Title Goes Here

Submitted by

Robert W.V. Gorrie
B.ASc. Computer Science (McMaster University)

Under the guidance of
Douglas Stebila

*Submitted in partial fulfillment of
the requirements for the award of the degree of*

**Masters of Science
in
Computer Science**

Department of Computing and Software
McMaster University
Hamilton, Ontario, Canada

Fall Semester 2017

Abstract

¡Abstract here¿

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

1.1.1 Literature Survey

1.2 Contributions

1.3 Structure

1.3.1 Layout

Over the course of the past decade, elliptic curve cryptography (ECC) has proven itself a mainstay in the wide world of applied cryptology. While isogeny based cryptography does build itself up from the same underlying field of mathematics as ECC, it simultaneously draws from a slightly more complicated space of algebraic notions. Much of this chapter will be dedicated to illuminating these notions in a manner that should be digestable for those without serious background in algebraic geometry, or abstract algebra in general.

1.3.2 Notation

Chapter 2

Technical Background

This chapter will cover the following preliminary topics: cryptographic primitives, isogenies and their relevant properties, supersingular isogeny Diffie-Hellman (SIDH), the Fiat-Shamir construction for digital signatures (and its quantum-safe adaptation), the current landscape of isogeny based signature schemes, and finally the C implementations of isogeny based protocols with which we are concerned.

In the first section of this chapter we will take some time to introduce a few ideas from modern cryptography. We will cover key exchange, identification schemes, and signature schemes - all at as high of an abstraction level as possible. Readers familiar with these topics can skip this section without harm.

Our discussion of isogenies will begin with some basic coverage of the underlying algebra. We will provide the material necessary for the remaining sections as we build up in the level of abstraction; working our way through groups, finite fields, elliptic curves, and finally isogenies and their properties.

Once we have presented the necessary algebra, we will illustrate the specifics of the supersingular isogeny Diffie-Hellman key-exchange protocol. We will spend most of this time dedicated to a modular deconstruction of the protocol, looking at the high-level procedures and algorithms which will be necessary for understanding in detail the signature protocol to come. This subsection will end with a briefing and analysis of the closely related zero-knowledge proof of identity (ZKPoI) isogeny protocol proposed in the original De Feo et al. paper [FJP12], as it is the foundation for the isogeny based signature scheme presented by Yoo et al [YAJ$^+$12].

In section 2.3 we will discuss the Fiat-Shamir transformation [Kat10]; a technique which, given a secure interactive identification scheme, creates a secure digital signature scheme. We will also look at the quantum-secure adaptation published by Unruh [Unr15] for applying a non-quantum-resistant transform to a quantum-resistant primitive would be rather frivolous.

Section 2.4 will be dedicated to covering current isogeny-based signature schemes - the topic about which this dissertation is mainly concerned. We will discuss the signature scheme of Yoo et al., which is a near direct application of Unruh's work to the SIDH zero-knowledge proof of identity.

Finally, the last section of this chapter will introduce the SIDH C library released by Microsoft Research, on top of which the core contributions of this thesis are implemented. We will also look at the implementation of the to-be-discussed signature scheme, which is a proof-of-concept built on top of the Microsoft API.

2.1 Cryptographic Primitives

Cryptographic primitives can be thought of as the basic building blocks used in the design of cryptographically secure applications and protocols. The idea of which being that if individual primitives are provably (or believeably) secure, we can be more confident in the security of the application as a whole.[1]

To quickly recap some basic information security, there are serveral different security properties a cryptographic primitive may aim to offer:

- *Confidentiality*: The notion that the information in question is kept private from unauthorized individuals.
- *Integrity*: The notion that the information in question has not been altered by unauthorized individuals.
- *Availability*: The notion that the information in question is available to authorized individuals when requested.
- *Authenticity*: The notion that the source of the information in question is verified.
- *Non-repudiation*: The notion that the source of the information in question **cannot** deny having originally provided the information.

The security of a particular cryptographic primitive is measured by two components. The first, referred to often as a "security guarantee", measures what conditions constitute a successful attack on the primitive. The second, known as the "threat model", makes assumptions about the computational powers that the adversary holds. The best practice in forming security proofs is to aim for security with respect to the most easily broken security guarantee and the most challenging possible threat model. The combination of a security guarantee and threat model is known as a *security goal.*

Each of the primitives to come are designed to offer some utility in the communication between a given pair of entities. We will refer to these entities as Alice and Bob. The schemes we are concerned with in this dissertation are strictly *public key* (also known as *asymmetric key*) schemes. In public key primitives, each user possesses a *public* key (visible to every user in the network) as well as a *private* key, which only they have access to.

The first class of primitives we will discuss, *key exchange* protocols, provide a means by which Alice and Bob can come to the agreement of some secret value. The goal of a key exchange protocol is for Alice and Bob, communicating over some open, insecure channel, to reach mutual agreement of the secret value while also ensuring the *confidentiality* of that value. The secret value is referred to as a *secret* or *shared* key and is intended for use in other cryptographic primitives.

Identification schemes are a class of primitives that aim to ensure *authenticity* of a given entity. If Alice is communicating with Bob and she wants to verify that Bob is who he claims to be, the two can utilize a secure identification scheme. After identification protocols we will look at signature schemes, which are somewhat of an extension of the former. Signature schemes aim to provide *authenticity* on every message sent from Bob

[1]This is not to say that software which implements provably secure primitives is guaranteed to be secure. In security, it should be expected that the weakest link in the system is the first to be exploited, and these weak links often lie in careless implementation details.

to Alice, as well as *non-repudiability* and *integrity* of those messages.

Random Oracle Model. Before continuing with our discussion of primitives, it is worth discussion a framework in cryptography known as the random oracle model. A "random oracle" is a theoretical black box which, for every unique input, responds with a truly random output. That is, if a query is made to a random oracle h with input x (written $h(x)$) multiple times, h with respond with the same (random) output every time.

For certain constructions to be proven secure, it is sometimes necessary or helpful to assume the existence of random oracles. While this assumption may seem greviously optimistic, *hash functions* are a widely diployed family of functions which are believed to approach the nature of random oracles to some degree. Much of the security of modern cryptography depends on the security of such hash functions.

2.1.1 Key Exchange

A key exchange protocol, which we will denote as Π_{kex}, can be represented in some contexts by a pair of polynomial time algorithms **KeyGen** and **SecAgr**: $\Pi_{kex} = (\mathbf{KeyGen}, \mathbf{SecAgr})$. Alice and Bob will each run both of these procedures. The first they will run on the same input, 1^λ, a bit string of λ 1's. The second, short for "secret agreement", they will run on both their outputs of **KeyGen** and their peers.

Execution of Π_{kex} between Alice and Bob involves the following:

(i) Alice and Bob run $\mathbf{KeyGen}(1^\lambda)$: A probabilistic algorithm with input 1^λ and output (sk, pk). Typically pk is the image of $f(sk)$, where f is some *one-way* function. We will denote the outputs of **KeyGen** for Alice and Bob as $(sk_{\text{Alice}}, pk_{\text{Alice}})$ and $(sk_{\text{Bob}}, pk_{\text{Bob}})$ respectively.

(ii) Alice and Bob exchange (over an insecure channel) their public keys pk_{Alice} and pk_{Bob}.

(iii) Alice runs $\mathbf{SecAgr}(sk_{\text{Alice}}, pk_{\text{Bob}})$: A deterministic algorithm with input sk_{Alice} and pk_{Bob} and output $k_{\text{Alice}} \in \{0,1\}^\lambda$. Bob runs $\mathbf{SecAgr}(sk_{\text{Bob}}, pk_{\text{Alice}})$ to obtain $k_{\text{Bob}} \in \{0,1\}^\lambda$.

Π_{kex} is said to uphold *correctness* if $k_{Alice} = k_{Bob}$ for all honestly derived keypairs $(sk_{\text{Alice}}, pk_{\text{Alice}})$ and $(sk_{\text{Bob}}, pk_{\text{Bob}})$. Because we deal only with correct Π_{kex}, we refer to the output of Π_{kex} as simply k. Figure 2.1 illustrates an execution of the Diffie-Hellman key exchange protocol which relies on the difficulty of the *discrete logarithm* problem for its one-way function f.

The security goal typical of a key exchange protocol is that an adversary with access to the session transcript (threat) cannot discern the resulting shared secret key from a randomly generated value (security guarantee).

2.1.2 Interactive Identification Schemes

Imagine Alice wishes to confirm the identity of Bob. The idea behind interactive identification protocols is to offer Bob some way of proving to Alice (or anyone) that he has knowledge of some secret which **only** Bob could possess. The goal, of course, being to

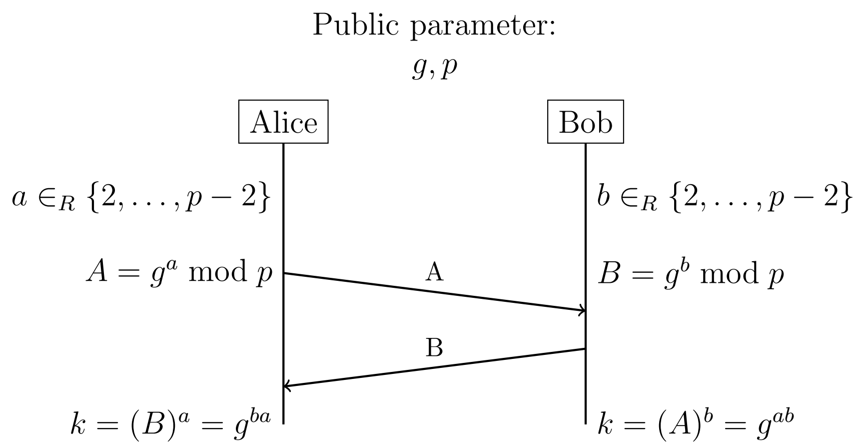

Figure 2.1: Alice and Bob's execution of Diffie-Hellman key exchange.

accomplish this without openly revealing the secret, so that it can continue to be used as an identifier for Bob.

An identification scheme (or otherwise "proof of identity") Π_{id} is composed of by the tuple of polynomial-time algorithms (**KeyGen**, **Prove**, **Verify**). Π_{id} is an interactive protocol, wherein the *prover* (Bob, for example) executes **Prove** and the *verifier* (Alice) executes **Verify**.

Execution of Π_{id} between Alice and Bob involves the following:

(i) Bob runs **KeyGen**(1^λ): A probabilistic algorithm with input 1^λ and output (sk, pk).

(ii) Bob sends to Alice his public key pk and a probabilistically generated initial commitment *com*. Alice will respond with a *challenge* value *ch*.

(iii) Bob runs **Prove**(sk, com, ch): A deterministic algorithm with input sk (Bob's secret key) and ch (challenge) and output *resp*.

(iv) Alice runs **Verify**$(pk, com, ch, resp)$: A deterministic algorithm with input pk (Bob's public key), com, ch, and $resp$ and output $b \in 0, 1$. Bob has successfully proven his identity to Alice if $b = 1$.

If Alice accepts Bob's response, and $b = 1$, then we refer to the tuple $(com, ch, resp)$ as an *accepting transcript*. In terms of security, it is common to show that an identification scheme is secure against *impersonation* under a *passive attack*. Proving such security implies that an adversary who eavesdrops on arbitrarily many executions of Π_{id} between a verifier $\mathcal{V}$ and a prover $\mathcal{P}$ cannot successfully impersonate $\mathcal{V}$.

There exist variations upon this type of primitive wherein Alice is not required to send Bob a specific challenge value. These are known as *non-interactive* identification schemes, or non-interactive proofs of identity (NIPoI). These non-interactive approaches to solving the problem of identity and *authentication* further bridge the gap between identification protocols and signature schemes.

2.1.3 Signature Schemes

We define a signature scheme as the tuple of algorithms $\Pi_{sig} = ($**KeyGen**, **Sign**, **Verify**$)$. Some execution of Π_{sig} between Alice and Bob for a particular message m sent from Bob to Alice involves the following... First, before any message is to be signed, the Bob must run the following:

- Bob runs **KeyGen**(1^λ): A probabilistic algorithm with input 1^λ and output (sk, pk).

Then, for every message m Bob wishes to authenticate and send to Alice:

(i) Bob sends his public key pk to Alice over an authenticated channel if he has not yet done so.

(ii) Bob runs **Sign**(sk, m): A probabilistic algorithm with input sk (Bob's secret key) and m (the message Bob wishes to authorize) and output σ, known as a *signature*.

(iii) Bob sends m and σ to Alice.

(iv) Alice runs **Verify**(pk, m, σ): A deterministic algorithm with input pk (Bob's public key), m, and σ and output $b \in \{0, 1\}$. Alice has confidence in the *integrity* and origin *authenticity* of m if $b = 1$.

As previously alluded to, it is worth noting that signature protocols and identification schemes are closely related. In essence, they are rather similar; but with two main differences. The first is rather comparable to the aforementioned difference between interactive identification schemes and non-interactive identification schemes. The second arises as a result of aiming to authenticate Bob on any particular message m. To achieve this, the signature scheme needs to be run every time Bob wishes to send a message to Alice. The details of this comparison are intentionally left vague, as it will from a topic of close inspection in section 2.4.

The strongest security goal for a signature scheme Π_{sig} is expressed as *existential unforgeability* under an *adaptive chosen-message attack*. If this goal is provably satisfied, an adversary with the ability to sign arbitrary messages will not be able to forge any conceivable and valid signature.

2.2 Algebraic Geometry & Isogenies

Groups & Varieties. A **group** is a 2-tuple composed of a set of elements and a corresponding group operation (also referred to as the group *law*). Given some group defined by the set G and the operation $\cdot$ (written as $(G, \cdot)$) it is typical to refer to the group simply as G. If $\cdot$ is equivalent to some rational mapping[2] $f_G : G \to G$, then the group $(G, \cdot)$ is said to form an **algebraic variety**. A group which is also an algebraic variety is referred to as an **algebraic group**.

