Molly Fryatt

Optimising Elliptic Curve Cryptography over Binary Finite Fields in Julia

Computer Science Tripos – Part II

St John's College

March 24, 2021

Proforma

Name:	**Molly Katherine Fryatt**
College:	**St John's College**
Project Title:	**Optimising Elliptic Curve Cryptography over Binary Finite Fields in Julia**
Examination:	**Computer Science Tripos – Part II, July 2021**
Word Count:	**TODO**[1]
Project Originator:	Dr Markus Kuhn
Supervisor:	Dr Markus Kuhn

Original Aims of the Project

Work Completed

All that has been completed appears in this dissertation.

Special Difficulties

None.

[1]This word count was computed by `detex diss.tex | tr -cd '0-9A-Za-z \n' | wc -w`

Declaration

I, Molly Katherine Fryatt of St John's College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

Signed [signature]

Date [date]

Contents

List of Figures

Chapter 1

Introduction

This dissertation describes the development of an elliptic curve cryptography and binary fields package for Julia [1], a high performance language launched in 2012. My main aims for this package, called BinaryECC, were to consider multiple algorithms for each of the key operations, comparing their performance on varying types and sizes of curve. In this chapter, I discuss the motivation behind elliptic curve cryptography and give an overview of the pre-existing Julia packages relevant to this area.

1.1 Motivation

Many cryptographic protocols, such as Diffie-Hellman key exchange ([2], 1976), depend upon the assumption that is computationally difficult to solve the Discrete Logarithm Problem (DLP). This is the problem of finding x, given $y = g^x \in \mathbb{G}$ (where g and $\mathbb{G}$ are publicly known and fixed within the system). There are several general-purpose algorithms for solving this problem in any group, but they have complexity at least $O(\sqrt{q}\cdot\text{polylog}(q))$. However, there also exists the Index Calculus Algorithm for computing, in subexponential time, discrete logarithms in the cyclic group $\mathbb{Z}_p^*$. This means that to achieve 80 bit bit security (meaning that we assume attackers cannot perform more than 2^{80} operations), the order of a group $\mathbb{Z}_p^*$ (and therefore also the keys used in the various cryptographic schemes), must be around 1024 bits long ("Guide To Elliptic Curve Cryptography", [3]).

An alternative type of group that can be used in place of $\mathbb{Z}_p^*$ are elliptic curve groups, in which the group operation involves "bouncing" a point around a curve. Given an initial point G and a final point P, it is computationally difficult to determine how many such "bounces" were made, giving rise to the Elliptic Curve Discrete Logarithm Problem (ECDLP). Because the Index Calculus algorithm is not applicable to this group and there are no (known) subexponential algorithms for solving the ECDLP, this problem is thought to be computationally harder (Silverman 1998, [4]). This allows an elliptic curve group of order roughly 2^{160} to be used for the same level of security as the group $\mathbb{Z}_p^*$ with $\log_2 p \approx 1024$, resulting in much smaller key sizes.

Elliptic curve groups are formed from an elliptic curve, E, and an underlying field, $\mathbb{K}$, in which the arithmetic for point addition and doubling is performed. In cryptography, we use either a prime field, i.e. $\mathbb{K} = \mathbb{Z}_p$, or a binary field, $\mathbb{K} = \mathbb{F}_{2^n}$. In this project, I focus only on curves defined over binary fields, because there are currently no packages available for Julia with this functionality.

In the rest of this dissertation, I explore the mathematics of elliptic curves and binary fields in more detail, and discuss the various implementation options that are available.

1.2 Related Work

This project tackles the problem of producing a Julia package for binary curves which allows the user a large amount of flexibility, while still maintaining high performance. At the time of writing, there is no other package that I am aware of which offers this complete functionality.

1.2.1 GaloisFields

This package, Kluck 2018 [5], provides support for Galois fields to Julia. It allows users to create elements using either a function or a macro, allowing the user to provide their own generator if desired, and then it offers a range of arithmetic operations that can be applied to the elements. However, this is a package is designed to offer many fields ($GF(p^m)$, for $m \geq 1$ and arbitrary prime p), and so it is not optimised for binary fields in particular.

1.2.2 ECC

This Julia package, Castano 2019 [6], provides functions for custom prime curves and one predefined curve (secp256k1, [7]), to be used for public key cryptography and signatures. This package offers elliptic curve cryptography, but it does not support binary field arithmetic or binary curves.

1.2.3 MIRACL

MIRACL (Multiprecision Integer and Rational Arithmetic Cryptographic Library) is a C software library for ECC ([8]). It provides a large number of features, including support for elliptic curves defined over binary fields.

