must be cryptographically secure and resistant to all kinds of attacks.

At the time of writing this thesis, the software supports the `ChaCha20` cipher, the `hmac_sha256` and `hmac_sha512` message authentication codes, the `SIDHex` key-exchange algorithm and the `SIDHex-sha512` authentication mechanism.

4.2.3 The `session` Class Interface

In this Section we describe the inner workings of the `session` class, which is the main (and almost only) class with which programs intending to use `pqc` should work.

The `session` class represents one side of a communication session secured by `pqc`—both a server application and a client application use it to communicate with each other.

Here is a brief explanation of what the `session` class does when used properly:

Session Beginning At the beginning, the application must call the method `start_client` or `start_server`. These initialize internal variables and put handshake data into internal buffer in the case of a client session.

Handshake The application cannot receive or send any plaintext data until the session is handshaken. Instead, the application polls for data on the internal session buffers (via the `bytes_outgoing` method) or on the communication channel (when data are available, the application puts them into the session internal buffer using the `write_incoming` method). After enough data for the handshake to complete are transferred between server and client, the shared secret key of the session is computed, from which the ephemeral keys are generated and the cipher and MAC instances are created and the session changes its state so that data can be sent and received.

Data Transfer When the application writes data with the `write` method, the data are divided into packets of 64 KiB, each packet is signed and encrypted and put into the outgoing buffer. The application shall then read the outgoing data and send them through the communication channel.

If the threshold for a rekey is reached (the threshold is set with the `set_rekey_after` method), a new nonce is generated and sent to the peer, and the ephemeral secret key for the cipher is recomputed.

The binary data written via `write_incoming` are put into packet reader, from which whole packets are retrieved whenever there are some. The data from the packets are then put into an internal buffer from which the application reads them with the `read` method.

In the case a rekey packet is received, the ephemeral secret key for cipher of the peer is recomputed with the nonce from the packet.

If a close packet comes, the state of the session is changed so that it will allow no more data from the peer.

Session Close To close the session, the application must call the `close` method. This will change the internal state of the session so that the application cannot write data anymore, and when the close packet is received from the peer, the session is closed permanently. No more data can be sent or received.

4.2.4 Other Classes

Here we briefly describe the other internal classes of the library, for developers interested in modifying the software.

`sidh_params` The set of parameters for the SIDH-based algorithm are represented by this class. At the time of writing this thesis, only one set of parameters is defined.

`sidh_key_basic` This class implements the primitives for the SIDH key-exchange algorithm. It either generates the private and public key-pairs and allows to export them, or can import the public key from another peer and generate shared secret.

`sidh_key` Extension of `sidh_key` which adds the index of the hash function used in the authentication protocol.

`packet` A packet, as described in Section 4.1.2, is represented by class `packet`. The buffer for the packet has to be given when constructing, in the form of a reference to a string and a position where the packet begins in that string. The data in the string may be changed when manipulating the packet.

`packet_reader` This class is used by the `session` class to store incoming data and decrypt them. Whenever a full packet is available in the decrypted data, it can be read and popped away from the reader.

`handshake` The parser of the handshake protocol is implemented here.

`chacha` Because the ChaCha20 cipher can only generate its stream in chunks of 64 bytes, we had to implement an interface with which it would be possible to encrypt/decrypt chunks of any length. If an incomplete chunk has to be used (when a string of length not divisible by 64 is to be processed), the rest is stored in an internal buffer to be used in the next processing.

4.3 Installation and Usage

4.3.1 Requirements

A C++14 capable compiler is needed for the `pqc` library to compile. Any recent versions of the GNU Compiler Collection [GCC] or Clang [cla] should be sufficient.

The library uses GNU Multiple Precision Arithmetic Library [GMP] for computations with big integers. All GNU-based systems are likely to have GMP in the software repository, and probably already installed (the GCC compiler does depend on the GMP library, for example).

The Nettle library [Net] is also required for some cryptographic primitives. This dependency will be probably optional in the future.

GNU Make [Mak] is required when building the library.

We have successfully compiled and tested the library on several Linux systems (Gentoo and Debian-testing distribution) and on FreeBSD.

4.3.2 Installation

After retrieving the sources of the library from Attachment A, or after cloning the git [git] repository from the URL `https://github.com/elkablo/pqc`, run:

```
$ gmake
$ gmake install
```

This will compile the sources with GCC and install the binaries and headers into the directory `/usr/local`. Another installation directory can be specified via the `PREFIX` variable. On FreeBSD, it is also needed to use the Clang compiler instead of GCC—this is done by specifying `CXX=clang++`.

This will also install the example programs: the SSH-like client called `pqc-telnet` together with a server called `pqc-telnetd` and a key-generation tool `pqc-keygen`.