G is said to be an *abelian* group if, in addition to the four traditional group axioms (closure, associativity, existence of an identity, existence of an inverse), G satisfies the condition of commutitiviy. More formally, for some group G with group operation $\cdot$, we say G is an abelian group iff $x \cdot y = y \cdot x \ \forall x, y \in G$. An algebraic group which is also abelian is referred to as an **abelian variety**.

> **Definition 1** (Abelian Variety)**.** for some algebraic group G with operation $\cdot$, we say G is an <u>abelian variety</u> iff $x \cdot y = y \cdot x \ \forall x, y \in G$.

[2] A rational map is a mapping between two groups which is defined by a polynomial function with rational coefficients.

For some group $(G, \cdot)$, some $x, y \in G$, and some rational mapping $f_G : G \to G$, let the following sequence of implications denote the classification of $(G, \cdot)$:

$$\text{group} \xRightarrow{x \cdot y = f_G(x,y)} \text{algebraic group} \xRightarrow{x \cdot y = y \cdot x} \text{abelian variety}$$

Morphisms. Let us again take for example some group $(G, \cdot)$. Let's also define some set $S_{(G,\cdot)}$ which contains every tuple (x, y, z) for group elements x, y, z which satisfy $x \cdot y = z$.

$$S_{(G,\cdot)} = \{x, y, z \in G | x \cdot y = z\}$$

Take also for example a second group $(H, *)$ and some map $\phi : G \to H$. ϕ is said to be *structure preserving* if the following implication holds:

$$(x, y, z) \in S_{(G,\cdot)} \Rightarrow (\phi(x), \phi(y), \phi(z)) \in S_{(H,*)}$$

A **morphism** is simply the most general notion of a structure preserving map. More specifically, in the domain of algebraic geometry, we will be dealing with the notion of a **group homomorphism**, defined as follows:

Definition 2 (Group Homomorphism)**.** For two groups G and H with respective group operations $\cdot$ and $*$, a group homomorphism is a structure preserving map $h : G \to H$ such that $\forall u, v \in G$ the following holds:

$$h(u \cdot v) = h(u) * h(v)$$

From this simple definition, two more properties of homomorphisms are easily deducible. Namely, for some homomorphism $h : G \to H$, the following properties hold:

- h maps the identity element of G onto the identity element of H, and
- $h(u^{-1}) = h(u)^{-1}, \forall u \in G$

Recall that for some morphism (or function) $h : G \to H$, we refer to G as the domain and H as the codomain.

Furthermore, an *endomorphism* is a special type of morphism in which the domain and the codomain are the same groups. We denote the set of enomorphisms defineable over some group G as $\text{End}(G)$. The *kernel* of a particular homomorphism $h : G \to H$ is the set of elements in G that, when applied to h, map to the identity element of H. We write this set as $ker(h)$, and it is much analogous to the familiar concept from linear algebra, wherein the kernel denotes the set of elements mapped to the zero vector by some linear map.

2.2.1 Fields & Field Extensions

A **field** is a mathematical structure which, while being similar to a group, demands additional properties. Fields are defined by some set F and two operations: *addition* and *multiplication*. In order for some tuple $(F, +, \cdot)$ to constitute a field, it must satisfy an assortment of axioms:

Addition axioms:

- (closure) If $x \in F$ and $y \in F$, then $x + y \in F$.
- $+$ is commutative.
- $+$ is associative.
- F contains an element 0 such that $\forall x \in F$ we have $0 + x = x$.
- $\forall x \in F$ there is a corresponding element $-x \in F$ such that $x + (-x) = 0$.

Multiplication axioms:

- (closure) If $x \in F$ and $y \in F$, then $x \cdot y \in F$.
- $\cdot$ is commutative.
- $\cdot$ is associative.
- F contains an element $1 \neq 0$ such that $\forall x \in F$ we have $x \cdot 1 = x$.
- $\forall x \neq 0 \in F$ there is a corresponding element $x^{-1} \in F$ such that $x \cdot (x^{-1}) = 1$.

Additionally, a field $(F, +, \cdot)$ must uphold the *distributive law*, namely:

$$x \cdot (y + z) = x \cdot y + x \cdot y \text{ holds } \forall x, y, z \in F$$

While these axioms are known to be satisfied by the sets $\mathbb{Q}$, $\mathbb{R}$, and $\mathbb{C}$ with typically defined $+$ and $\cdot$, our focus will be on a particular class of field known as a *finite field.* Finite fields, as the name suggests, are fields in which the set F contains finitely many elements - we refer to the number of elements in F as the *order* of the field.

Let us take some prime number p. We can construct a finite field by taking F as the set of numbers $\{0, 1, ...p - 1\}$ and defining $+$ and $\cdot$ as addition and multiplication *modulo* p. Finite fields defined in this fashion are denoted as $\mathbb{F}_p$, and have order p.

$$\forall x, y \in \mathbb{F}_p,\ x + y = (x + b) \mod p, \text{ and}$$
$$\forall x, y \in \mathbb{F}_p,\ x \cdot y = (x \cdot b) \mod p$$

For any given field K there exists a number q such that, for every $x \in K$, adding x to itself q times results in the additive identity 0. This number is referred to as the *characteristic* of K, for which we write $\text{char}(K)$. Finite fields are the only type of field for which $\text{char}(K) > 0$. Furthermore, if the field in question is finite and has prime order, then the order and the characteristic are equivalent.

A particular field K' is called an *extension field* of some other field K if $K \subseteq K'$. The complex numbers $\mathbb{C}$, for example, are an extension field of $\mathbb{R}$. A given field K is *algebraically closed* if there exists a root for every non-constant polynomial defined over K. If K itself is not algebraically closed, we denote the extension of K that is by $\overline{K}$.

An algebraic group G_a is defined over a field K if each element $e \in G_a$ is also an element of the field K, and the corresponding f_{G_a} is defined over K. To show that a particular algebraic group G_a is defined over some field K we will henceforth denote the group/field pairing as $G_a(K)$. Naturally, in the case where our field is a finite field of order p, we write $G_a(\mathbb{F}_p)$.

These algebraic structures are all important for building up to the concept of an *isogeny.* The lowest-level object we will be concerned with when discussing the forthcoming isogeny-based protocols will typically be elements of abelian varieties. The lowest-level structure in the SIDH C codebase is a finite field element.

Montgomery Arithmetic. We will now briefly discuss a technique for efficiently performing modular arithmetic. This method is widely deployed in cryptosystems centered around finite fields, and is abundantly used in the $\texttt{SIDH}_{\texttt{C}}$ library that we will shortly be examining.

In 1985, Peter Montgomery introduced a method for efficiently computing the modular multiplication of two elements a and b. The technique begins with the construction of some constant R, whose value depends solely on the modulus N and the underlying computer architecture.[3]

With the retrieval of R, $aR \mod N$ and $bR \mod N$ are constructed and referred to as the Montgomery *representation* of a and b respectively. Montgomery multiplication outlines an algorithm for computing $abR \mod N$ (the Montgomery product of a and b), from which $ab \mod N$ can be recovered through conversion back to standard representation. Once in Montgomery representation, other arithmetic can be performed (including field element inversions) in order to leverage the performance improvement offered by Montgomery modular multiplication – converting back to regular representation when necessary.

Applying Montgomery multiplication has the added benefit of decreasing the amount of field element inversions that need to be computed. Because of this, the technique is particularly relevant to this dissertation. We continue this discussion in section 3.2.

2.2.2 Elliptic Curves

An elliptic curve is an algebraic curve defined over some field K, the most general representation of which is given by

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

This representation encapsulates elliptic curves defined over any field. If, however, we are dicussing curves defined specifically over a field K such that $\text{char}(K) > 3$ (see [?]), then the more compact form $y^2 = x^3 + ax + b$ can be applied (see Figure 2.2 for a geometric visualization). In this dissertation we will default to this second representation, as the schemes with which we are concerned will always be defined over $\mathbb{F}_p$ for some large prime p.

We can define a group structure over the points of a given elliptic curve (or any other smooth cubic curve). If we wish to define a group in accordance to a particular curve, we do so with the following notation:

$$E : y^2 = x^3 + ax + b$$

Wherein E denotes the group in question, the elements of which are all the points (solutions) of the curve. Throughout much of this section, the words *point* and *element* can be used interchangeably.

The Group Law. The group operation we define for E, denoted $+$, is better understood geometrically than algebraically. Consider the following.

[3] The specifics of how R is constructed are beyond the scope of this dissertation; if they feel so inclined, the reader should refer to [Mon85]

Given two elements P and Q of some arbitrary elliptic curve group E, we define $+$ geometrically as follows: drawing the line L through points P and Q, we follow L to its third intersection on the curve (which is guaranteed to exist), which we will denote as $R = (x_R, y_R)$. We then set $P + Q = -R$, where $-R$ is the reflection of R over the x-axis: $(x_R, -y_R)$. This descriptive definition of $+$ is suitable for all situations *except* for when L is tangent to E or when L is parallel to the y-axis. These cases will be covered in a short moment. See Figure 2.2 for an illustrated representation of this process.

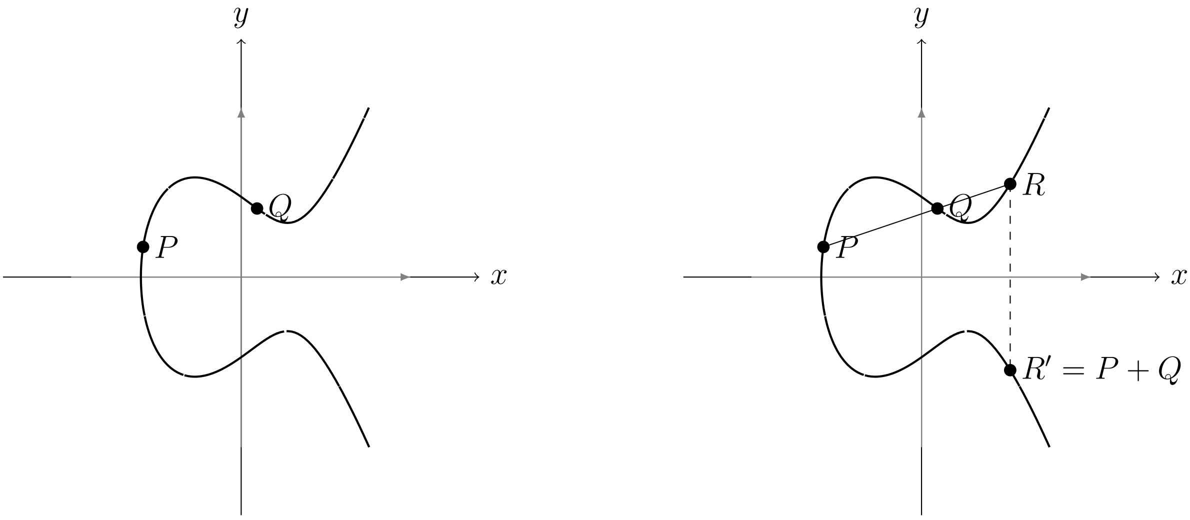

Figure 2.2: $+$ acting over points P and Q of $y^2 = x^3 - 2x + 2$.

The group operation $+$ is referred to as *pointwise addition*. In order for $(E, +)$ to properly form a group under pointwise addition, it must satisfy the four group axioms:

- *Closure*: Because elliptic curves are polynomials of degree of 3, we know any given line passing through two points P and Q of E will pass through a third point R. The exceptions here are twofold. First, when $P = Q$ and thus our line is tangent to E, and second, when $Q = -P$ and our line is parallel with the y-axis. We resolve the first case nicely by defining $P + P$ by means of taking L to be the line tangent to E at point P. In the second case, $P + (-P)$, by group axiom, should yield the identity element of the group. We will define this element and resolve this issue below.

- *Identity*: The identity element of elliptic curve groups, denoted as $\mathcal{O}$, is a specially defined point satisfying $P + \mathcal{O} = \mathcal{O} + P = P$, $\forall P \in E$. Because of the inclusion of this special element, we have that $\#(E(K))$ is equal to 1 + the number geometric points on E defined over K. This of course is only a noteworthy claim when K is a finite field (otherwise there are already infinitely many elements in E).

- *Associativity*: For all points P, Q, and R in E, it must be the case $((P+Q)+R = P+(Q+R))$ holds. It is rather easy to see visually why this is true for geometrically defined points in E (see Figure 2.3). Additionally, we can trivially show that this holds when any combination of P, Q, and R are $\mathcal{O}$ by applying the axiom of the identity.

- *Inverse*: Due to the x-symmetry of elliptic curves, every point $P = (x_P, y_P)$ of E has an associated point $-P = (x_P, -y_P)$. If we apply $+$ to P and $-P$, L assumes the line parrallel to the y-axis at $x = x_P$. As discussed above, in this case there is no third intersection of L on E. In light of this, $\mathcal{O}$ can be thought of as a point residing infinitely far in both the positive and negative directions of the y-axis. $\mathcal{O}$ is equivalently referred to as the *point at infinity* (see Figure 2.3).[4]

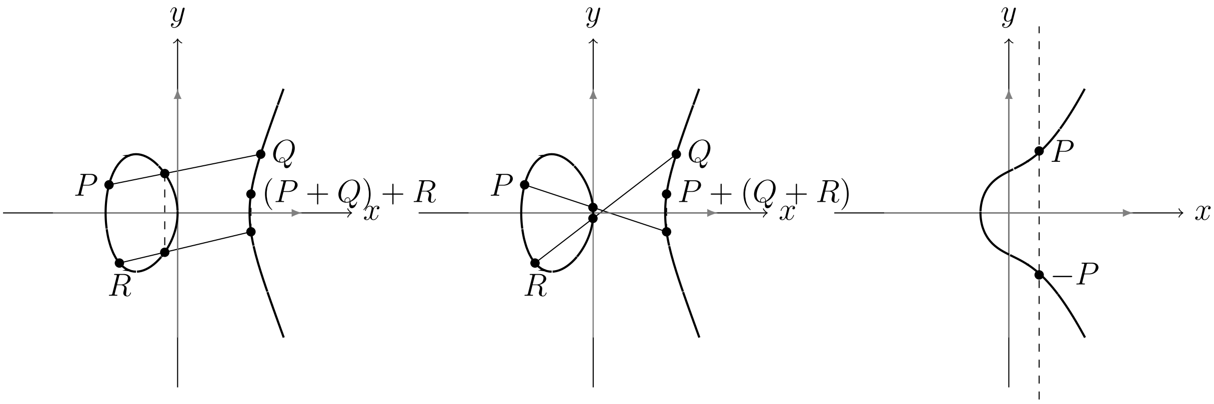

Figure 2.3: associativity illustrated on $y^2 = x^3 - 3x$ (left & center) and $P + (-P) = \mathcal{O}$ illustrated for $y^2 = x^3 + x + 1$ (right).

