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 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[1], some more text

¡Sub-subsection title¿

even more text[1], and even more.

1.3 Motivation

[1] ¡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 digestable 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 procotols, 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 isogeny protocol proposed in the original JDP paper, as it is necessary for understanding the isogeny based signature scheme presented by Yoo et. Al.

2.1 Algebraic Geometry & Isogenies

Let us begin with some group G. G is said to be an *abelian* group if, in addition to the four traditional group axioms, G satisfies the condition of commutitiviy. More formally: for some group G with group operation $\cdot$, we say G is an abelian group iff $x \cdot y = y \cdot x$ $\forall x, y \in G$.

A *morphism* is the somewhat general notion of 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 group operations $*$ and $\cdot$ respectively, a *group homomorphism* is a mapping $h : G \to H$ 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 $h : G \to H$, 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.

2.1.1 Elliptic Curves

geometry stuff: An elliptic curve is an algebraic curve defineable by an equation of the form $y^2 = x^3 + ax + b$. 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.

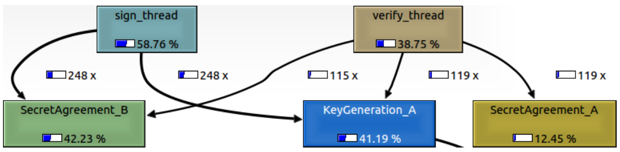

Figure 2.1: Temporary. To be replaced with illustration of EC group operation.

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

2.1.2 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.

2.1.3 Isogenies & Their Properties

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

Definition 2 (Isogeny)**.** Let G and H be abelian varieties[ref]. An isogeny is a homomorphism[ref] $h : G \to H$ possessing a finite kernel.

2.2 Supersingular Isogeny Diffie-Hellman

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

SIDH protocols, as the name suggests, work over supersingular curves of smooth order. Let $\mathbb{F}_q = \mathbb{F}_{p^2}$ be the finite field over which our curve is defined. 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.

2.2.1 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

Location	Efficient Algo's Appendix A
Input	$x_{P_B}, x_{P_A}, y_{P_A}$, $SK_{Alice} = m_A \cdot l_A$
Output	$PK_{Alice} = [x_{\Phi_A}(P_B), x_{\Phi_A}(Q_B), x_{\Phi_A}(Q_B - P_B)]$

EphemeralKeyGeneration_A

Location	kex.c
Input	unsigned char* PrivateKeyA, unsigned char* PublicKeyA, PCurveIsogenyStruct CurveIsogen invBatch* batch
Output	publickey_t PublicKeyA, digit_t PrivateKeyA

Ephemeral Key Generation – Bob:

Key generation for Bob

Location	Efficient Algo's Appendix A
Input	$x_{P_A}, x_{P_B}, y_{P_B}$, $SK_{Bob} = m_B \cdot l_B$
Output	$PK_{Bob} = [x_{\Phi_B}(P_A), x_{\Phi_B}(Q_A), x_{\Phi_B}(Q_A - P_A)]$

EphemeralKeyGeneration_B

Location	kex.c
Input	unsigned char* PrivateKeyB, unsigned char* PublicKeyB, PCurveIsogenyStruct CurveIsogen invBatch* batch
Output	publickey_t PublicKeyB, digit_t PrivateKeyB

Ephemeral Secret Agreement – Alice:

Shared secret algorithm for Alice

Location	Efficient Algo's Appendix A
Input	$PK_{Bob} = [x_{\Phi_B}(P_A), x_{\Phi_B}(Q_A), x_{\Phi_B}(Q_A - P_A)]$ $SK_{Alice} = m_A \cdot l_A$
Output	A shared secret j-invariant of an elliptic curve

EphemeralSecretAgreement_A

Location	kex.c
Input	const unsigned char* PrivateKeyA const unsigned char* PublicKeyB unsigned char* SharedSecretA, PCurveIsogenyStruct CurveIsogen invBatch* batch
Output	f2elm_t SharedSecretA,

Ephemeral Secret Agreement – Bob:

Shared secret algorithm for Bob

Location	Efficient Algo's Appendix A
Input	$PK_{Alice} = [x_{\Phi_A}(P_B), x_{\Phi_A}(Q_B), x_{\Phi_A}(Q_B - P_B)]$ $SK_{Bob} = m_B \cdot l_B$
Output	A shared secret j-invariant of an elliptic curve

EphemeralSecretAgreement_B

Location	kex.c
Input	const unsigned char* PrivateKeyB const unsigned char* PublicKeyA unsigned char* SharedSecretB, PCurveIsogenyStruct CurveIsogen invBatch* batch
Output	f2elm_t SharedSecretB,

2.2.2 Security Assumptions

2.2.3 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 signature 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 $\phi : E \to E/\langle S \rangle$. Return (pk, sk) where pk = $(E/\langle S \rangle, \phi(P_B), \phi(Q_B))$ and sk = S.
Sign():
Verify():

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

Algorithm 1 KeyGen(λ)

Pick a random point S of order $\ell_A^{e_A}$
Compute the isogeny $\phi : E \to E/\langle S \rangle$
pk $\leftarrow (E/\langle S \rangle, \phi(P_B), \phi(Q_B))$
sk $\leftarrow S$
return (pk,sk)