1.3 Challenges

Undertaking this project presented me with two key challenges: firstly, I had decided to use Julia, a language that was new to me and which only released a stable version in 2018; and secondly, much of the mathematics required to understand elliptic curve cryptography is beyond the scope of material taught in the Tripos. Due to these challenges, and the constraint of this being a one-year project, I decided to focus more on implementing an efficient package than on security aspects, such as resistance to side channel attacks, etc.

1.4 Results

Chapter 2

Preparation

2.1 Background

Elliptic curve cryptography uses elliptic curve groups, which are formed from the set of points in a field that are on the given curve. In cryptography, this field is either a binary finite field, $GF(2^m)$, or a prime field, $GF(p)$, and it is the first of these two which I focus on. In this section, I begin by describing binary fields, and then move on to elliptic curve groups, before outlining cryptographic schemes which make use of these objects.

2.1.1 Binary Fields

A field is a set of elements, $\mathbb{F}$, with two operations, $+$ and $\cdot$, such that $(\mathbb{F}, +)$ is an abelian group with neutral element 0_F, $(\mathbb{F}, \cdot)$ is a commutative monoid with neutral element 1_F, $(\mathbb{F}\backslash\{0_F\}, \cdot)$ is an abelian group with neutral element 1_F and the distributive law holds. A field is finite (or known as a Galois field) if $\mathbb{F}$ is a finite set, and the order of a finite field is the number of elements it contains.

In this project, I use binary finite fields constructed with a polynomial basis representation. A binary field $GF(2^m)$ has order 2^m, and elements are represented by binary polynomials of degree of at most $m-1$, meaning that they can be written in the form:

$$a_{m-1}x^{m-1} + a_{m-2}x^{m-2} + \cdots + a_2x^2 + a_1x + a_0, \text{ where } a_i \in \{0, 1\} \tag{2.1}$$

The neutral elements of this field are $0_F = 0$ and $1_F = 1$. Such fields also have a reduction polynomial, which is an irreducible binary polynomial of degree m and is referred to here as $f(x)$. Although fields of the same order are isomorphic to one another, the performance of binary fields depends on the choice of reduction polynomial. Standards organisations, such as NIST [9] and SECG [7], provide recommended reduction polynomials for commonly used field orders.

Addition of elements, $a + b$, can be performed by simply adding the two polynomials together (where addition of coefficients $a_i + b_i$ is performed modulo 2). For example:

$$\begin{array}{rr}
a_{m-1}x^{m-1} + a_{m-2}x^{m-2} + \cdots + a_2x^2 + a_1x + a_0 & \\
b_{m-1}x^{m-1} + b_{m-2}x^{m-2} + \cdots + b_2x^2 + b_1x + b_0 & + \\
\hline
(a_{m-1}+b_{m-1})x^{m-1} + (a_{m-2}+b_{m-2})x^{m-2} + \cdots + (a_2+b_2)x^2 + (a_1+b_1)x + (a_0+b_0) &
\end{array}$$

Because addition and subtraction are equivalent in the field $\mathbb{F}_2$ (both can be implemented as exclusive-or), they are also equivalent in binary fields.

Similarly, multiplication of binary field elements, $a \cdot b$ is performed as multiplication of the two binary polynomials. However, this may produce a polynomial of degree greater than $m-1$ (at most, it will be degree $2m-2$), which is not an element of $GF(2^m)$. As a result, the product of the polynomials must be reduced modulo $f(x)$ to produce the corresponding field element. Every element of a binary field, except 0, has a multiplicative inverse which can be found using the extended polynomial version of Euclid's algorithm.

2.1.2 Elliptic Curve Groups

Elliptic curves are defined by the Weierstrass equation,

$$E : y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6 \text{ where } a_i \in \mathbb{K} \tag{2.2}$$

An elliptic curve group, with curve E and underlying field $\mathbb{L}$, can then be defined as the set of $\mathbb{L}$-rational points on the curve with an additional point at infinity:

$$E(\mathbb{L}) = \{(x,y) \in \mathbb{L}^2 \mid y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6\} \cup \{\mathcal{O}\} \tag{2.3}$$

The order of an elliptic curve group is therefore one plus the number of points on the curve in the underlying field, and is written as $\#E(\mathbb{K}) = nh$, where n is the largest prime factor and h is its cofactor. The security level of the group defined to be n, and so we prefer to use curves which are cyclic (i.e. $h = 1$) or almost cyclic ($h \in \{2, 4\}$).