Of course, there are relatively simple formulas for algebraically defining point-wise addition and inverse computation. We have opted to describe these operations geometrically simply for ease of communication.

Additionally, we shorthand $\overbrace{P + P + ... + P}^{n}$ as nP, analogous to scalar multiplication.

Consequently, because groups defined over elliptic curves in this fashion are commutitive, they also constitute abelian varieties.

When referring to curves as abelian varieties defined over a field, we will write them as $E_\alpha(K)$, for some curve E_α and some field K. If we are only concerned with the geometric properties of the curve, or curves as distinct elements of some group structure, it will suffice to write E_α. Moving forward from here, we will assume all general curves discussed are capable of definition over some finite field $\mathbb{F}_p$.

The *r-torsion group* of E is the set of all points $P \in E(\overline{\mathbb{F}}_q)$ such that $rP = \mathcal{O}$. We denote the r-torsion group of some curve as $E[r]$.

Supersingular Curves. An elliptic curve can be either *ordinary* or *supersingular*. There are several equivalent ways of defining supersingular curves (and thus the distinction between them and ordinary curves) in a general setting, but each of these goes well beyond our scope. In the context of curves defined over finite fiels, however, the following succinct definition holds:

[4]One might suspect that the inclusion of this (apparently) non-algebraic element $\mathcal{O}$ suggests that $+$ is not a rational-map. The operator $+$ *can* be shown to be a rational-mapping if we define our elliptic curve groups in three-dimensional projective space.

Definition 3 (Supersingular Curve)**.** Let E be an elliptic curve defined over the finite field $\mathbb{F}_p$. E is said to be supersingular if $\#(E(\mathbb{F}_p)) = p + 1$.[a]

[a] Readers are welcomed to investigate [Cos] for further details.

For the remainder of this paper, unless otherwise noted, all elliptic curves in discussion will supersingular.

Projective Space. While elliptic curves are naturally defined in two-dimensional affine space, there are many benefits to expressing them through three-dimension projective coordinates. First and foremost, expressing curves in projective space allows us to reason geometrically about $\mathcal{O}$. This is done by defining a curve E such that it resides in some two-dimension subspace of 3-space, the point $\mathcal{O}$ then resides at some point in 3-space outside of the residing plane of E.

Representing a curve in 3-space requires some substitution of x and y coordinates, a typical forma for achieving this is the following:

$$x = X/Z \quad y = Y/Z \quad Z = 1$$

Such a representation of elliptic curve points offers more computationally efficient arithmetic over points. This is conceptually similar to the previously mentioned Montgomery arithmetic regarding finite field elements. Other substitutions offer different computational advantages, but the implementations we will discuss make use of this typical approach.

2.2.3 Isogenies & Their Properties

Definition 4 (Isogeny)**.** Let G and H be algebraic groups. An isogeny is a morphism $h : G \to H$ possessing a finite kernel.

In the case of the above definition where G and H are abelian varieties (such as elliptic curves,) the isogeny h is homomorphic between G and H. Because of this, isogenies over elliptic curves (and other abelian varieties) inherit certain characteristics.
For an isogeny $h : E_1 \to E_2$ defined over elliptic curves E_1 and E_2, the following holds:

- $h(\mathcal{O}) = \mathcal{O}$, and
- $h(u^{-1}) = h(u)^{-1}, \forall u \in G$

If there exists some isogeny ϕ between curves E_1 and E_2 then E_1 and E_2 are said to be *isogenous*. All supersingular curves are isogenous only to other supersingular curves. The equivalent statement holds for ordinary curves. With this in mind, we can concieve a sort of graph structure connecting all isogenous curves, these graphs pertaining to either the supersingular or ordinary variety of curves.

We write $\text{End}(E)$ to denote the ring formed by all the isogenies acting over E which are also endomorphisms. Note that m-repeated pointwise addition of a point with itself can equivalently be modelled by an endomorphism, we denote the application of such an endomorphism to a point P as $[m]P$, such that $[m] : E \to E$ and $[m]P = mP$.

An important facet of isogenies is that they can be uniquely identified by their kernel. If S is the group of points denoting the kernel of some isogeny ϕ with domain E, we write $\phi : E \to E/S$. Because the subgroup S sufficiently identifies ϕ, any given generator of S equivalently identifies ϕ. Therefore, if R generates the subgroup S we can write $\phi : E \to E/\langle R \rangle$. Moreover, we will have a specific interest in isogenies with kernels defined by some *torsion subgroup*.

Lemma 1 (Uniquelu identifying isogenies)**.** *Let E be an elliptic curve and let Φ be a finite subgroup of E. There are a unique elliptic curve E' and a seperable isogeny $\phi : E \to E'$ satisfying ker$(\phi) = \Phi$.*

2.3 Supersingular Isogeny Diffie-Hellman

This section will aim to accomplish two things. First, we will briefly explain the isogeny-level & key-exchange-level procedures of the SIDH protocol. Second, we will illuminate how these procedures map onto Microsoft Research's C implementation of SIDH. In this regard, this section can be considered an attempt to meld two domains of SIDH functions & procedures, in hopes of easing the navigation from the SIDH protocol to Microsoft's C implementation, and vice versa.

The original work of De Feo, Jao, and Plut [FJP12] outlines three different isogeny-based cryptographic primitives: Diffie-Hellman-esque key exchange, public key encryption, and the aforementioned zero-knowledge proof of identity (ZKPoI). Because all three of these protocols require the same initialization and public parameters, we will begin by covering these parameters in detail. Immediately after, we will analyze the key exchange at a relatively high level. Our goal of this section is to explain in detail the algorithmic and cryptographic aspects of the ZKPoI scheme, as this forms the conceptual basis for the signature scheme we will be investigating. We begin with the key exchange protocol because its sub-routines are integral to the Yoo et al. signature implementation.

For the discussion that follows, we will assume every instance of an SIDH protocol occurs between two parties, A and B (eg. Alice & Bob,) for which we will colorize information particular to A in red and B in blue. This will include private keys & public keys as well as the variables and constants used in their generation.

Public Parameters. As the name suggests, SIDH protocols work over supersingular curves (with no singular points). Let $\mathbb{F}_q = \mathbb{F}_{p^2}$ be the finite field over which our curves are defined, $\mathbb{F}_{p^2}$ denoting the quadratic extension field of $\mathbb{F}_p$. p is a prime defined as follows:

$$p = \ell_A^{e_A} \ell_B^{e_B} \cdot f \pm 1$$

Wherein ℓ_A and ℓ_B are small primes (typically 2 & 3, respectively) and f is a cofactor ensuring the primality of p. We then define globally a supersingular curve E_0 defined over $\mathbb{F}_q$ with cardinality $(\ell_A^{e_A} \ell_B^{e_B} f)^2$. Consequently, the torsion group $E_0[\ell_A^{e_A}]$ is $\mathbb{F}_q$-rational and has ${\ell_A}^{e_A - 1}(\ell_A + 1)$ cyclic subgroups of order $\ell_A^{e_A}$, with the analogous statement being true for $E_0[\ell_B^{e_B}]$. Additionally, we include in the public parameters the bases $\{P_A, Q_A\}$ and $\{P_B, Q_B\}$, generating $E[\ell_A^{e_A}]$ and $E[\ell_B^{e_B}]$ respectively.

This brings our set of global parameters, G, to the following:

$$G = \{p, E_0, \ell_A, \ell_B, e_A, e_B, \{P_A, Q_A\}, \{P_B, Q_B\}\}$$

2.3.1 SIDH Key Exchange

This subsection will illustrate an SIDH key exchange run between party members Alice and Bob. The general idea of the protocol can be surmised by the diagram below. In the scheme, **private keys** take the form of isogenies defined with domain E, and **public keys** are the associated co-domain curve of said isogenies.[FJP12] For the entirety of this section we will denote isogenies by their function symbols, but when we come section 2.6 to will show how we can efficiently represent isogenies in a computational environment.

$$\begin{tikzcd}
E_0 \arrow[r, "\phi_A"] \arrow[d, "\phi_B"'] & E_0/\langle A\rangle \arrow[d, "\phi'_B"] \\
E_0/\langle B\rangle \arrow[r, "\phi'_A"'] & E_0/\langle A, B\rangle
\end{tikzcd}$$

The premise of the protocol is that both parties each generate a random point (A or B in the diagram,) which, according to proposition 1, identifies some distinct isogeny $\phi_A : E_0 \to E/\langle A\rangle$ (or equivalent for B). Alice and Bob then exchange codomain curves and compute

$$\phi_A(E_0/\langle B\rangle)$$
or
$$\phi_B(E_0/\langle A\rangle).$$

From these isogenies, Alice and Bob arrive at their shared secret agreement: the mutual codomain curve of $\phi_A(E_0/\langle B\rangle)$ and $\phi_B(E_0/\langle A\rangle)$, denoted E_{AB}.

Below we've outlined the SIDH key exchange protocol $\Pi_{\text{SIDH}} = (\textbf{KeyGen}, \textbf{SecAgr})$ in a descriptive manner. We do not provide algorithmic definitions for these procedures, but C code corresponding to their implementations in $\texttt{SIDH}_\texttt{C}$ can be found in Appendix [include appendix].

KeyGen(λ): Alice chooses two random numbers $m_A, n_A \in \mathbb{Z}/\ell_A^{e_A}\mathbb{Z}$ such that $(\ell_A \nmid m_A) \vee (\ell_A \nmid n_A)$. Alice then computes the isogeny $\phi_A : E_0 \to E_A$ where $E_A = E_0/\langle [m_A]P_A, [n_A]Q_A\rangle$ (equivalently, $ker(\phi_A) = \langle [m_A]P_A, [n_A]Q_A\rangle$). Bob does the same for random elements $m_B, n_B \in \mathbb{Z}/\ell_B^{e_B}\mathbb{Z}$.

Alice then applies her isogeny to the points which Bob will use in the creation of of his isogeny: $(\phi_A(P_B), \phi_A(Q_B))$. Bob performs the analogous operation. This leaves us with the following private and public keys for Alice and Bob:

$$\begin{aligned}
sk_A &= (m_A, n_A) \\
pk_A &= (E_A, \phi_A(P_B), \phi_A(Q_B)) \\
sk_B &= (m_B, n_B) \\
pk_B &= (E_B, \phi_B(P_A), \phi_B(Q_A))
\end{aligned}$$

PK Exchange: After Alice and Bob successfully complete their key generation, they perform the following over an insecure channel:

- Alice sends Bob $(E_A, \{\phi_A(P_B), \phi_A(Q_B)\})$
- Bob sends Alice $(E_B, \{\phi_B(P_A), \phi_B(Q_A)\})$

Again, we remind the reader that we will show how curves such as E_A and E_B can be represented efficient and compactly in a computing environment when we come to our section on implementations of isogeny based systems (2.6).

SecAgr(sk_1,pk_2): After reception of Bob's tuple, Alice computes the isogeny $\phi'_A : E_B \to E_{AB}$ and Bob acts analogously. Alice and Bob then arrive at the equivalent image curve:

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

From this they can derive their shared secret k as the common j-invariant of E_{AB}.

We have included a graphical illustration of the entire SIDH key exchange process in Figure 2.4, wherein solid lines denote private computations, and dashed lines denote information sent over an insecure channel.

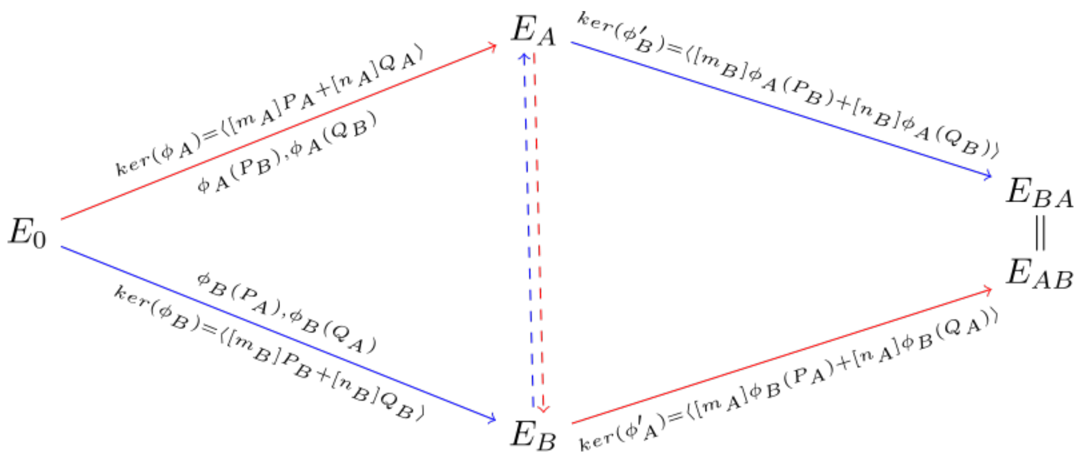

Figure 2.4: SIDH key exchange between Alice & Bob [FJP12]

2.3.2 Zero-Knowledge Proof of Identity

Recall the earlier discussed notion of an identification scheme. A canonical identification scheme $\Pi_{\text{SID}} = (\textbf{KeyGen}, \textbf{Prove}, \textbf{Verify})$ can be derived somewhat analogously to the SIDH protocol, and is outlined in the original work of De Feo et al.

