

# 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	Isogenies . . . . .	2
2.1.1	¶Sub-section title¶ . . . . .	2
2.1.2	¶Sub-section title¶ . . . . .	2
2.1.3	¶Sub-section title¶ . . . . .	2
2.1.4	¶Sub-section title¶ . . . . .	2
2.2	SIDH . . . . .	2
2.2.1	Zero-Knowledge Proof of Identity . . . . .	2
2.3	Fiat-Shamir Heuristic . . . . .	2
2.3.1	Unruh’s Construction . . . . .	2
2.4	Isogeny Based Signatures . . . . .	2
<b>3</b>	<b>Batching Operations for Isogenies</b>	<b>5</b>
3.1	Batching Procedure in Detail . . . . .	5
3.1.1	Projective Space . . . . .	5
3.1.2	Remaining Opportunities . . . . .	5
3.2	Implementation . . . . .	6
3.3	Results . . . . .	6
3.3.1	Analysis . . . . .	7
<b>4</b>	<b>Compressed Signatures</b>	<b>8</b>
4.1	Compression of Public Keys . . . . .	8
4.1.1	¶Sub-section title¶ . . . . .	8
4.1.2	¶Sub-section title¶ . . . . .	8
4.1.3	¶Sub-section title¶ . . . . .	8
4.1.4	¶Sub-section title¶ . . . . .	8
4.1.5	¶Sub-section title¶ . . . . .	8
4.2	Implementation . . . . .	8
4.3	Results . . . . .	8

<b>5</b>	<b>Discussion &amp; Conclusion</b>	<b>9</b>
5.1	Results & Comparisons . . . . .	9
5.2	Additional Opportunities for Batching . . . . .	9
5.3	Future Work . . . . .	9
	<b>Acknowledgements</b>	<b>10</b>
	<b>References</b>	<b>11</b>

# List of Figures

3.1	¡Caption here¿ . . . . .	6
4.1	¡Caption here¿ . . . . .	8

# 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

### 2.1 Isogenies

#### 2.1.1 ;Sub-section title;

#### 2.1.2 ;Sub-section title;

some text[2], some more text

#### 2.1.3 ;Sub-section title;

#### 2.1.4 ;Sub-section title;

Refer figure 4.1.

### 2.2 SIDH

#### 2.2.1 Zero-Knowledge Proof of Identity

### 2.3 Fiat-Shamir Heuristic

#### 2.3.1 Unruh's Construction

### 2.4 Isogeny Based Signatures

---

**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** ( $\text{pk}, \text{sk}$ )
-

---

**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:  $J_1 \parallel \dots \parallel J_{2\lambda} \leftarrow H(pk, m, (\text{com}_i)_i, (\text{ch}_{i,j})_{i,j}, (h_{i,j})_{i,j})$ 
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:  $(pk, sk) \leftarrow \text{KeyGeneration.B}()$ 
2: return  $(pk, sk)$ 
```

---



---

**Algorithm 5**  $\text{Sign}(\text{sk}, m)$ 

---

```
1: for  $i = 1..2\lambda$  do
2:    $(E, R, \psi) \leftarrow \text{KeyGeneration.A}()$ 
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

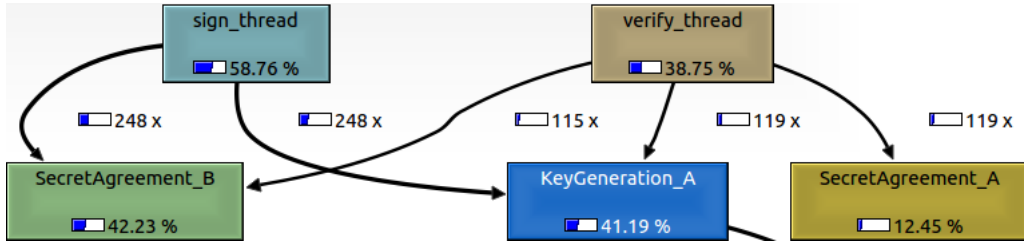


Figure 3.1: ;Caption here;

## 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

#### 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 4.1.

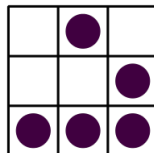


Figure 4.1: ;Caption here;

#### 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>