For binary curves, $\mathbb{L} = \mathbb{K} = GF(2^m)$. If the curve is non-supersingular (that is, the determinant $\Delta = a_1$ is not zero), we can rewrite the curve equation to be

$$E : y^2 + xy = x^3 + ax^2 + b \tag{2.4}$$

Group Law

Elliptic curve groups are additive, meaning the group operation allows points to be added together and every point has an inverse (its reflection about the x axis). For a binary curve, the inverse of a point $P = (x, y)$ is $-P = (x, x+y)$, and the inverse of $\mathcal{O}$ is $\mathcal{O}$.

General point addition (that is, addition of points $P_1 + P_2$, where $P_1 \neq P_2$ and $P_1 \neq -P_2$ and neither point is $\mathcal{O}$) is performed using the chord rule. Firstly, a line is drawn through P_1 and P_2, which will have three intersections with the curve E. The third intersection our line with E is found, and then the inverse of this point is calculated to produce the result, $P_3 = P_1 + P_2$. If we have that $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$, then their sum is $P_3 = (x_3, y_3)$ where

$$x_3 = \lambda^2 + \lambda + x_1 + a \tag{2.5}$$

$$y_3 = \lambda(x_1 + x_3) + x_3 + y_1 \tag{2.6}$$

$$\lambda = \frac{y_1 + y_2}{x_1 + x_2} \tag{2.7}$$

Point doubling (that is, where $P_1 = P_2$) uses the tangent rule. Firstly, a line is drawn from P_1 with the gradient of the curve E at that point, and then the process is similar to the chord rule: the result is the inverse of the third intersection of our line with E. For a point $P_1 = (x_1, y_1)$, $P_3 = 2 \cdot P_1 = (x_3, y_3)$, where

$$x_3 = \lambda^2 + \lambda + a \tag{2.8}$$

$$y_3 = x_1^2 + \lambda x_3 + x_3 \tag{2.9}$$

$$\lambda = x_1 + \frac{y_1}{x_1} \tag{2.10}$$

To complete the coverage of every possible case we also note that:

- $\mathcal{O} + P_2 = P_2$, since $\mathcal{O}$ is the neutral element.
- $P_1 + \mathcal{O} = P_1$, similarly.
- $P_1 + (-P_1) = \mathcal{O}$, since the line through P_1 and $-P_1$ would be a vertical line, and so would pass through $\mathcal{O}$, the point at infinity.

From these rules, it is clear that the group operation is commutative, and so this is an abelian group.

Scalar multiplication

Scalar multiplication, $P \cdot k$, is the addition of the curve point P to itself, k times. For cryptographic values of k (that is, where $\log_2 k \approx 100$, this naive approach is computationally infeasible. Instead, the binary representation of k can be used with a double-and-add method:

$$P \cdot k = \sum_i 2^i k_i P \text{ where } k_i \in \{0, 1\} \tag{2.11}$$

Because the inverse of a point is cheap to compute (as $-P = (x, x + y)$, requiring just one addition in the underlying field), it can also be useful to represent k in a binary non-adjacent form (NAF), in which $k_i \in \{-1, 0, 1\}$. This representation is computed in a similar way to a binary representation, except that when we set $k_i \neq 0$, we can choose whether $k_i = 1$ or $k_i = -1$ to ensure that the next digit of the representation (k_{i-1}) is zero. Therefore the non-adjacent form of an integer has the property that there are no adjacent nonzero values, producing a more sparse representation of k (the average density of zeros across NAFs of the same length is $\frac{1}{3}$). This allows us calculate $P \cdot k$ with fewer curve point additions. This concept can be generalised to a windowed NAF, where a window size of w allows $k_i \in (-2^{w-1}, 2^{w-1})$.

Projective Coordinates

The standard representation of points in a two-dimensional field $\mathbb{L}^2$ uses affine coordinates, so that a point is represented by two values from $\mathbb{L}$, for example $P = (x, y) \in \mathbb{L}^2$. An alternative is to use a projective coordinate system, parameterised by $c, d \in \mathbb{N}$, in which points are represented by three values from $\mathbb{L}$, written as $P = (X : Y : Z) \in \mathbb{L}^3$. In such a coordinate system, we define an equivalence relation: $P_1 = (X_1, Y_1, Z_1) \sim P_2 = (X_2, Y_2, Z_2)$ if $X_1 = \lambda^c X_2$, $Y_1 = \lambda^d Y_2$ and $Z_1 = \lambda Z_2$ for some $\lambda \in \mathbb{L}^*$. A bijection between affine coordinates and a projective coordinate system is given by the mapping $(x, y) \mapsto (x : y : 1)$, and $(X : Y : Z) \mapsto (X/Z^2, Y/Z^3)$ is the inverse when $Z \neq 0$.