Say Bob has derived for himself the key pair (sk_B, pk_B) with $sk_B = \{m_B, n_B\}$ and $pk_B = E_B = E_0/\langle [m_B]P_B + [n_B]Q_B \rangle$ in relation to the public parameters E_0 and $\ell_B^{e_B}$. With E_0 and E_B publicly known, Π_{ZKPoI} revolves around Bob trying to prove to Alice that he knows the generator for E_B without revealing it.

To achieve this, Bob internally mimicks an execution of the key exchange protocol Π_{SIDH} with an arbitrary "random" entity Randall.

KeyGen: Key generation is performed exactly as in Π_{SIDH}, the only difference being that in Π_{ZKPoI} only the prover (Bob, in our example,) needs to generate a keypair.

Commitment: Bob generates a random point $R \in E_0[\ell_A^{e_A}]$ ($R = [m_R]P_A + [n_R]Q_A$) along with the corresponding isogenies necessary to compute the diagram below in full (if Alice were acting as the prover in Π_{ZKPoI}, then she would choose $R \in E_0[\ell_B^{e_B}]$). Bob sends his commitment *com* as $(com_1, com_2) = (E/\langle R\rangle, E/\langle B, R\rangle)$ to Alice.

$$\begin{tikzcd}
E_0 \arrow[r, "\phi_B"] \arrow[d, "\psi_R"'] & E_0/\langle B\rangle \arrow[d, "\psi'_R"] \\
E_0/\langle R\rangle \arrow[r, "\phi'_B"'] & E_0/\langle B, R\rangle
\end{tikzcd}$$

Response: Alice chooses a bit b at random and sends her challenge $ch = b$ to Bob.

Prove(sk, ch): If Alice's challenge bit $ch = 0$ then Bob reveals the isogenies ψ_R and ψ'_R (to do this, he can simply reveal the generators of the kernels of ψ_R and ψ'_R; R and $\phi_B(R)$ respectively). This proves he knows the information necessary to form a shared secret with Randall *if and only if* he happens to know the private key $B = \{[m_B]P_B + [n_B]Q_B\}$. If $ch = 1$, Bob reveals the isogeny ϕ'_B. This proves that Bob knows the information necessary to form a shared secret with Randall *if and only if* he knows Randall's secret key R.

$$\begin{tikzcd}
E_0 \arrow[r, dashed, no head, "\phi_B"] \arrow[d, Rightarrow, "\psi_R"'] & E_0/\langle B\rangle \arrow[d, Rightarrow, "\psi'_R"] \\
E_0/\langle R\rangle \arrow[r, dashed, no head, "\phi'_B"'] & E_0/\langle B, R\rangle
\end{tikzcd}
\qquad
\begin{tikzcd}
E_0 \arrow[r, dashed, no head, "\phi_B"] \arrow[d, dashed, no head, "\psi_R"'] & E_0/\langle B\rangle \arrow[d, dashed, no head, "\psi'_R"] \\
E_0/\langle R\rangle \arrow[r, Rightarrow, "\phi'_B"'] & E_0/\langle B, R\rangle
\end{tikzcd}$$

Note that Bob cannot at once reveal all of the information necessary to convince Alice that he knows B. If he reveals R, $\phi_B(R)$, and ϕ'_B all in one go, he incidentally reveals his secret key $B = [m_B]P_B + [n_B]Q_B$. This is because Bob reveals ϕ'_B by revealing the generator of $ker(\phi'_B)$, namely:

$$\langle B, R\rangle = [m_B]P_B + [n_B]Q_B, [m_R]P_A + [n_R]Q_A\rangle$$

How Π_{ZKPoK} handles this is by having Bob and Alice run **Prove()** and **Verify()** for λ iterations, with a different $(com, ch, resp)$ transcript generated for every instance. This way, if Bob is able to provide a *resp* that satisfies Alice's *ch* for every iteration, she can be sufficiently confident that Bob has knowledge of B.

Verify(pk, com, ch): Like the proving procedure, verification is a conditional function depending on the value of b:

- if $ch = 0$: return 1 *if and only if* R and $\phi_B(R)$ have order $\ell_A^{e_A}$ and generate the kernels of isogenies from $E_0 \to E_0/\langle R\rangle$ and $E_0/\langle B\rangle \to E_0/\langle B, R\rangle$ respectively.

- if $ch = 1$: return 1 *if and only if* $\psi_R(B)$ has order $\ell_B^{e_B}$ and generates the kernel of an isogeny over $E_0/\langle R\rangle \to E_0/\langle B, R\rangle$.

This scheme constitutes what is known in the literature as a *zero knowledge* proof of identity. It is referred to as such because Alice, acting as the verifier, does not gain any information about Bob's secret key sk.

2.4 Fiat-Shamir Construction

The Fiat-Shamir construction (also frequently referred to as the Fiat-Shamir transform, or Fiat-Shamir hueristic,) is a high-level technique for transforming a canonical identification scheme into a secure signature scheme.

The construction is rather simple. The idea is to first transform a given interactive identification protocol Π_{ID} into a *non-interactive* identification protocol. To achieve this, instead of allowing input from the verifier $\mathcal{V}$, we have our prover $\mathcal{P}$ generate the challenge ch by itself. In order for the verifier to be able to check that ch was generated honestly, we define $ch = H(com)$, where H is some secure hash function. If we model H as a *random oracle*[ref], $H(com)$ is truly random; from this it can be shown that it is just as difficult for an impersonator of $\mathcal{P}$ to find an accepting transcript $(com, H(com), resp)$ as it would be for them to successfully impersonate $\mathcal{P}$ in Π_{ID}.

Now that we've paired Π_{ID} with H to achieve a non-interactive identification scheme Π_{NID}, we need only to factor in some message m from $\mathcal{P}$ to have constructed a signature scheme Π'_{ID}. This can be achieved by including m in our calculation of the challenge: $ch = H(com, m)$. Therefore, given theorem 1, if $(com, H(com), resp)$ is an accepting transcript of Π_{NID}, then $(com, H(com, m), resp)$ is a secure signature for the message m. Of course, because $H(com, m)$ can be constructed by any passively observing party, it is redundant to include; and so $(com, resp)$ constitutes a valid signature for m. A proof of theorem 1 can be found in [Kat10]. The security of the Fiat-Shamir construct was first proven by Pointcheval & Stern in [PS96].

Theorem 1 (Fiat-Shamir Security). *Let $\Pi_{ID} = (\textbf{\textit{KeyGen}}, \textbf{\textit{Prove}}, \textbf{\textit{Verify}})$ be a canonical identification scheme that is secure against a passive attack. Then, if H is modeled as a random oracle, the signature scheme Π'_{ID} that results from applying the Fiat-Shamir transform to Π_{ID} is* classically *existentially unforgeable under an adaptive chosen-message attack.*

We will write $\mathbf{FS}(\Pi)$ to denote the result of applying the Fiat-Shamir transformation to some identification protocol Π.

2.4.1 Unruh's Post-Quantum Adaptation

In 2014, Ambainis et al. showed that classical security proofs for "proof of knowledge" protocols are insecure in the quantum setting. This is due to a technique used in the proof of FST's security whereby the random oracle is subject to "rewinding": the proof simulates multiple runs of FST with different responses from the random oracle [ARU14].

Following this insight, Unruh proposed in 2015 a construction based off that of Fiat & Shamir which he proved to be secure in both the classical and quantum random oracle models.

Because the focus of this dissertation is not the security (post-quantum or otherwise) of any particular protocol, our coverage in this section is left brief.

2.5 Isogeny Based Signatures

Since publication of the SIDH suite, there have been several attempts at providing authentication schemes within the same framework. The post-quantum community had demonstrated undeniable signatures[JS14], designated verifier signatures[STW12], and undeniable blind signatures[SC16] all within the framework of isogeny based systems. It was not until the work of Yoo et al., however, that an isogeny based protocol for general authentication was shown as demonstrably secure. This protocol, particularly its C implementation, is where we have decided to focus our efforts.

Now that we've seen the zero-knowledge proof of identity (ZKPoI) from [FJP12] as well as Unruh's quantum-safe Fiat-Shamir adaption, we have presented all of the material necessary for an indepth analysis of the isogeny based signature scheme presented by Yoo et al. The signature protocol, which we'll denote as Σ', is a near-trivial application of Unruh's construction to the SIDH ZKPoI. In this section we will refer to the SIDH ZKPoI as Σ.

Σ' is defined in the traditional manner, by a tuple of algorithms for key generation, signing, and verifying: $\Sigma' = (\textbf{KeyGen}(), \textbf{Sign}(), \textbf{Verify}())$. As could be predicted, **KeyGen()** in Σ' is defined identically to the key generation found in SIDH key exchange. **Sign()** and **Verify()** are defined as equivalent to **Prove()** and **Verify()** $\in$ **FS**(Σ), respectively.

For our discussion of the signature scheme, we will make use of the naming conventions used in section 2.3. That is, we will discuss Σ' as occuring between entities Bob and Alice, with Bob imitating the role of an arbitrary third party Randall during **Sign**.

The public parameters used in Σ' are the same as outlined above for all of the protocols found in [FJP12]. Namely, we have $p = \ell_A^{e_A}\ell_B^{e_B} \cdot f \pm 1$ where $\ell_A^{e_A} = 2$, $\ell_B^{e_B} = 3$, and f is a cofacter such that p is prime. We also set as parameter the curve E such that $\#(E(F_{p^2})) = (\ell_A^{e_A}\ell_B^{e_B})^2$. And again, we include the sets of points (P_A, Q_A) and (P_B, Q_B) generating $E[\ell_A^{e_A}]$ and $E[\ell_B^{e_B}]$ respectively. We have chosen E over the previously used E_0 simply for ease of notation.

2.5.1 Algorithmic Definitions

It will be useful for us to outline in more detail the procedures of Σ', at the very least to ease the transition into our discussion of the C implementation. In this subsection we will look at isogeny-level algorithmic definitions for **KeyGen()**, **Prove()**, and **Verify()**, and then look at how these procedures can be expressed in terms of the procedures of Π_{SIDH}.

KeyGen(λ, $User$): As previously mentioned, key generation in Σ' is identical to Σ:**KeyGen(λ)**, which in turn is identical to Π_{SIDH}:**KeyGen(λ)**. We've included a parameter $User$ equaling either *Alice* or *Bob* – this denotes whether the user running the procedure uses blue or red constants. We've also obfuscated the lower level details in regards to how points are generated and how isogenies can be constructed. We write P_{User} and Q_{User} for P_A & Q_A or P_B & Q_B, depending on $User$. The result is the following:

Algorithm 1 – **KeyGen**(λ, $User$)

if $User$ = $Alice$ **then**
  Pick a random point S of order $\ell_A^{e_A}$
if $User$ = Bob **then**
  Pick a random point S of order $\ell_B^{e_B}$
Compute the isogeny $\phi : E \to E/\langle S \rangle$
$pk \leftarrow (E/\langle S \rangle, \phi(P_{User}), \phi(Q_{User}))$
$sk \leftarrow S$
return (sk, pk)

Transcribing this to the procedures of Π_{SIDH} we arrive (quite trivially) at:

Algorithm 2 – **KeyGen**(λ)

$(sk, pk) \leftarrow \Pi_{\text{SIDH}}$:**KeyGen**($\lambda$)
return (sk, pk)

For **Sign**(sk, m) and **Verify**(pk, m, σ) we assume Bob to be the signer and Alice to be the verifier. Consequently, we will write the signers key pair (sk, pk) as (B, ϕ_B). Algorithms for which the roles are reversed can be constructed simply by replacing red constants with their blue correspondants, and vice-versa.

Sign(sk, m): The sign procedure, as a consequence of the Unruh construction, makes use of two random oracle functions **H** amd **G**. In the sign algorithm below, make note of how Bob computes both commitments and their corresponding responses for every iteration i before he computes the challenge values (the bits of J). He then uses the 2λ bits of J to decide which responses to include in σ.

Algorithm 3 – **Sign**($sk = B$, m)

for `i = 1..2`λ **do**
  Pick a random point R of order $\ell_A^{e_A}$
  Compute the isogeny $\psi_R : E \to E/\langle R \rangle$
  Compute either $\phi'_B : E/\langle B \rangle \to E/\langle B, R \rangle$ or $\psi'_R : E/\langle R \rangle \to E/\langle R, B \rangle$
  $(E_1, E_2) \leftarrow (E/\langle R \rangle, E/\langle R, B \rangle)$
  $com_i \leftarrow (E_1, E_2)$
  $ch_{i,0} \leftarrow_R \{0, 1\}$
  $(resp_{i,0}, resp_{i,1}) \leftarrow ((R, \phi_B(R)), \psi_R(B))$
  if $\mathtt{ch}_{i,0} = 1$ **then**
    Swap($resp_{i,0}$,$resp_{i,1}$)
  $h_{i,j} \leftarrow$ **G**($resp_{i,j}$)
$J_1 \parallel \ldots \parallel J_{2\lambda} \leftarrow$ **H**(ϕ_B, m, $(com_i)_i$,$(ch_{i,j})_{i,j}$,$(h_{i,j})_{i,j}$)
return $\sigma \leftarrow ((com_i)_i, (ch_{i,j})_{i,j}, (h_{i,j})_{i,j}, (resp_{i,J_i})_i)$

If we write out **Sign()** using the Π_{SIDH} API, we see that the only real computation is being performed by **KeyGen()** and **SecAgr()**, and our two random oracles **H()** and

$\mathbf{G}()$. The rest of the algorithm is merely organizing the information we've generated into the transcript $(com, ch, resp)$ and then finally into σ.

