

# 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

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Background and Recent Research . . . . .	1
1.1.1	any sub section here . . . . .	1
1.1.2	Literature Survey . . . . .	1
1.2	Layout of Paper . . . . .	1
1.3	Motivation . . . . .	1
<b>2</b>	<b>Technical Background</b>	<b>2</b>
2.1	Algebraic Geometry & Isogenies . . . . .	3
2.1.1	Elliptic Curves . . . . .	3
2.1.2	Isogenies & Their Properties . . . . .	4
2.2	Supersingular Isogeny Diffie-Hellman . . . . .	4
2.2.1	Public Parameters . . . . .	4
2.2.2	Modular Breakdown . . . . .	5
2.2.3	Security Assumptions . . . . .	6
2.2.4	Zero-Knowledge Proof of Identity . . . . .	6
2.3	Fiat-Shamir Construction . . . . .	6
2.3.1	Unruh’s Post-Quantum Adaptation . . . . .	6
2.4	Isogeny Based Signatures . . . . .	6
2.4.1	Modular Breakdown . . . . .	6
<b>3</b>	<b>Batching Operations for Isogenies</b>	<b>9</b>
3.1	Batching Procedure in Detail . . . . .	9
3.1.1	Projective Space . . . . .	9
3.1.2	Remaining Opportunities . . . . .	9
3.2	Implementation . . . . .	10
3.3	Results . . . . .	10
3.3.1	Analysis . . . . .	11
<b>4</b>	<b>Compressed Signatures</b>	<b>12</b>
4.1	Compression of Public Keys . . . . .	12
4.1.1	Sub-section title . . . . .	12
4.1.2	Sub-section title . . . . .	12
4.1.3	Sub-section title . . . . .	12
4.1.4	Sub-section title . . . . .	12
4.1.5	Sub-section title . . . . .	12
4.2	Implementation . . . . .	12
4.3	Results . . . . .	12

<b>5</b>	<b>Discussion &amp; Conclusion</b>	<b>13</b>
5.1	Results & Comparisons . . . . .	13
5.2	Additional Opportunities for Batching . . . . .	13
5.3	Future Work . . . . .	13
	<b>Acknowledgements</b>	<b>14</b>
	<b>References</b>	<b>15</b>

# List of Figures

2.1	Temporary. To be replaced with illustration of EC group operation. . . .	3
3.1	¡Caption here¿ . . . . .	10

# Chapter 1

## Introduction

### 1.1 Background and Recent Research

1.1.1 ;any sub section here;

1.1.2 Literature Survey

### 1.2 Layout of Paper

;Sub-subsection title;

some text<sup>[1]</sup>, some more text

;Sub-subsection title;

even more text<sup>1</sup>, and even more.

### 1.3 Motivation

---

<sup>1</sup>;footnote here;

# Chapter 2

## Technical Background

Over the course of the past decade, elliptic curve cryptography (henceforth referred to as 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 mathematical concepts as ECC, it simultaneously draws from a slightly more complicated space of niche algebraic notions. Much of this chapter will be dedicated to illuminating these notions in a manner that should be digestible for those without serious background in algebraic geometry, or abstract algebra in general.

This chapter will cover the following preliminary topics: isogenies and their relevant properties, Supersingular Isogeny Diffie-Hellman and its related protocols, the Fiat-Shamir construction and its quantum-safe adaptation, and finally the current landscape of isogeny based signature schemes.

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 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, laying bare the underlying isogeny-level procedures and algorithms that will be necessary for understanding the protocols to come in detail. Another task of this section will be to introduce the SIDH C library released by Microsoft Research, on top of which the core contributions of this thesis are implemented. This subsection will end with a thorough briefing and analysis of the closely related zero-knowledge proof of identity (ZKPoI) isogeny protocol proposed in the original JDP paper, as it is necessary for understanding the isogeny based signature scheme presented by Yoo et. Al.

In section 2.3 we will discuss the Fiat-Shamir transformation; a technique which, given a secure interactive proof of knowledge (/proof of identity), creates a secure digital signature scheme. We will also look at the quantum-secure adaptation published by Unruh, (applying a non-quantum-resistant transform to a quantum-resistant primitive would be rather frivolous,)

Finally, the last section of this chapter will be dedicated to covering current isogeny-based signature schemes - the topic of which this dissertation is mainly concerned. We will primarily discuss the signature scheme of Yoo et. Al, which is a near direct application of Unruh's Fiat-Shamir adaptation to the SIDH zero-knowledge proof of identity to be discussed at the end of section 2.2.