4.3.3 Usage of the Example Programs

Here we explain how to use the bundled `pqc-telnet` program. At first, a key-pair for server authentication has to be create:

```
$ pqc-keygen SIDHex-sha512 server.priv server.pub
```

After this, we can run the server on TCP port 8822 with private authentication key `server.priv`:

```
$ pqc-telnetd server.priv 8822
```

To open a shell prompt on the machine with IP address 10.20.30.40 where the server is running we execute the client program:

```
$ pqc-telnet server.pub 10.20.30.40 8822
sh-4.3$
```

4.4 Evaluation of the Software

We have performed several tests with the example programs. All benchmarks presented in this Section were measured on Intel i7-3520M clocked at 2.90GHz.

The example `pqc-telnet` program required 792 milliseconds of CPU time on the client side and 552 milliseconds of CPU time on the server side, on average, to perform the handshake (which consists of key-exchange and peer authentication) and telnet session negotiation and establishment.

The overhead of raw data transfer with the `pqc-telnet` program was measured to be insignificant in comparison to the handshake. Transferring of 1 MiB of data to a remote machine took 830 milliseconds on the client side and 642 milliseconds on the server side, on average.

4.4.1 Comparison with other SIDH Implementations

We have compared the speed of our isogeny computation code to that of the SIDH Library. Other implementations either use the SIDH Library (Open Quantum Safe), or are difficult to use in an lightweight environment (the implementation from the authors of SIDH requires SageMath [sag], which is a very complex software used for research).

The comparison is shown in table 4.1. The `pqc` implementation is, as expected, roughly 10 times slower than the optimized version of the SIDH Library, since it represents elliptic curves in Weierstrass form, which requires more operations on big integers than on Montgomery curves. It is interesting to note that if the optimizations specific for the x86 platform are turned off in the SIDH Library, our implementation is only around 20% slower in average. This leads us to believe that the implementation of Montgomery curves in our library could make it outperform the SIDH Library on non-x86 platforms by a significant margin.

	`pqc` library	SIDH Library generic	SIDH Library assembly
keygen A	206	161	20
keygen B	204	189	24
final A	218	149	19
final B	222	177	23

Table 4.1: Comparison of various SIDH key-exchange operations with other software. All values are in milliseconds. Timings for key generation and shared secret final computation are compared for both sides of the key-exchange, since the sides compute isogenies of different degrees.

Conclusion

We have achieved the stated goal by implementing a new software library for real-time socket-like communication, which provides post-quantum alternative to the TLS protocol and utilizes the SIDH key-exchange and public-key encryption schemes. The implemented protocol has been designed for simplicity—our library is implemented in less than 6000 lines of code. As a result, it should be easily auditable for security vulnerabilities. The handshake part of the protocol is designed to be extensible, to be able to accommodate changes possibly required for future development.

There are several example programs that demonstrate the functionality, including a post-quantum SSH-like client program together with a corresponding server, that can already be used in practical environment. We have also implemented a tiny tool for generating the pairs of asymmetric keys that are used for authentication.

A description of the inner workings of the library, together with enough programmer-targeted information necessary for its proper usage, is included in the thesis.

The theory behind SIDH required for understanding the trapdoor function used by the implementation, including the required introduction to Galois field and elliptic curve theory, are given in the first chapters of the thesis. For maintaining brevity, we have sometimes been forced into deciding what parts of the rather complex theory (considering the scope of this thesis) we should elude or explain less formally. In such cases, we refer to literature that contains full formal explanation.

The process of implementing a secure authentication mechanism from the SIDH public-key encryption scheme has been somewhat hindered by recent development of a new attack, as described in the last paragraph of Section 3.3.4. This attack allows a potential man-in-the-middle to gradually compromise the authenticity of the connection, if he can actively interfere with the handshake. There are, fortunately, several methods that can prevent the attack.

Further Development

There are several viable starting points for future research and work on the library:

- The most pressing task for future development is to implement the countermeasure for the attack mentioned above. One such countermeasure is described in the presentation of Kirkwood et al. [KLM+15].

- More key-exchange and authentication schemes should also be added—for example, a SIDH-based scheme which works with elliptic curves in Montgomery form, like the implementation from the authors of SIDH does, because the arithmetic on Montgomery curves requires less long-integer operations.
- An API accessible to programs written in the C language would also come handy in the future.
- An implementation of a public-key infrastructure using certificate authorities could allow an almost seamless integration into existing software (e.g. Web browsers).

Bibliography

[BHH$^+$15] 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 *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 368–397. Springer, 2015.

[BJS14] Jean-François Biasse, David Jao, and Anirudh Sankar. *A Quantum Algorithm for Computing Isogenies between Supersingular Elliptic Curves*, pages 428–442. Springer International Publishing, Cham, 2014.

[Bor] `https://www.chromium.org/Home/chromium-security/boringssl`. Accessed: 2016-12-16.

[Brö09] Reinier Bröker. Constructing supersingular elliptic curves. *J. Comb. Number Theory*, 1(3):269–273, 2009.