Algorithm 4 – $\mathbf{Sign}(sk = B,\ m)$ via Π_{SIDH}

for `i = 1..2`λ **do**
 $(R, \psi_R) \leftarrow \Pi_{\text{SIDH}}\text{:}\mathbf{KeyGen}(\lambda,\ Alice)$
 $\phi'_B : E/\langle B \rangle \to E/\langle B, R \rangle \leftarrow \Pi_{\text{SIDH}}\text{:}\mathbf{SecAgr}(B,\ \psi_R)$
 $(E_1, E_2) \leftarrow (E/\langle R \rangle, E/\langle B, R \rangle)$
 $com_i \leftarrow (E_1, E_2)$
 $ch_{i,0} \leftarrow_R \{0, 1\}$
 $(resp_{i,0}, resp_{i,1}) \leftarrow ((R, \phi_B(R)), \psi_R(B))$
 if $\mathtt{ch}_{i,0} = 1$ **then**
  $\mathbf{Swap}(resp_{i,0}, resp_{i,1})$
 $h_{i,j} \leftarrow \mathbf{G}(resp_{i,j})$
$J_1 \parallel \ldots \parallel J_{2\lambda} \leftarrow \mathbf{H}(\phi_B,\ m,\ (com_i)_i, (ch_{i,j})_{i,j}, (h_{i,j})_{i,j})$
return $\sigma \leftarrow ((com_i)_i, (ch_{i,j})_{i,j}, (h_{i,j})_{i,j}, (resp_{i,J_i})_i)$

$\mathbf{Verify}(pk,\ m,\ \sigma)$: Alice begins her execution of $\mathbf{Verify}()$ where Bob ended his execution of $\mathbf{Sign}()$, with the computation of J. Alice then knows at each iteration what check to perform on Bob's response, based on a conditional branch. You will notice that Bob's secret key B occurs in the negative path of this branch; this is not a security concern because it is actually the point $\psi_R(B)$ that is communicated in σ, from which B cannot be recovered.

Algorithm 5 – $\mathbf{Verify}(pk = \phi_B,\ m,\ \sigma)$

$J_1 \parallel \ldots \parallel J_{2\lambda} \leftarrow \mathbf{H}(\phi_B,\ m,\ (com_i)_i, (ch_{i,j})_{i,j}, (h_{i,j})_{i,j})$
for `i = 0..2`λ **do**
 check $h_{i,J_i} = G(resp_{i,J_i})$
 if $ch_{i,J_i} = 0$ **then**
  Parse $(R, \phi_B(R)) \leftarrow resp_{i,J_i}$
  check $(R, \phi_B(R))$ have order $\ell_A^{e_A}$
  check R generates the kernel of the isogeny $E \to E_1$
  check $\phi_B(R)$ generates the kernel of the isogeny $E/\langle B \rangle \to E_2$
 else
  Parse $\psi_R(B) \leftarrow resp_{i,J_i}$
  check $\psi_R(B)$ has order $\ell_B^{e_B}$
  check $\psi_R(B)$ generates the kernel of the isogeny $E_1 \to E_2$
if all checks succeed **then**
 return 1
else
 return 0

What we are checking for in the verification process is whether or not Bob and Randall performed an honest and valid key exchange. And so, if the challenge bit is 0, we can use SIDH key generation to determine that R and ψ_R are a valid key pair and then run SIDH secret agreement with R and Bob's public key ϕ_B to confirm that it properly executes outputting an isogeny with kernel generated by $\phi_B(R)$. If the challenge bit is 1, we can run an instance of SIDH secret agreement to verify that $\psi_R(B)$ generates the kernel of an isogeny with domain E_1 and co-domain E_2 (refer again to the diagrams outlining **Prove** of section 2.3.2).

These observations are formalized in Algorithm 6, where we rewrite Σ':**Verify()** in terms of Π_{SIDH} procedure calls. Note, in line 10 of Algorithm 6, the call Π_{SIDH}:**SecAgr(**$\psi_R(B)$**,** ψ_R**)**. It should be noted that $\psi_R(B)$ is not the proper secret key input used by Bob in **Sign()**, but we will see in the section to follow how we can use $\psi_R(B)$ in the C implementation of **SecAgr** to perform our verification (without compromising Bob's secret key B).

Algorithm 6 – **Verify(**$pk = \phi_B$**,** m**,** σ**)** via Π_{SIDH}

1: $J_1 \parallel ... \parallel J_{2\lambda} \leftarrow$ **H(**ϕ_B**,** m**,** $(com_i)_i$**,**$(ch_{i,j})_{i,j}$**,**$(h_{i,j})_{i,j}$**)**
2: **for** `i = 0..2`λ **do**
3:     **check** $h_{i,J_i} = G(resp_{i,J_i})$
4:     **if** $ch_{i,J_i} = 0$ **then**
5:         Parse $(R, \phi_B(R)) \leftarrow resp_{i,J_i}$
6:         **check** (R, ψ_R) is a valid output of Π_{SIDH}:**KeyGen(**λ**,** *Alice***)**
7:         **check** that Π_{SIDH}:**SecAgr(**R**,** ϕ_B**)** successfully outputs an isogeny with co-domain E_2
8:     **else**
9:         Parse $\psi_R(B) \leftarrow resp_{i,J_i}$
10:         **check** that Π_{SIDH}:**SecAgr(**$\psi_R(B)$**,** ψ_R**)** successfully outputs an isogeny with co-domain E_2
11: **if** all checks succeed **then**
12:     **return** 1
13: **else**
14:     **return** 0

2.6 Implementations of Isogeny Based Cryptographic Protocols

Having now introduced all of the background material necessary for understanding SIDH and the isogeny based signature scheme in detail, we will investigate the portions of the SIDH C library which are relevent to our contributions.

The SIDH C library, written by a research team at Microsoft Research, was released in 2016 alongside an article titled *Efficient Algorithms for Supersingular Isogeny Diffie-Hellman* (see [CLN16]). The article in question details several adjustments to the algorithms and data-representations outlined in [FJP12], leading to improved performance and key-sizes. Their C implementation (which we will henceforth refer to as $\texttt{SIDH}_\texttt{C}$) is built on top of these improved algorithms and tailored to a specific set of parameters, all-in-all offering 128-bit quantum security and 192-bit classical security key exchange up to 2.9 times faster than any previous implementation. We will look at some of the details of $\texttt{SIDH}_\texttt{C}$ below.

Before proceeding, it may be advisable to briefly overview the section on notation (**??**) if one has not already.

2.6.1 Parameters & Data Representation

Parameters. $\texttt{SIDH}_\texttt{C}$ operates over the underlying basefield $\mathbb{F}_p$ where $p = \ell_A^{e_A} \cdot \ell_B^{e_B} - 1$, with $\ell_A = 2$, $\ell_B = 3$, $e_A = 372$, and $e_B = 239$, giving p a bitlength of 751. Now, recall the Montgomery representation of a curve:

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

$\texttt{SIDH}_\texttt{C}$ uses the public parameter curve E defined in Montgomery form with $A = 0$, $B = 1$, and $C = 1$. The point pairs (P_A, Q_A) and (P_B, Q_B), generating $E[\ell_A^{e_A}]$ and $E[\ell_B^{e_B}]$ respectively, are hard-coded as an array of bytes. This set of parameters (including related data such as the bitlength of certain constants) is referred to in the system as `SIDHp751`, and is stored in the struct `CurveIsogenyStaticData` which will be outlined below.

Data Structures. There are several custom-defined data structures that are integral to $\texttt{SIDH}_\texttt{C}$. Below, we will briefly cover the ones which are likely to arise in our discussion:

Field elements

- `felm_t` – buffer of bytes representing elements of $\mathbb{F}_p$.
- `f2elm_t` – pair of `felm_t` representing elements of $\mathbb{F}_{p^2}$.

Elliptic curve points

- `point_affine` – an `f2elm_t` `x` and an `f2elm_t` `y` representing a point in affine space.
- `point_proj` – an `f2elm_t` `X` and an `f2elm_t` `Z` representing a point as projective XZ Montgomery coordinates.
- `point_full_proj` – `f2elm_t` elements `X`, `Y`, and `Z` representing a point in projective space.
- `point_basefield_affine` – an `felm_t` `x` and an `felm_t` `y` representing a point in affine space over the base field.
- `point_basefield_proj` – an `felm_t` `X` and an `felm_t` `Z` representing a point as projective XZ Montgomery coordinates over the base field.

Crypto structures

- `publickey_t` – three `f2elm_t`s representing a public key.
  `publickey_t[0]` = users private isogeny applied to the other parties generator P_x
  `publickey_t[1]` = users private isogeny applied to the other parties generator Q_x
  `publickey_t[2]` = users private isogeny applied $P_x - Q_x$

Curve structures

- `CurveIsogenyStruct` – Structure containing all necessary public parameter data.
- `CurveIsogenyStaticData` – The same as `CurveIsogenyStruct`, but with buffer sizes fixed for `SIDHp751`.

The reader may note that `publickey_t` does not contain any information defining the users co-domain curve $E/\langle S\rangle$ (with S as the users secret key). It just so happens that in Π_{SIDH} key exchange, the curves $E/\langle A\rangle$ and $E/\langle B\rangle$ are simply intermediary steps (useful for conceptualizing the protocol) and not necessary for computing the shared secret $j(E_{AB})$.

2.6.2 Design Decisions

Projective Space. As is common in ECC, a vast majority of the procedures of $\texttt{SIDH}_\texttt{C}$ operate over the *projective space*(2.2.2) representation of elliptic curve points. This widely-deployed technique is used to avoid the substantial cost of field element inversions (computing x^{-1} for some element $x \in \mathbb{F}_{p^2}$). This means the majority of our calculations are performed over `point_proj` structures using *montgomery arithmetic*(2.2.1) and converted to `point_affine` when necessary. This general design strategy is highly related to our first contribution, which will be elaborated upon in the sections to come.

Secret Keys. Recall the origin of an Π_{SIDH} private key (m, n): the goal is to randomly select a generator of the torsion group $E[\ell_A^{e_A}]$ (or $E[\ell_B^{e_B}]$ for Bob). It is noted in [FJP12] that any generator of the required torsion group is sufficient. It is also noted that m, unless equal to the order of the torsion group, is invertible. Because of this, Alice, for example, can simple compute $R = P_A + [m^{-1}n]Q_A$, thus enabling secret keys to be stored as a single $\mathbb{F}_{p^2}$ element (which is referred to as m). This technicality has been implemented in $\texttt{SIDH}_\texttt{C}$, which both saves on storage as well as offers a means for generating secret keys that is more efficient than the trivial scalar multiplication and point-wise addition approach to computing $[m]P + [n]Q$.

Tailor-made Montgomery Multiplication.

2.6.3 Key Exchange & Critical Functions

There are 3 central modules (C file) in $\texttt{SIDH}_\texttt{C}$, all dealing with different levels of abstraction in the Π_{SIDH} protocol. Operating at the lowest abstraction level is the module `fpx.c`, wherein functions for manipulating $\mathbb{F}_p$ and $\mathbb{F}_{p^2}$ elements are defined. One level up from `fpx.c` we have `ec_isogeny.c`, containing functions pertaining to elliptic curves and point arithmetic (such as `j_inv(...)` for computing the j-invariant of a curve and `secret_pt(...)` for computing a users secret point S given their private key m). The final, highest abstraction-level module we will discuss is `kex.c`. `kex.c` contains the protocol-level functions for performing Π_{SIDH}, namely `KeyGeneration_A(...)` and `KeyGeneration_B(...)` for generating Alice and Bob's private and public keys, as well as `SecretAgreement_A(...)` and `SecretAgreement_B(...)` for completing the secret agreement from both sides of the key exchange. Figure 2.5 illustrates the relationship between these modules and the abstraction levels of isogeny based key exchange.

The estbalished notation for functions in `fpx.c` is to prepend function names with either `fp` or `fp2` to signify if it is defined for elements of $\mathbb{F}_p$ or $\mathbb{F}_{p^2}$. Additionally, it is common to append `_mont` to the name of functions which are defined using Montgomery arithmetic. Functions in `fpx.c` are largely defined by byte and memory arithmetic, with the exception of slightly higher-level functions such as field element inversion (`fpinv751_mont(...)`) which are defined in terms of other `fpx.c` functions. Furthermore, for efficiency, functions

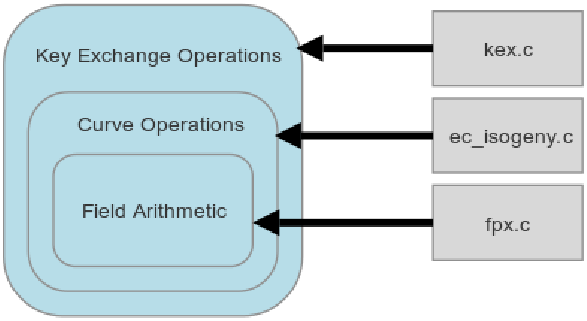

Figure 2.5: Relationship between Π_{SIDH} & $\mathtt{SIDH}_{\mathtt{C}}$ modules

of `fpx.c` are defined as `__inline` when applicable.

`ec_isogeny.c` functions are defined almost exclusively in terms of `fpx.c` functions, with a few occurances of internal function calling. Functions in this module that are signifant to our our work are briefly surmised in figure ??. The implementation specifics of most other `ec_isogeny.c` functions are not critical to our work, and so have been excluded. The design and efficiency of these algorithms do, however, have a rich background and can be further read about in [FJP12] and [CLN16].

The key exchange procedures found in `kex.c` are composed entirely of calls to `fpx.c` and `ec_isogeny.c` functions, modulo some basic branching logic.