Algorithm 2 Sign(sk, m)

for `i = 1..2`λ **do**
  Pick a random point R of order $\ell_B^{e_B}$
  Compute the isogeny $\psi : E \to E/\langle R \rangle$
  Compute either $\phi' : E/\langle R \rangle \to E/\langle R, S \rangle$ or $\psi' : E/\langle S \rangle \to E/\langle R, S \rangle$
  $(E_1, E_2) \leftarrow (E/\langle R \rangle, E/\langle R, S \rangle)$
  $\mathtt{com}_i \leftarrow (E_1, E_2)$
  $\mathtt{ch}_{i,0} \leftarrow_R \{0, 1\}$
  $(\mathtt{resp}_{i,0}, \mathtt{resp}_{i,1}) \leftarrow ((R, \phi(R)), \psi(S))$
  if $\mathtt{ch}_{i,0} = 1$ **then**
    $\mathtt{swap}(\mathtt{resp}_{i,0}, \mathtt{resp}_{i,1})$
  $h_{i,j} \leftarrow G(\mathtt{resp}_{i,j})$
$J_1 \parallel ... \parallel J_{2\lambda} \leftarrow H(pk, m, (\mathtt{com}_i)_i, (\mathtt{ch}_{i,j})_{i,j}, (h_{i,j})_{i,j})$
return $\sigma \leftarrow ((\mathtt{com}_i)_i, (\mathtt{ch}_{i,j})_{i,j}, (h_{i,j})_{i,j}, (\mathtt{resp}_{i,J_i})_i)$

Algorithm 3 Verify(pk, m, σ)

$J_1 \parallel ... \parallel J_{2\lambda} \leftarrow H(pk, m, (\mathtt{com}_i)_i, (\mathtt{ch}_{i,j})_{i,j}, (h_{i,j})_{i,j})$
for `i = 0..2`λ **do**
  check $h_{i,J_i} = G(\mathtt{resp}_{i,J_i})$
  if $\mathtt{ch}_{i,J_i} = 0$ **then**
    Parse $(R, \phi(R)) \leftarrow \mathtt{resp}_{i,J_i}$
    check $(R, \phi(R))$ have order $\ell_B^{e_B}$
    check R generates the kernel of the isogeny $E \to E_1$
    check $\phi(R)$ generates the kernel of the isogeny $E/\langle S \rangle \to E_2$
  else
    Parse $\psi(S) \leftarrow \mathtt{resp}_{i,J_i}$
    check $\psi(S)$ has order $\ell_A^{e_A}$
    check $\psi(S)$ generates the kernel of the isogeny $E_1 \to E_2$
if all checks succeed **then**
  return 1

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

Algorithm 4 KeyGen(λ)

(pk, sk) $\leftarrow$ `KeyGeneration_B()`
return (pk,sk)

Algorithm 5 Sign(sk, m)

for $\texttt{i = 1..2}\lambda$ **do**
    $(, R, \psi) \leftarrow \texttt{KeyGeneration\_A(E)}$
    $E_1 \leftarrow E/\langle R \rangle$
    $(E_2, E/\langle R, S \rangle) \leftarrow \texttt{SecretAgreement\_B()}$
    $(E_1, E_2) \leftarrow (E/\langle R \rangle, E/\langle R, S \rangle)$
    $\texttt{com}[i] \leftarrow (E_1, E_2)$
    $\texttt{ch}[i] \leftarrow_R \{0, 1\}$
    $(\texttt{resp}[i]_0, \texttt{resp}[i]_1) \leftarrow ((R, \phi(R)), \psi(S))$
$J_1 \parallel ... \parallel J_{2\lambda} \leftarrow H(pk, m, (\texttt{com}_i)_i, (\texttt{ch}_i)_i, (h_{i,j})_{i,j})$
return $\sigma \leftarrow ((\texttt{com}_i)_i, (\texttt{ch}_{i,j})_{i,j}, (h_{i,j})_{i,j}, ((\texttt{resp})[J_i])$

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 arithmitic 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

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**
        $dest[i][0] \leftarrow a[i] * vec[i][0]$
        $vec[i][1] \leftarrow -1 * vec[i][1]$
        $dest[i][1] \leftarrow a[i] * vec[i][1]$

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

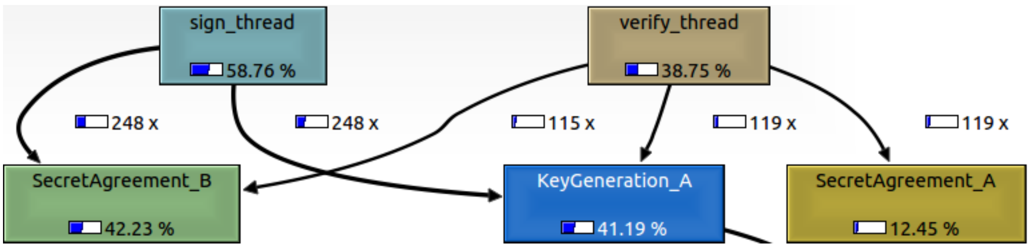

Figure 3.1: ¡Caption here¿

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 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 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] ¡Name of the reference here¿, <urlhere>

[2] ¡Name of the reference here¿, <urlhere>