## 2.1 Algebraic Geometry & Isogenies

*Groups & Varieties.* A *group* is a 2-tuple composed of a set of elements and a corresponding group operation. Given some group defined by the set  $G$  and the group operation  $*$  (written as  $(G, *)$ ) it is typical to refer to the group simply as " $G$ ".  $G$  is said to be an *abelian* group if, in addition to the four traditional group axioms,  $G$  satisfies the condition of commutativity. More formally: for some group  $G$  with group operation  $\cdot$ , we say  $G$  is an abelian group iff  $x \cdot y = y \cdot x \ \forall x, y \in G$ . A group which is also an algebraic variety is referred to as an **algebraic group**.

An algebraic group  $G_a$  is defined over a field  $K$  if each element  $e \in G_a$  is defined over  $K$  and  $f_G, \phi_G$  are also 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  $G_a(K)$ .

*Morphisms.* A *morphism* is a structure-preserving map. Morphisms can be thought of as functions from some mathematical structure ( $A$ ) to another ( $B$ ). More specifically, in the domain of algebraic geometry, we will be dealing with the notion of a *group homomorphism*, defined as follows:

**Definition 1** (Group Homomorphism). For two groups  $G$  and  $H$  with respective group operations  $*$  and  $\cdot$ , a *group homomorphism* is a mapping  $h : G \rightarrow H$  such that  $\forall u, v \in G$  the following holds:

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

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

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

Furthermore, an *endomorphism* is a special type of morphism in which the domain and the codomain are the same mathematical object.

*Field Extensions.*

### 2.1.1 Elliptic Curves

An elliptic curve is an algebraic curve definable by an equation of the form  $y^2 = x^3 + ax + b$ . We will see later other possible algebraic representations of elliptic curves which have varying benefits. The point at infinity.

algebra stuff: elliptic curves over finite fields are abelian varieties. group to abelian group to abelian variety to morphism/homomorphism to isogenies are homomorphisms over abelian varieties.

blah blah the points on an elliptic curve defined over some finite field form an abelian variety

When referring to curves as abelian varieties defined over a field, we will write them as  $E_\alpha[K]$ , for some  $\alpha$  instance of a curve and some field  $K$ . If we are only concerned



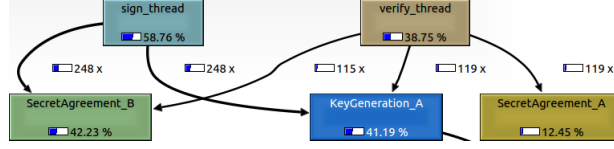


Figure 2.1: Temporary. To be replaced with illustration of EC group operation.

with the geometric properties of the curve, or curves as distinct elements of some group structure, it will suffice to write  $E_\alpha$ .

*Supersingular Curves.* An elliptic curve can be either *ordinary* or *supersingular*. There are several equivalent ways to define supersingular curves (and thus the distinction between them and ordinary curves,)

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

## 2.1.2 Isogenies & Their Properties

An isogeny is a type of function or map which is defined in relation to the earlier discussed concept of a morphism[link]. The formal definition is as follows:

**Definition 2** (Isogeny). Let  $G$  and  $H$  be algebraic groups[ref]. An isogeny is a morphism[ref]  $h : G \rightarrow H$  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*[ref] between  $G$  and  $H$ . Because of this, isogenies over elliptic curves (and other abelian varieties) inherit certain characteristics.

For an isogeny  $h : E_1 \rightarrow E_2$  defined over elliptic curves  $E_1$  and  $E_2$ , the following holds:

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

## 2.2 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.

### 2.2.1 Public Parameters

All three of the isogeny-based protocols proposed by Jao, DeFeo, and Plut have the same initialization and general set of public parameters. We will cover these now.

As the name suggests, SIDH protocols work over supersingular curves (of a smooth order). Let  $\mathbb{F}_q = \mathbb{F}_{p^2}$  be the finite field over which our curves are defined.  $p$  is a prime defined as follows:

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

Wherein  $\ell_A$  and  $\ell_B$  are small primes (typically 2 & 3, respectively) and  $f$  is a cofactor ensuring the primality of  $p$ . We then define a global parameter curve  $E$  with cardinality  $(\ell_A^{e_A} \ell_B^{e_B} f)^2$ .

## 2.2.2 Modular Breakdown

Before drawing parallels between the SIDH protocol and its C implementation, we will first offer a brief digest of the libraries unit organization.