`fpx.c` functions

Function	Input	Output
`to_fp2mont` Converts an $\mathbb{F}_{p^2}$ element to Montgomery representation	`f2elm_t a`	`f2elm_t ma`
`from_fp2mont` Converts an $\mathbb{F}_{p^2}$ element from Montgomery representation to regular form	`f2elm_t ma`	`f2elm_t a`
`fp2inv751_mont_bingcd` performs *non-constant* time inversion of a $\mathbb{F}_{p^2}$ element	`f2elm_t a`	`f2elm_t` $\mathtt{a}^{-1}$
`fp2inv751_mont` performs *constant* time inversion of a $\mathbb{F}_{p^2}$ element	`f2elm_t a`	`f2elm_t` $\mathtt{a}^{-1}$

`ec_isogeny.c` functions

Function	Input	Output
`j_inv` computes the j-invariant of a curve with represented in Montgomery form with `A` and `C`	`f2elm_t A` `f2elm_t C`	`f2elm_t jinv`

`secret_pt`
generates the secret
point `R` from
secret key `m`
	`point_basefield P`
	`digit_t m`
	`SIDHp751`
	`int AliceOrBob`
		`point_proj R`

`inv_3_way`
performs simultaneous inversion
of three elements
	`f2elm_t z1`
	`f2elm_t z2`
	`f2elm_t z3`
		`f2elm_t` $z1^{-1}$
		`f2elm_t` $z2^{-1}$
		`f2elm_t` $z3^{-1}$

`inv_4_way`
performs simultaneous inversion
of 4 elements
	`f2elm_t z1`
	`f2elm_t z2`
	`f2elm_t z3`
	`f2elm_t z4`
		`f2elm_t` $z1^{-1}$
		`f2elm_t` $z2^{-1}$
		`f2elm_t` $z3^{-1}$
		`f2elm_t` $z4^{-1}$

`generate_2_torsion_basis` constructs a basis $(\{R1, R2\})$ generating $E[\ell_A^{e_A}]$	`f2elm_t A` `SIDHp751`	`point_full_proj R1` `point_full_proj R2`
`generate_3_torsion_basis` constructs a basis $(\{R1, R2\})$ generating $E[\ell_B^{e_B}]$	`f2elm_t A` `SIDHp751`	`point_full_proj R1` `point_full_proj R2`

`kex.c` functions

Function	Input	Output
`KeyGeneration_A` performs key generation for Alice	`unsigned char* privateKeyA` `bool generateRandom`	`unsigned char* privateKeyA` `unsigned char* publicKeyA`
`KeyGeneration_B` performs key generation for Bob		`unsigned char* privateKeyB` `unsigned char* publicKeyB`

`SecretAgreement_A`
computes the shared secret
from Alice's perspective
	`unsigned char* privateKeyA`
	`unsigned char* publicKeyB`
	`point_proj kerngen`
		`unsigned char* sharedSecretA`
		`point_proj kerngen`

`SecretAgreement_B`
computes the shared secret
from Bob's perspective
	`unsigned char* privateKeyB`
	`unsigned char* publicKeyA`
	`point_proj kerngen`
		`unsigned char* sharedSecretB`
		`point_proj kerngen`

The reader may note that `privateKeyA` (in `KeyGeneration_A`) and `kerngen` (in both secret agreements) appear as both inputs and outputs. This is no mistake. In `KeyGeneration_A`, if `generateRandom = false` is passed as an input, then `privateKeyA` is expected to be set, and the corresponding public key is computed. In secret agreement, if `kerngen` is set to null then the algorithm proceeds normally. If it is set to a valid point, however, it can be used in place of a secret key input (which in such a case is expected to be null). Both of these details are critical to the design of signature functions as they are described below.

2.6.4 Signature Layer

Yoo et al. provided, along with their publication of [YAJ+12], an implementation of their signature scheme as a fork to $\texttt{SIDH}_\texttt{C}$. All of their functions are written specifically for an instance of Σ' where the signer is assuming the B role (meaning that Randall assumes the A role), but their algorithms could be trivially modified to provide versions supporting a signer in the A role. Their contributions to the $\texttt{SIDH}_\texttt{C}$ codebase come in the form of the functions listed below – their relation to the rest of $\texttt{SIDH}_\texttt{C}$ is illustrated in figure 2.6.

Function	Input	Output
`isogeny_keygen` generates the signers key pair		`unsigned char* privateKeyB` `unsigned char* publicKeyB`

`isogeny_sign`
produces a signature for a message
	`privateKey`
	`publicKey`
	message m
		`Signature sig`

`sign_thread` performs a single iteration of the for-loop in **Sign**		
`isogeny_verify` checks the validity of a signature	`Signature sig`	`true` or `false`
`verify_thread` performs a single iteration of the for-loop in **Verify**		

To begin, `isogeny_keygen` has a trivial definition; `KeyGeneration_B` is called and populates the signer's public and private keys. `isogeny_keygen` returns the success status of the call to `KeyGeneration_B`.

In their original fork of $\texttt{SIDH}_\texttt{C}$, Yoo et al. included these functions in the file `kex_tests.c`. This file was originally intended for testing the functions of `kex.c`, and so our fork of the library has placed the signature functions in a new file `SIDH_signature.c`. We have also included a file `sig_tests.c` for testing the contents and performance of `SIDH_signature.c` functions.

If we transcribe the procedures Σ':**Sign** and Σ':**Verify** as described in section 2.5 to the language of the $\texttt{SIDH}_\texttt{C}$ API, we have in essense the following:

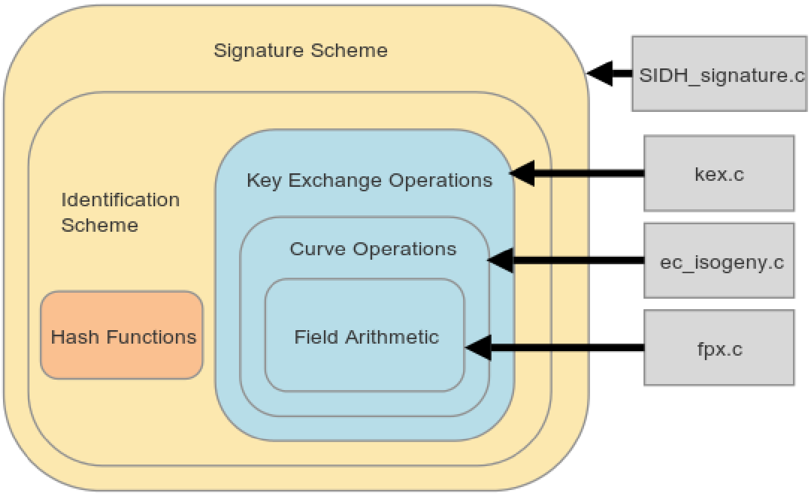

Figure 2.6: Relationship between SIDH based signatures & Our fork of the SIDH C library

Algorithm 7 – $\texttt{Sign}(sk_B,\ m)$

1: **for** `i = 1..`2λ **do**
2: $\quad (sk_R = R, pk_R) \leftarrow \texttt{KeyGeneration\_A(NULL, true)}$
3: $\quad (E/\langle B, R\rangle, \psi_R(B)) \leftarrow \texttt{SecretAgreement\_B}(sk_B, pk_R, \texttt{NULL})$
4: $\quad (E_1, E_2) \leftarrow (E/\langle R\rangle, E/\langle B, R\rangle)$
5: $\quad (\texttt{com}[i]_0, \texttt{com}[i]_1) \leftarrow (E_1, E_2)$
6: $\quad (\texttt{resp}[i]_0, \texttt{resp}[i]_1) \leftarrow (R, \psi_R(B))$
7: $\quad h[i] \leftarrow \texttt{keccak}(\texttt{resp}[i]_0)\,|\,\texttt{keccak}(\texttt{resp}[i]_1)$
8: $J_1 \parallel \ldots \parallel J_{2\lambda} \leftarrow \texttt{keccak}(\texttt{com},\ m,\ h)$
9: **return** $\sigma \leftarrow ((\texttt{com}_i)_i, (\texttt{ch}_{i,j})_{i,j}, (h_i)_i, ((\texttt{resp})[J_i])$

Algorithm 8 – Verify($pk = \phi_B$, m, σ)

```
J_1 || ... || J_2λ ← keccak(com, m, h)
for i = 0..2λ do
    check h[i] = keccak(resp[i]_0)|keccak(resp[i]_1)
    if J_i = 0 then
        R ← resp[i]_0
        pk_R ← KeyGeneration_A(R, false)
        check pk_R = com[i]_0
        E_RB ← SecretAgreement_A(R, φ_B, NULL)
        check E_RB = com[i]_1
    else
        ψ_R(B) ← resp[i]_1
        pk_R ← com[i]_0
        E_BR ← SecretAgreement_B(NULL, pk_R, ψ_R(B))
        check E_BR = com[i]_1
if all checks succeed then
    return 1
else
    return 0
```

Outside of simply replacing Π'_{SIDH} procedure calls with $\texttt{SIDH}_\texttt{C}$ functions, the reader may notice additional differences between `Sign` and `Verify` and their Σ' counterparts. Namely, Yoo et al. have chosen to exclude the challenge bit *ch* in the $\texttt{SIDH}_\texttt{C}$ implementations of these functions, consequently excluding the conditional and **Swap** statement of lines 8 and 9 in algorithm 4.

Chapter 3

Batching Operations for Isogenies

Our first contribution to the $\texttt{SIDH}_\texttt{C}$ codebase is the implementation and integration of a procedure for batching together many $\mathbb{F}_{p^2}$ element inversions. This contribution is discussed in detail in the following chapter. The chapter is split into three sections: a high-level discussion of the procedure itself, the low-level details of its integration into $\texttt{SIDH}_\texttt{C}$, and finally, the resulting affects of this procedure on the performance of $\texttt{SIDH}_\texttt{C}$.

In the first section of this chapter we will detail the specifics of the partial batched inversion procedure. We will show how the procedure can be constructed by combining two techniques: a well known method for reducing a $\mathbb{F}_{p^2}$ inversion to several $\mathbb{F}_p$ operations, and an inversion batching technique outlined in [SB01].

As we then venture into the lower-level implementation details, we will explore how the procedure can be leveraged optimally in the codebase. We will take a closer look at several of the aforementioned $\texttt{SIDH}_\texttt{C}$ functions as we illustrate some of the performance bottlenecks in the system. At this time, we will also discuss the design decisions made while implementing the partial batched inversion procedure as well as some of the functions lower-level minutiae.

We will end this chapter by taking a detailed look at the performance gains offered by the inclusion of partial batched inversions in $\texttt{SIDH}_\texttt{C}$. More precisely, we will be examining the effects of the procedure on the Yoo et al. signature layer. We will contrast the measured performance of our implementation with an analytical calculation of the expected improvement, and discuss the possible origins of divergent behaviour.

3.1 Partial Batched Inversions

We will now outline our first contribution to $\texttt{SIDH}_\texttt{C}$. The "partial batched inversion" procedure in question reduces arbitrarily many *unrelated* $\mathbb{F}_{p^2}$ inversions to a sequence of $\mathbb{F}_p$ operations. The fact that the elements being inverted need not hold any relation will be significant to the applicability of this procedure. For brevities sake, we will henceforth refer to this procedure as `pb_inv` in the $\texttt{SIDH}_\texttt{C}$ context, and **PartialBatchedInversion** in the more general mathematical context.

As mentioned above, `pb_inv` is constructed by combining two distinct techniques. Both of these techniques improve the efficiency of computing field element inversions: the first is specific to extension fields (in our case, $\mathbb{F}_{p^2}$ elements,) but the second is a technique applicable to field element inversions in a more general setting.

We will begin with a dissection of these two techniques, starting first with the "partial" inversion technique and then looking at batched inversions. The definitions we will give for these techniques below are given at the level of field arithmetic. When we proceed to sketch `pb_inv`, we will offer two definitions: one in this section given at the abstraction-level of field arithmetic, and one in the proceeding section given in terms of $\texttt{SIDH}_\texttt{C}$ syntax.

In the subsections to come, when we are working at the level of field arithmetic we will denote the first and second portions of an arbitrary $x \in \mathbb{F}_{p^2}$ as x_a and x_b respectively. We identify x by saying $x = \{x_a, x_b\}$. Recall from 2.2.1 that both x_a and x_b are valid $\mathbb{F}_p$ elements.

We will express the performance of the comming procedures in terms of the sum number of underlying field operations within them. We denote the computation time for base field arithmetic with bold letters (such as $\mathbf{a}$ for $\mathbb{F}_p$ **a**ddition), and we use bold letters accented with a "closure" bar for extension field arithmetic ($\bar{\mathbf{a}}$ for $\mathbb{F}_{p^2}$ **a**ddition). For example, the performance of some procedure P, which we might write as C_P, may look like the following:

$$C_P = 2\bar{\mathbf{a}} + x\bar{\mathbf{i}} + y\mathbf{m} + \mathbf{s}$$

Which denotes that P is a procedure composed of 2 $\mathbb{F}_{p^2}$ additions, x-many $\mathbb{F}_{p^2}$ inversions, y-many $\mathbb{F}_p$ multiplications, and a single $\mathbb{F}_p$ squaring. We reserve uppercase bold letters for arithmetic over elliptic curve points (such as $\mathbf{A}$ to denote the point-wise addition operation).

3.1.1 $\mathbb{F}_{p^2}$ Inversions done in $\mathbb{F}_p$

There is a simple way in which we can perform one $\mathbb{F}_{p^2}$ inversion by means of doing several $\mathbb{F}_p$ operations. We will begin by considering multiplicative inverses of complex numbers. Fields of the form $\mathbb{F}_{q^2}$ for some prime q are, after all, quadratic extension fields; because of this $\mathbb{F}_{p^2}$ arithmetic is treated, for the most part, analogously to complex number arithmetic.