[cla] `http://clang.llvm.org/`. Accessed: 2016-12-16.

[CLN] Craig Costello, Patrick Longa, and Michael Naehrig. Efficient algorithms for supersingular isogeny Diffie-Hellman. In *Advances in Cryptology*. Springer.

[Con96] Ian Connell. Elliptic Curve Handbook. `http://www.math.mcgill.ca/connell/`, 1996.

[Cub] Cubic Function. `https://en.wikipedia.org/wiki/Cubic_function`. Accessed: 2016-12-16.

[DB14] Nikesh S. Dattani and Nathaniel Bryans. Quantum factorization of 56153 with only 4 qubits, 2014.

[DR08] Tim Dierks and Eric Rescorla. The Transport Layer Security (TLS) Protocol, Version 1.2. `https://tools.ietf.org/html/rfc5246`, August 2008. Accessed: 2016-12-16.

[Euc] `https://en.wikipedia.org/wiki/Extended_Euclidean_algorithm`. Accessed: 2016-12-16.

[FJP11] Luca De Feo, David Jao, and Jérôme Plût. Towards quantum-resistant cryptosystems from supersingular elliptic curve isogenies. Cryptology ePrint Archive, Report 2011/506, 2011. http://eprint.iacr.org/2011/506.

[GCC] http://gcc.gnu.org/. Accessed: 2016-12-16.

[git] https://git-scm.com/. Accessed: 2016-12-16.

[GMP] https://gmplib.org/. Accessed: 2016-12-16.

[GPST16] 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*, pages 63–91. Springer, 2016.

[Jud13] Thomas W. Judson. *Abstract Algebra: Theory and Applications.* 2013. Accessed: 2016-12-16.

[KLM$^+$15] 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. 2015.

[Lib] https://en.wikipedia.org/wiki/LibreSSL. Accessed: 2016-12-16.

[Luc14] Michael Lucas. LibreSSL at BSDCan. http://blather.michaelwlucas.com/archives/2071, May 17 2014. Accessed: 2016-12-16.

[Mak] https://www.gnu.org/software/make/. Accessed: 2016-12-16.

[MSR] https://www.microsoft.com/en-us/research/project/sidh-library/. Accessed: 2016-12-16.

[Net] http://www.lysator.liu.se/~nisse/nettle/. Accessed: 2016-12-16.

[Ope] https://marc.info/?l=openbsd-cvs&m=139715677231774. Accessed: 2016-12-16.

[OQSa] https://openquantumsafe.org/. Accessed: 2016-12-16.

[OQSb] `https://github.com/open-quantum-safe/openssl/`. Accessed: 2016-12-16.

[Oss] `https://github.com/xoloki/openssl-sidh`. Accessed: 2016-12-16.

[PZ03] John Proos and Christof Zalka. Shor's discrete logarithm quantum algorithm for elliptic curves. 2003.

[QCA] Qt Cryptographic Architecture. `http://delta.affinix.com/docs/qca/index.html`. Accessed: 2016-12-16.

[Ram] `http://opensslrampage.org/`. Accessed: 2016-12-16.

[Ray03] Eric S. Raymond. *The Art of Unix Programming*. Addison-Wesley, 2003.

[sag] `http://www.sagemath.org/`. Accessed: 2016-12-16.

[Sho95] Peter W. Shor. Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer. 1995.

[Shu09] Daniel Shumow. Isogenies of Elliptic Curves: A Computational Approach. *CoRR*, abs/0910.5370, 2009.

[Sil92] Joseph H. Silverman. *The arithmetic of elliptic curves*, volume 106 of *Graduate Texts in Mathematics*. Springer-Verlag, New York, 1992. Corrected reprint of the 1986 original.

[Sin15] Vikram Singh. A Practical Key Exchange for the Internet using Lattice Cryptography. Cryptology ePrint Archive, Report 2015/138, 2015. `http://eprint.iacr.org/2015/138`.

[SSi] `https://github.com/defeo/ss-isogeny-software`. Accessed: 2016-12-16.

[Tan07] Seiichiro Tani. Claw Finding Algorithms Using Quantum Walk. 2007.

[TLS] Transport Layer Security. `https://en.wikipedia.org/wiki/Transport_Layer_Security`. Accessed: 2016-12-16.

[TRe] `https://is.cuni.cz/webapps/zzp/`. Accessed: 2016-12-16.

[Zha05] Shengyu Zhang. Promised and distributed quantum search. In *International Computing and Combinatorics Conference*, pages 430–439. Springer, 2005.

List of Algorithms

List of Figures

List of Tables

Attachments

Attachment A - the Enclosed CD

On the CD attached to this thesis (and on the online Thesis Repository of Charles University [TRe]) we enclose the source codes of the implemented software together with the source codes of its dependencies (GNU Nettle and GMP), and also the code of the SIDH Library, so it can be used for comparison when benchmarking.

The electronic version of this thesis is also enclosed.