Header files:	
File	Description of contents
SIDH.h	
SIDH_internal.h	
SIDH_api.h	
keccak.h	
Source files:	
File	Description of contents
SIDH.c	
kex.c	
ec_isogeny.c	
fpx.c	
keccak.c	
sha256.c	
Test files:	
File	Description of contents
kex_tests.c	Correctness & performance tests for key exchange
arith_tests.c	Correctness & performance testing of field arithmetic

### Ephemeral Key Generation – Alice:

Key generation for Alice		EphemeralKeyGeneration_A	
Location	Efficient Algo's Appendix A	Location	kex.c
Input	$x_{P_B}, x_{P_A}, y_{P_A},$ $SK_{Alice} = m_A \cdot l_A$	Input	unsigned char* PrivateKeyA, unsigned char* PublicKeyA, PCurveIsogenyStruct CurveIsogeny, invBatch* batch
Output	$PK_{Alice} = [x_{\Phi_A}(P_B), x_{\Phi_A}(Q_B), x_{\Phi_A}(Q_B - P_B)]$	Output	publickey_t PublicKeyA, digit_t PrivateKeyA

### Ephemeral Key Generation – Bob:

Key generation for Bob		EphemeralKeyGeneration_B	
Location	Efficient Algo's Appendix A	Location	kex.c
Input	$x_{P_A}, x_{P_B}, y_{P_B},$ $SK_{Bob} = m_B \cdot l_B$	Input	unsigned char* PrivateKeyB, unsigned char* PublicKeyB, PCurveIsogenyStruct CurveIsogen invBatch* batch
Output	$PK_{Bob} = [x_{\Phi_B}(P_A), x_{\Phi_B}(Q_A), x_{\Phi_B}(Q_A - P_A)]$	Output	publickey_t PublicKeyB, digit_t PrivateKeyB
<b>Ephemeral Secret Agreement – Alice:</b> Shared secret algorithm for Alice		EphemeralSecretAgreement_A	
Location	Efficient Algo's Appendix A	Location	kex.c
Input	$PK_{Bob} = [x_{\Phi_B}(P_A), x_{\Phi_B}(Q_A), x_{\Phi_B}(Q_A - P_A)]$ $SK_{Alice} = m_A \cdot l_A$	Input	const unsigned char* PrivateKeyA const unsigned char* PublicKeyB unsigned char* SharedSecretA, PCurveIsogenyStruct CurveIsogen invBatch* batch
Output	A shared secret j-invariant of an elliptic curve	Output	f2elm_t SharedSecretA,
<b>Ephemeral Secret Agreement – Bob:</b> Shared secret algorithm for Bob		EphemeralSecretAgreement_B	
Location	Efficient Algo's Appendix A	Location	kex.c
Input	$PK_{Alice} = [x_{\Phi_A}(P_B), x_{\Phi_A}(Q_B), x_{\Phi_A}(Q_B - P_B)]$ $SK_{Bob} = m_B \cdot l_B$	Input	const unsigned char* PrivateKeyE const unsigned char* PublicKeyA unsigned char* SharedSecretB, PCurveIsogenyStruct CurveIsogen invBatch* batch
Output	A shared secret j-invariant of an elliptic curve	Output	f2elm_t SharedSecretB,

### 2.2.3 Security Assumptions

### 2.2.4 Zero-Knowledge Proof of Identity

Recall the notion of a simple identification scheme:

## 2.3 Fiat-Shamir Construction

The Fiat-Shamir Construction (sometimes referred to as the Fiat-Shamir Heuristic,) is used

### 2.3.1 Unruh's Post-Quantum Adaptation

## 2.4 Isogeny Based Signatures

Now that we've introduced the zero-knowledge proof of identity scheme from [REFERENCE] as well as Unruh's quantum-safe Fiat-Shamir adaption, the isogeny based sig-

nature scheme presented by Yoo et. Al is a near-trivial application of the latter to the former.

### 2.4.1 Modular Breakdown

The isogeny based signature scheme presented by Yoo et. Al is defined, in the traditional manner, by a tuple of algorithms. Namely, the scheme is defined by the tuple (KeyGen, Sign, Verify) with each algorithm loosely defined as follows:

**KeyGen()**: Select a random point  $S$  of order  $\ell_A^{e_A}$ , compute the isogeny  $\phi : E \rightarrow E/\langle S \rangle$ . Return (pk, sk) where  $\text{pk} = (E/\langle S \rangle, \phi(P_B), \phi(Q_B))$  and  $\text{sk} = S$ .