Consider the complex number $C = a+bi$. We have that $C^{-1} = 1/(a+bi)$, from which we can rationalize the denominator like so:

$$C^{-1} = \frac{1}{(a+bi)} \cdot \frac{(a-bi)}{(a-bi)}$$

$$C^{-1} = \frac{a-bi}{(a+bi)(a-bi)}$$

Here we note that $(a+bi)(a-bi)$ is equivalently (a^2-b^2) and so we can rewrite C^{-1} as the following:

$$C^{-1} = \frac{a-bi}{(a)^2-(bi)^2}$$

$$C^{-1} = \frac{a-bi}{a^2+b^2}.$$

Elements in the quadratic extension of a finite field are treated similarly, such that if we take some element $x = \{x_a, x_b\} \in \mathbb{F}_{p^2}$ for some prime p, we can equivalently represent x as x_a+x_bi and treat arithmetic on x exactly as we would for a complex number (modulo p, of course). From this we can see that x^{-1} can be defined as:

$$x^{-1} = \{\frac{x_a}{x_a^2+x_b^2}, \frac{-x_b}{x_a^2+x_b^2}\}$$

Now it is clear that we can compute the multiplicative inverse of x by computing: the inverse of $x_a^2 + x_b^2$, which is of course an inversion in $\mathbb{F}_p$, and $-x_b$, which is a relatively inexpensive operation (also in the base field). Below, we formulate this technique as algorithm 11, which we refer to as **PartialInv**.

Algorithm 9 – **PartialInv**($x \in \mathbb{F}_{p^2}$)

1: $den \leftarrow x_a^2 + x_b^2$
2: $den_{inv} \leftarrow den^{-1}$
3: $a \leftarrow x_a \cdot den_{inv}$
4: $b \leftarrow -(x_b) \cdot den_{inv}$
5: $x^{-1} \leftarrow \{a, b\}$
6: **return** x^{-1}

Effectively, this procedure reduces one $\mathbb{F}_{p^2}$ inversion to the following operations:

- 2 $\mathbb{F}_p$ squarings – *line 1 of algorithm* 11
- 1 $\mathbb{F}_p$ addition – *line 1 of algorithm* 11
- 1 $\mathbb{F}_p$ inversion – *line 2 of algorithm* 11
- 3 $\mathbb{F}_p$ multiplications – *lines 3 & 4 of algorithm* 11

We say "reduces", but it may not be immediately clear that this technique is computationally favourable over any other approach to computing an $\mathbb{F}_{p^2}$ inversion.

3.1.2 Batching Field Element Inversions

The second technique used in the composition of `pb_inv` reduces arbitrarily many (general) field element inversions to *one* inversion and a linearly scaling amount of multiplcations in the *same* field.

This technique was outlined by Shacham and Boneh in [SB01]. Shacham and Boneh provided several techniques for improving the performance of SSL handshakes, most of which built on the earlier efforts of Amos Fiat in batching multiple RSA decryptions. While somewhat related, Fiat's work admittedly is only applicable to the RSA cryptosystem, and requires additional constraints on the elements being batched.

One improvement offered by Shacham and Boneh, however, is their proposed notion of batching together divisions from across multiple unrelated SSL instances.

Suppose we want to compute the inverses of three elements $x, y, z \in F$ where F is some arbitrary field. The batched division technique allows us to reduce these three inversions to one. The technique can be organized into three phases. In the first phase, all the elements of the batch are multiplied together into one product, yielding $a = xyz$. We refer to this first phase as "upward-percolation". Next, we compute the inverse of a: $a^{-1} = (xyz)^{-1}$, which we refer to as the inversion phase. In the final phase, "downward-percolation", we can compute each individual element's multiplicative inverse as follows:

$$x^{-1} = a^{-1} \cdot (yz)$$

$$y^{-1} = a^{-1} \cdot (xz)$$

$$z^{-1} = a^{-1} \cdot (xy)$$

Let us analyse these phases a little more closely while we generalize to n-many elements. In the upward-percolation phase, constructing a requires $n-1$ multiplications; and so has a complexity of $\mathcal{O}(n)$. The inversion phase requires one field element inversion, and so has complexity of $\mathcal{O}(1)$.

If we implement the downward-percolation phase directly as outlined in the three-element example above, computing every output requires n products each composed of $n-1$ multiplications. These n products are each also multiplied by a^{-1}. This multiplication by a^{-1} can be added to our $n-1$ inversion count resulting in n-many products composed of n multiplications; bringing the complixity of the downward-percolation phase to $\mathcal{O}(n^2)$.

Let B_0 denote the performance of the approach above to batching field element inversions. We have, then, that

$$B_0 = n^2\bar{\mathbf{m}} + (n-1)\bar{\mathbf{m}} + \bar{\mathbf{i}}.$$

This batching proceedure can be thought of as analogous to traditional time-memory tradeoff algorithms. In a general time-memory tradeoff algorithm you can continue to make some linear or polynomial (or otherwise) sacrifice of memory in order to gain some increase in performance. In the batching procedure described above we are in some sense sacrificing some marginal amount of memory to gain an increase in performance, but it is not a tradeoff that we can adjust to our liking.

There is a way, much akin to this time-memory tradeoff strategy, that we can further reduce the execution time of this procedure. In the upward-percolation phase, we currently store in a the product of elements $x_0 \cdot x_1 \cdot ... \cdot x_{n-1}$. Suppose instead that we store in a an *array* (size n) of elements, defined in the following way:

$$a_i = \begin{cases} x_0 & i = 0 \\ a_{i-1} \cdot x_i & \text{otherwise} \end{cases}$$

Equivalently, the elements of this array are

$$a_0 = x_0, \quad a_1 = x_0 \cdot x_1, \quad a_2 = x_0 \cdot x_1 \cdot x_2, \quad ...$$

and so on and so forth up to $n-1$. In the inversion phase we will compute $inv = a_{n-1}^{-1}$; we are still inverting the product of all the elements, but because we have stored the value of the product at every step of the way, we can save on a significant number of operations in the downward-percolation phase.

Going into the final stage of the procedure now, we can compute x_{n-1}^{-1} simply by computing $inv \cdot a_{n-2}$. Moving forward (or backwards, technically), we peel the previously used x_{n-1}^{-1} off of inv by computing $inv := inv \cdot x_{n-1}$ and, with our updated inv, we compute $x_{n-2}^{-1} = inv \cdot a_{n-3}$. We proceed in this fashion until we reach x_0^{-1}, which (if we've been updating inv every step of the way) is simply equal to inv.

We provide an algorithm under the name of **BatchedInv** implementing this approach, generalized to n-many elements (See Algorithm 10). In Algorithm 10, lines 1–3 implement the upward-percolation phase. Line 4 carries out the second phase: the inversion of a_{n-1}. The third and final stage, downward-percolation, occurs from lines 5 to 7.

BatchedInv can be used to reduce n-many $\mathbb{F}_{p^2}$ inversions to the following operations:

Algorithm 10 – **BatchedInv**$(\{x_0, x_1, ..., x_n - 1\} \in \mathbb{F}_{p^2}^n)$

1: $a_0 \leftarrow x_0$
2: **for** `i = 1..(n-1)` **do**
3: $\quad a_i \leftarrow a_{i-1} \cdot x_i$
4: $inv \leftarrow a_{n-1}^{-1}$
5: **for** `i = (n-1)..1` **do**
6: $\quad x_i^{-1} \leftarrow a_{i-1} \cdot inv$
7: $\quad inv \leftarrow inv \cdot x_i$
8: $x_0^{-1} = inv$

- $n - 1$ $\mathbb{F}_{p^2}$ multiplications – *line 2-3 of algorithm* 10
- 1 $\mathbb{F}_{p^2}$ inversion – *line 4 of algorithm* 10
- $2(n-1)$ $\mathbb{F}_{p^2}$ multiplications – *line 5-7 of algorithm* 10

Let B denote the performance of **BatchedInv**.

$$B = 2(n-1)\bar{\mathbf{m}} + (n-1)\bar{\mathbf{m}} + \bar{\mathbf{i}}.$$

Comparing the performance of **BatchedInv** with our initial construction, we see that $B < B_0$ holds when the following holds:

$$2(n-1)\bar{\mathbf{m}} + (n-1)\bar{\mathbf{m}} + \bar{\mathbf{i}} < n^2\bar{\mathbf{m}} + (n-1)\bar{\mathbf{m}} + \bar{\mathbf{i}}$$

$$2(n-1)\bar{\mathbf{m}} < n^2\bar{\mathbf{m}}$$

$$2(n-1) < n^2$$

And so, because n^2 is always larger than $2(n-1)$ for all $n \in \mathbb{R}$, **BatchedInv** outperforms our original roughly-sketched procedure for every possible batch size. This can be checked, if one so wishes, by setting $n^2 = 2(n-1)$, simplifying to $n^2 - 2n + 2 = 0$, and noting that the discriminant $(2^2 - 4 \cdot 2)$ is negative.

3.1.3 Partial Batched Inversions

We have now outlined the following: **PartialInv** as a technique for computing $\mathbb{F}_{p^2}$ inversions by means of $\mathbb{F}_p$ arithmetic, and **BatchedInv** as a technique for batching together arbitrarily many inversion operations. We will now combine these procedures in a (near) trivial manner to achieve the partial batched inversion algorithm.

At first glance, an attempt to meld these two techniques together might be made in the following way:

Algorithm 11 – **PartialBatchedInvAttempt**$(\{x_0, x_1, ..., x_{n-1}\})$

$a \leftarrow$ upward-percolation of elements $\{x_0, x_1, ..., x_{n-1}\}$
$a^{-1} \leftarrow$ **PartialInv**(a)
$\{x_0^{-1}, x_1^{-1}, ..., x_{n-1}^{-1}\} \leftarrow$ downward-percolation of a^{-1}
return $\{x_0^{-1}, x_1^{-1}, ..., x_{n-1}^{-1}\}$

Counting the sum of operations in this proposed approach, we have the following:

- n $\mathbb{F}_{p^2}$ multiplications – *upward-percolation phase*
- 2 $\mathbb{F}_p$ squarings, 1 $\mathbb{F}_p$ addition, 1 $\mathbb{F}_p$ inversion, and 3 $\mathbb{F}_p$ multiplications – *call to* **PartialInv**(a)
- $2n$ $\mathbb{F}_{p^2}$ multiplications – *downward-percolation phase*

To measure the complixity in terms of field operations, denoted C', we can surmize the the total operation count as:

$$C' = (n\bar{\mathbf{m}}) + (2\mathbf{s} + \mathbf{a} + \mathbf{i} + 3\mathbf{m}) + (2n\bar{\mathbf{m}})$$

$$C' = 3n\bar{\mathbf{m}} + 2\mathbf{s} + \mathbf{a} + \mathbf{i} + 3\mathbf{m}$$

Below we provide an alternative approach to building **PartialBatchedInv** that relies on only $\mathbb{F}_p$ operations. Afterward, we show by simply analysis why this approach yields the better performance. This procedure is formalized in a mathematical setting in algorithm 12. We give a precise C function definition in section 3.2.

Algorithm 12 – **PartialBatchedInversion**($\mathbb{F}_{p^2}$ $\{x_0, x_1, ..., x_n - 1\}$)

1: **procedure** PARTIAL_BATCHED_INV($\mathbb{F}_{p^2}$[ ] VEC, $\mathbb{F}_{p^2}$[ ] DEST, INT N)
2:     **for** `i = 0..(n-1)` **do**
3:         $den_i \leftarrow (x_i)_a^2 + (x_i)_b^2$
4:     $a_0 \leftarrow den_0$
5:     **for** `i = 1..(n-1)` **do**
6:         $a_i \leftarrow a_{i-1} \cdot den_i$
7:     $inv \leftarrow inv(a_{n-1})$
8:     **for** `i = n-1..1` **do**
9:         $a_i \leftarrow inv \cdot dest_{i-1}$
10:         $inv \leftarrow inv \cdot den_i$
11:     $a_0 \leftarrow a_{inv}$
12:     **for** `i = 0..(n-1)` **do**
13:         $(xinv_i)_a \leftarrow a_i \cdot (x_i)_a$
14:         $(xinv_i)_b \leftarrow a_i \cdot -(x_i)_b$
15:         $x_i^{-1} \leftarrow \{(xinv_i)_a, (xinv_i)_b\}$
16:     **return** $\{x_0^{-1}, x_1^{-1}, ..., x_n - 1^{-1}\}$

a is a simple auxillary set we use to hold the inverted $\mathbb{F}_p$ elements. After these are all computed via the for-loop on line 8, we can reconstruct $\mathbb{F}_p$

More specifically, the procedure takes us from n $\mathbb{F}_{p^2}$ inversions to:

- $2n$ $\mathbb{F}_p$ squarings
- n $\mathbb{F}_p$ additions
- 1 $\mathbb{F}_p$ inversion
- 3(n-1) $\mathbb{F}_p$ multiplications
- $2n$ $\mathbb{F}_p$ multiplications

And so, with C measuring the performance of **PartialBatchedInversion**, we have

$$C = 2n\mathbf{s} + n\mathbf{a} + \mathbf{i} + 3(n-1)\mathbf{m} + 2n\mathbf{m}$$

We can further simplify C if we presume that a the execution time of squaring is roughly the same as multiplication. Additionally, we can simplify $3(n-1)$ to $3n$ in the spirit of complexitiy theory. With these simplifications we arrive at

$$C \approx 7n\mathbf{m} + n\mathbf{a} + \mathbf{i}$$

Applying the same simplifying assumptions to C', we arrive at

$$C' \approx 3n\bar{\mathbf{m}} + 5\mathbf{m} + \mathbf{a} + \mathbf{i}$$

We note here that an $\mathbb{F}_{p^2}$ multiplication ($\mathbf{M}$) is performed simply by means of 4 $\mathbb{F}_p$ multiplications (again, recall the multiplcation of complex numbers). So we have $\mathbf{M} = 4\mathbf{m}$, and can further simplify C':

$$C' \approx (12n + 5)\mathbf{m} + \mathbf{a} + \mathbf{i}$$

Finally we've simplified C and C' to forms that are more easily compared. Let $\mathbf{P}$ denote the proposition that C runs in fewer operations than C':

$$\mathbf{P} \equiv C < C'$$

$$\mathbf{P} \equiv 7n\mathbf{m} + n\mathbf{a} + \mathbf{i} < (12n + 5)\mathbf{m} + \mathbf{a} + \mathbf{i}$$

Simplifying slightly, we need now to resolve

$$7n\mathbf{m} + n\mathbf{a} < (12n + 5)\mathbf{m} + \mathbf{a}$$

3.1.4 Applicability to $\texttt{SIDH}_\texttt{C}$

Because the work of Yoo et al. was built on top of the original Microsoft SIDH library, all underlying field arithmetic (and as such, pointwise arithmetic) is performed in projective space using Montgomery representation. As was mentioned in the previous chapter, doing such allows us to avoid a great deal of field element inversions. We simply convert to Montgomery representation at the beginning of heavy arithmetic, perform the desired operations, and then convert back to standard representation when complete. Throughout the codebase, (most noticeably in `kex.c`) we can see an analogous approach taken in for pointwise arithmetic, by first converting to a projective space representation, performing operations, and then converting back to affine coordinates.

The downside of this (for our work) is that the number opportunities for implementing the batched inversion algorithm becomes greatly limited.

3.2 Implementation Details

We will now take the work of the previous section and explain in detail how it can be applied to the Yoo et al. signature layer of $\texttt{SIDH}_\texttt{C}$. We begin first by transcribing **PartialBatchedInversion** to its programmatic variant, `pb_inv`. In the subsections to come, we offer explainations for some of the design choices made in the implementation of `pb_inv`, both at a micro level (algorithmic details of the procedure) and a macro level (specifics of implementation within $\texttt{SIDH}_{\texttt{C}}$).

With Algorithm 13 we provide an explicit C definition for the function `pb_inv`. For descriptions of the functions called in this procedure, the reader can refer to section 2.6.3. For explicit definitions of some of these functions, the reader can refer to appendix [insert appendix].

Algorithm 13 – `pb_inv`

procedure PARTIAL_BATCHED_INV($\mathbb{F}_{p^2}[\ ]$ VEC, $\mathbb{F}_{p^2}[\ ]$ DEST, INT N)
    initialize $\mathbb{F}_p$ $den[n]$
    for `i = 0..(n-1)` **do**
        $den[i] \leftarrow a[i][0]^2 + a[i][1]^2$
    $a[0] \leftarrow den[0]$
    for `i = 1..(n-1)` **do**
        $a[i] \leftarrow$ `a[i-1]*den[i]`
    $a_{inv} \leftarrow$ `inv(a[n-1])`
    for `i = n-1..1` **do**
        $a[i] \leftarrow a_{inv} * dest[i-1]$
        $a_{inv} \leftarrow a_{inv} * den[i]$
    $dest[0] \leftarrow a_{inv}$
    for `i = 0..(n-1)` **do**
        $dest[i][0] \leftarrow a[i] * vec[i][0]$
        $vec[i][1] \leftarrow -1 * vec[i][1]$
        $dest[i][1] \leftarrow a[i] * vec[i][1]$

3.2.1 Parallelizing Signatures

Because every 2λ iteration of the **Sign** and **Verify** procedures are entirely independent of each other, these functions present themselves as embarrassingly parallel.[1]

Recall from section 2.6.4 the table of functions that can be found in `SIDH_signature`. `isogey_sign` acts as the entry point for `Sign` and spawns a POSIX thread for every instance of the for-loop; each one calling `sign_thread` which performs Bob's interaction with Randall. Verification proceeds analogously; `isogeny_verify` is executed and spawns a POSIX threads executing `verify_thread` until all 2λ iterations are complete.

This parallelization of the signature scheme was the approach taken by Yoo et al. in their original implementation. It also lends itself rather nicely

`inv_4_way`.

`j_inv`.

[1]in the field of high performance computing, a problem that is trivially parallizable is often referred to as *embarrassingly* parallizable.

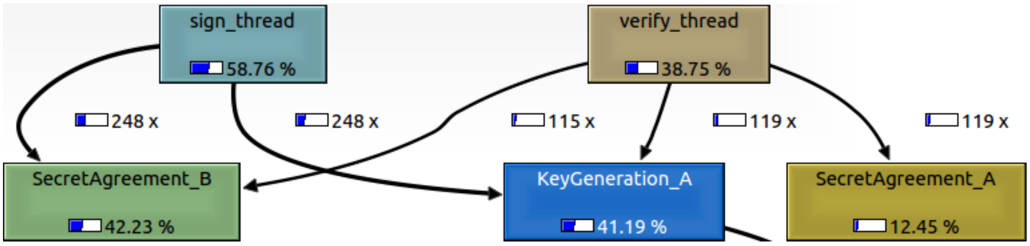

Figure 3.1: ¡Caption here¿

3.2.2 Security Concerns

Recall the notion of a general side-channel attack: A side-channel attack is performed when an unauthorized individual is able to acquire information by measuring properties of the physical implementation of the system at hand. This can be done by analyzing the power consumption, timing properties, or electromagnetic leaks of a CPU while it operates on (or generates) confidential information.

In the context of information security, algorithms for performing operations over mathematical objects can be said to fall under one of two categories: *constant time* and *non-constant time* algorithms. Constant time algorithms are designed to protect confidential information from side-channel attacks, but come at the cost of computational efficiency.

In the $\mathtt{SIDH_C}$ library, there are two distinct functions for computing field element inversions: `fp2inv751_mont` and `fp2inv751_mont_bingcd`. `fp2inv751_mont_bingcd` performs inversion by means of the binary GCD (greatest common denominator) algorithm, and is a *non-constant time* implementation. `fp2inv751_mont` is a *constant time* implementation, and as such runs slower than `fp2inv751_mont_bingcd` in nearly all cases, but protects against timing based side-channel attacks. They perform comparatively as such:

Procedure	Performance in clock cyckes	System A With Batching
`fp2inv751_mont`	68,881,331	68,881,331
`fp2inv751_mont_bingcd`	15,744,477,032	15,565,738,003

Take for example some private data c being manipulated or operated on by some algorithm $\mathbf{A}$. In order to be entirely certain that c in $\mathbf{P}(c)$ is not vulnerable to *any* imagineable side-channel attack it must be the case that the structure of $\mathbf{P}$ does not in anyway depend on the information stored in c.

Let us look more closely at the elements that are inverted by `pb_inv`. In `KeyGeneration_A` the elements being passed to `pb_inv` are simply the constituents of Randall's public key for every iteration of the signing procedure. Because the data being operated on is publicly available information, we needn't consider if it is vulnerable to side-channel analysis.

In `SecretAgreement_A`, `pb_inv` is called by `j_inv`, which produces the j-invariant of a particular curve. When `SecretAgreement_A` is used in the context of SIDH key exchange, `j_inv` is used to compute the shared secret between party members A and B, and so would need to be protected against side-channel attacks. This is not the case in the context of signatures, however. Recall the structure of SIDH signatures: every signature includes, for every 2λ iterations, the commitment E_1, which is precisely the shared secret between the signer and Randall. And so, every use of `pb_inv` *in the context of signatures* does not require extra precautions to side-channel analysis. The same is true for `SecretAgreement_B`.

While it is not the case for our implementations that the data on which `pb_inv` operates would find itself the subject of side-channel analysis, it is not difficult to imagine a scenario where this might be the case.

3.3 Results

Modulus Size	Regular Batch	Partial Batched Inversion	Unbatched
32 Inversion	0.033685	0.00080204	
64 Inversion	0.00033685	0.00080204	
128 Inversion	0.00033685	0.00080204	
256 Inversion	0.00033685	0.00080204	
512 Inversion	0.00033685	0.00080204	
1024 Inversion	0.00033685	0.00080204	
2048 Inversion	0.00033685	0.00080204	

Modulus Size	Regular Batch	Partial Batched Inversion	Unbatched
32	0.00033685	0.00080204	
64	0.00033685	0.00080204	
128	0.00033685	0.00080204	
256	0.00033685	0.00080204	
512	0.00033685	0.00080204	
1024	0.00033685	0.00080204	
2048	0.00033685	0.00080204	

Modulus Size	Regular Batch	Partial Batched Inversion	Unbatched
32	0.00033685	0.00080204	
64	0.00033685	0.00080204	
128	0.00033685	0.00080204	
256	0.00033685	0.00080204	
512	0.00033685	0.00080204	
1024	0.00033685	0.00080204	
2048	0.00033685	0.00080204	

Two different machines were used for benchmarking. System A denotes a single-core, 1.70 GHz Intel Celeron CPU. System B denotes a quad-core, 3.1 GHz AMD A8-7600.

The two figures below provide benchmarks for KeyGen, Sign, and Verify procedures with both batched partial inversion implemented (in the previously mentioned locations) and not implemented. All benchmarks are averages computed from 100 randomized sample runs. All results are measured in clock cycles.

Procedure	System A Without Batching	System A With Batching
KeyGen	68,881,331	68,881,331
Signature Sign	15,744,477,032	15,565,738,003
Signature Verify	11,183,112,648	10,800,158,871

Procedure	System B Without Batching	System B With Batching
KeyGen	84,499,270	84,499,270
Signature Sign	10,227,466,210	10,134,441,024
Signature Verify	7,268,804,442	7,106,663,106

System A: With inversion batching turned on we notice a 1.1 % performance increase for key signing and a 3.5 % performance increase for key verification.

System B: With inversion batching turned on we a observe a 0.9 % performance increase for key signing and a 2.3 % performance increase for key verification.

3.3.1 Analysis

It should first be noted that, because our benchmarks are measured in terms of clock cycles, the difference between our two system clock speeds should be essentially ineffective.

In the following table, "Batched Inversion" signifies running `pb_inv` on 248 $\mathbb{F}_{p^2}$ elements.

Procedure	Performance
Batched Inversion	1721718
$\mathbb{F}_{p^2}$ Montgomery Inversion	874178

The following are all measured in clock cycles, as the computed average of 1000 distinct executions:

Modulus Size	Multiplication	Inversion Time
32	0.00033685	0.00080204
64	0.00033685	0.00080204
128	0.00033685	0.00080204
256	0.00033685	0.00080204
512	0.00033685	0.00080204
1024	0.00033685	0.00080204
2048	0.00033685	0.00080204

Do performance increases observed make sense?

3.3.2 Remaining Opportunities

There are two functions called in the isogeny signature system that perform a $\mathbb{F}_{p^2}$ inversion: `j_inv` and `inv_4_way`. These functions are called once in SecretAgreement and KeyGeneration operations respectively. SecretAgreement and KeyGeneration are in turn called from each signing and verification thread.

This means that in the signing procedure there are 2 opportunities for implementing batched partial-inversion with a batch size of 248 elements. In the verify procedure, however, there are 3 opportunities for implementing batched inversion with a batch size of roughly 124 elements.

Another avenue for implementation of this procedure, much in line with Fiat's original RSA batch, lies in . We discuss this approach in greater detail in section 5.2.

Chapter 4

Compressing Signatures

4.1 SIDH Key Compression Background

We discussed rejection sampling A values from signature public keys until we found an A that was also the x-coord of a point. After some simple analysis, however, we found that it was extremely unlikely for A to be a point on the curve.

4.1.1 Motivation & Overview

4.1.2 Construction of Bases

4.1.3 Pohlig-Hellman

4.1.4 Decompression

4.2 Implementation Details

4.2.1 Tailoring Compression for Signatures

4.2.2 Decompressing $\psi(S)$

4.3 Results

4.4 Analysis

Chapter 5

Discussion & Conclusion

5.1 Results & Comparisons

5.2 Additional Opportunities for Batching

5.3 Future Work

¡Conclusion here¿

Acknowledgments

¡Acknowledgements here¿

¡Name here¿

¡Month and Year here¿
National Institute of Technology Calicut

References

[ARU14] Andris Ambainis, Ansis Rosmanis, and Dominique Unruh. Quantum attacks on classical proof systems (the hardness of quantum rewinding). pages 474–483, 2014.

[CLN16] Craig Costello, Patrick Longa, and Michael Naehrig. Efficient algorithms for supersingular isogeny diffie-hellman. IACR-CRYPTO-2016, 2016.

[Cos] Craig Costello. Pairings for beginners.

[FJP12] Luca De Feo, David Jao, and Jerome Plut. Towards quantum-resistant cryptosystems from supersingular elliptic curvce isogenies. 2012.

[JS14] David Jao and Vladimir Soukharev. Isogeny-based quantum resistant undeniable signatures. 2014.

[Kat10] Jonathan Katz. *Digital Signatures*. Springer, 2010.

[Mon85] Peter L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 1985.

[PS96] David Pointcheval and Jacques Stern. Security proofs for signature schemes. 1996.

[SB01] Hovav Shacham and Dan Boneh. Improving ssl handshake performance via batching. 2001.

[SC16] M. Seshadri Srinath and V. Chandrasekaren. Isogeny-based quantum-resistant undeniable blind signature scheme. 2016.

[STW12] Xi Sun, Haibo Tian, and Yumin Wang. Toward quantum-resistant strong designated verifier signature from isogenies. 2012.

[Unr15] Dominique Unruh. Non-interactive zero-knowledge proofs in the quantum random oracle model. 2015.

[YAJ+12] Youngho Yoo, Reza Azarderakhsh, Amir Jalali, David Jao, and Vladimir Soukharev. A post-quantum digital signature scheme based on supersingular isogenies. 2012.