**Sign()**:

**Verify()**:

Shortly after, the following, more in-depth algorithms are given as definitions:

---

#### Algorithm 1 KeyGen( $\lambda$ )

---

- 1: Pick a random point  $S$  of order  $\ell_A^{e_A}$
  - 2: Compute the isogeny  $\phi : E \rightarrow E/\langle S \rangle$
  - 3:  $\text{pk} \leftarrow (E/\langle S \rangle, \phi(P_B), \phi(Q_B))$
  - 4:  $\text{sk} \leftarrow S$
  - 5: **return** (pk,sk)
- 

---

#### Algorithm 2 Sign(sk, $m$ )

---

- 1: **for**  $i = 1 \dots 2\lambda$  **do**
  - 2:   Pick a random point  $R$  of order  $\ell_B^{e_B}$
  - 3:   Compute the isogeny  $\psi : E \rightarrow E/\langle R \rangle$
  - 4:   Compute either  $\phi' : E/\langle R \rangle \rightarrow E/\langle R, S \rangle$  or  $\psi' : E/\langle S \rangle \rightarrow E/\langle R, S \rangle$
  - 5:    $(E_1, E_2) \leftarrow (E/\langle R \rangle, E/\langle R, S \rangle)$
  - 6:    $\text{com}_i \leftarrow (E_1, E_2)$
  - 7:    $\text{ch}_{i,0} \leftarrow_R \{0, 1\}$
  - 8:    $(\text{resp}_{i,0}, \text{resp}_{i,1}) \leftarrow ((R, \phi(R)), \psi(S))$
  - 9:   **if**  $\text{ch}_{i,0} = 1$  **then**
  - 10:      $\text{swap}(\text{resp}_{i,0}, \text{resp}_{i,1})$
  - 11:    $h_{i,j} \leftarrow G(\text{resp}_{i,j})$
  - 12:  $J_1 \parallel \dots \parallel J_{2\lambda} \leftarrow H(\text{pk}, m, (\text{com}_i)_i, (\text{ch}_{i,j})_{i,j}, (h_{i,j})_{i,j})$
  - 13: **return**  $\sigma \leftarrow ((\text{com}_i)_i, (\text{ch}_{i,j})_{i,j}, (h_{i,j})_{i,j}, (\text{resp}_{i,J_i})_i)$
-

---

**Algorithm 3** Verify(pk,  $m$ ,  $\sigma$ )

---

```
1:  $J_1 \parallel \dots \parallel J_{2\lambda} \leftarrow H(pk, m, (\text{com}_i)_i, (\text{ch}_{i,j})_{i,j}, (h_{i,j})_{i,j})$ 
2: for  $i = 0 \dots 2\lambda$  do
3:   check  $h_{i,J_i} = G(\text{resp}_{i,J_i})$ 
4:   if  $\text{ch}_{i,J_i} = 0$  then
5:     Parse  $(R, \phi(R)) \leftarrow \text{resp}_{i,J_i}$ 
6:     check  $(R, \phi(R))$  have order  $\ell_B^{e_B}$ 
7:     check  $R$  generates the kernel of the isogeny  $E \rightarrow E_1$ 
8:     check  $\phi(R)$  generates the kernel of the isogeny  $E/\langle S \rangle \rightarrow E_2$ 
9:   else
10:    Parse  $\psi(S) \leftarrow \text{resp}_{i,J_i}$ 
11:    check  $\psi(S)$  has order  $\ell_A^{e_A}$ 
12:    check  $\psi(S)$  generates the kernel of the isogeny  $E_1 \rightarrow E_2$ 
13: if all checks succeed then
14:   return 1
```

---

If we transcribe the above to the language of the Microsoft SIDH API, we have in essence the following:

---

**Algorithm 4** KeyGen( $\lambda$ )

---

```
1: (pk, sk)  $\leftarrow$  KeyGeneration.B()
2: return (pk, sk)
```

---

---

**Algorithm 5** Sign(sk,  $m$ )

---

```
1: for  $i = 1 \dots 2\lambda$  do
2:    $(, R, \psi) \leftarrow \text{KeyGeneration.A}(E)$ 
3:    $E_1 \leftarrow E/\langle R \rangle$ 
4:    $(E_2, E/\langle R, S \rangle) \leftarrow \text{SecretAgreement.B}()$ 
5:    $(E_1, E_2) \leftarrow (E/\langle R \rangle, E/\langle R, S \rangle)$ 
6:    $\text{com}[i] \leftarrow (E_1, E_2)$ 
7:    $\text{ch}[i] \leftarrow_R \{0, 1\}$ 
8:    $(\text{resp}[i]_0, \text{resp}[i]_1) \leftarrow ((R, \phi(R)), \psi(S))$ 
9:  $J_1 \parallel \dots \parallel J_{2\lambda} \leftarrow H(pk, m, (\text{com}_i)_i, (\text{ch}_i)_i, (h_{i,j})_{i,j})$ 
10: return  $\sigma \leftarrow ((\text{com}_i)_i, (\text{ch}_{i,j})_{i,j}, (h_{i,j})_{i,j}, ((\text{resp})[J_i])$ 
```

---

# Chapter 3

## Batching Operations for Isogenies

### 3.1 Batching Procedure in Detail

One of our main contributions is the embedding of a low-level  $\mathbb{F}_{p^2}$  procedure into Microsofts pre-existing SIDH library. The procedure in question reduces arbitrarily many unrelated/potentially parallel  $\mathbb{F}_{p^2}$  inversions to a sequence of  $\mathbb{F}_p$  multiplications & additions, as well as one  $\mathbb{F}_p$  inversion.

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

The procedure is as follows:

#### 3.1.1 Projective Space

Because the work of Yoo et al. was built on top of the original Microsoft SIDH library, all underlying field operations (and isogeny arithmetic) are performed in projective space. Doing field arithmetic in projective space allows us to avoid many inversion operations. The downside of this (for our work) is that the number opportunities for implementing the batched inversion algorithm becomes greatly limited.

#### 3.1.2 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.

---

**Algorithm 6** Batched Partial-Inversion

---

```
1: procedure PARTIAL_BATCHED_INV( $\mathbb{F}_{p^2}[\ ]$  VEC,  $\mathbb{F}_{p^2}[\ ]$  DEST, INT N)
2:   initialize  $\mathbb{F}_p$  den[n]
3:   for i = 0..(n-1) do
4:     den[i]  $\leftarrow a[i][0]^2 + a[i][1]^2$ 
5:   a[0]  $\leftarrow$  den[0]
6:   for i = 1..(n-1) do
7:     a[i]  $\leftarrow a[i-1] * \text{den}[i]$ 
8:   ainv  $\leftarrow$  inv(a[n-1])
9:   for i = n-1..1 do
10:    a[i]  $\leftarrow a_{\text{inv}} * \text{dest}[i-1]$ 
11:    ainv  $\leftarrow a_{\text{inv}} * \text{den}[i]$ 
12:  dest[0]  $\leftarrow a_{\text{inv}}$ 
13:  for i = 0..(n-1) do
14:    dest[i][0]  $\leftarrow a[i] * \text{vec}[i][0]$ 
15:    vec[i][1]  $\leftarrow -1 * \text{vec}[i][1]$ 
16:    dest[i][1]  $\leftarrow a[i] * \text{vec}[i][1]$ 
```

---

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.

## 3.2 Implementation

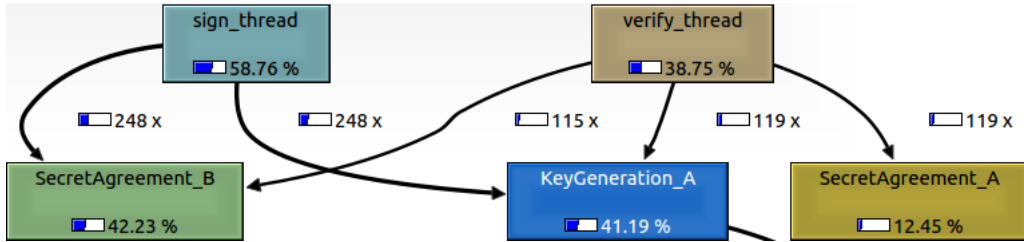


Figure 3.1: ;Caption here;

## 3.3 Results

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 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 the batched partial-inversion procedure on 248  $\mathbb{F}_{p^2}$  elements. The procedure uses the binary GCD  $\mathbb{F}_p$  inversion function which, unlike regular  $\mathbb{F}_{p^2}$  montgomery inversion, is not constant time.

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

Do performance increases observed make sense?



# Chapter 4

## Compressed Signatures

### 4.1 Compression of Public Keys

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 ;Sub-section title;

#### 4.1.2 ;Sub-section title;

some text[2], some more text

#### 4.1.3 ;Sub-section title;

#### 4.1.4 ;Sub-section title;

Refer figure 3.1.

#### 4.1.5 ;Sub-section title;

### 4.2 Implementation

### 4.3 Results

# 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

- [1] iName of the reference here<sub>i</sub>, <urlhere>
- [2] iName of the reference here<sub>i</sub>, <urlhere>