Projective coordinates allow the point doubling and point addition formulae to be rewritten without any inversions in the field $\mathbb{K}$, which, for $\mathbb{K} = GF(2^m)$, can be expensive compared to multiplications. There are two forms of projective coordinates which I explore in this project: Jacobian coordinates and Lopez-Dahab coordinates.

For Jacobian coordinates, we have the parameters $c = 2$ and $d = 3$. The curve equation can be rewritten as

$$E : Y^2 + XYZ = X^3 + aX^2Z^2 + bZ^6 \tag{2.12}$$

meaning that all representatives of equivalence classes which are "on the curve E" will satisfy this new form of E. New formulae for point addition and doubling can be produced by substituting $x = \frac{X}{Z^2}$ and $y = \frac{Y}{Z^3}$ into the versions for affine coordinates. For addition, this is:

$$\frac{X_3}{Z_3^2} = \lambda^2 + \lambda + \frac{X_1}{Z_1^2} + \frac{X_2}{Z_2^2} + a \qquad \frac{Y_3}{Z_3^3} = \lambda\left(\frac{X_1}{Z_1^2} + \frac{X_3}{Z_3^2}\right) + \frac{X_3}{Z_3^2} + \frac{Y_1}{Z_1^3}$$

$$\lambda = \frac{Y_1 Z_2^3 + Y_2 Z_1^3}{X_1 Z_1 Z_2^3 + X_2 Z_1^3 Z_2}$$

These can then be rearranged, allowing formulae for X_3, Y_3 and Z_3 to be extracted as:

$$X_3 = A^2 + AB + C^3 + aB^2) \cdot Z_1^4 \tag{2.13}$$
$$Y_3 = A(X_1 Z_3^2 + X_3 Z_1^2) + CZ_2(X_3 Z_1^3 + Y_1 Z_3^2) \tag{2.14}$$
$$Z_3 = BZ_1^2 \tag{2.15}$$

where we have $\quad A = Y_1 Z_2^3 + Y_2 Z_1^3 \quad C = X_1 Z_2^2 + X_2 Z_1^2 \quad B = Z_1 Z_2 C$

For point doubling, the same process can be followed to produce the formulae for $2 \cdot P_1 = P_3 = (X_3, Y_3, Z_3)$:

$$X_3 = (A^2 + AB + aB^2) \cdot Z_1^8 \tag{2.16}$$
$$Y_3 = BX_1^2 Z_3^2 + (A + B)X_3 Z_1^4 \tag{2.17}$$
$$Z_3 = BZ_1^4 \tag{2.18}$$

where we have $\quad A = X_1^2 + Y_1 Z_1 \quad B = X_1 Z_1^2$

For Lopez-Dahab coordinates, the parameters are $c = 1$ and $d = 2$. The curve equation is

$$E : Y^2 + XYZ = X^3Z + aX^2Z^2 + bZ^4 \tag{2.19}$$

and the formulae for point addition (derived through the same method as above) are:

$$X_3 = A^2 + AB + BC^2 + aB^2 \tag{2.20}$$
$$Y_3 = ACZ_2(X_1Z_3 + X_3Z_1) + C^2Z_2^2(X_3Z_1^2 + Y_1Z_3) \tag{2.21}$$
$$Z_3 = B^2 \tag{2.22}$$

where $A = Y_1Z_2^2 + Y_2Z_1^2$ $\quad C = X_1Z_2 + X_2Z_1$ $\quad B = Z_1Z_2C$

The formulae for point doubling are:

$$X_3 = A(A + B) + aB^2 \tag{2.23}$$
$$Y_3 = X_1^4Z_3 + X_3B(A + B) \tag{2.24}$$
$$Z_3 = B^2 \tag{2.25}$$

where $A = X_1^2 + Y_1$ $\quad B = Z_1X_1$

2.1.3 Elliptic Curve Cryptography

In elliptic curve groups, we have the Elliptic Curve Discrete Logarithm Problem (ECDLP): given $P = G \cdot k$ (where the group $E(\mathbb{F}_q)$ and the generating point G are known), find k. There are currently no known algorithms to solve this problem in subexponential time, making it a suitable group for many cryptographic schemes.

In such schemes, details of the group and generating point are fixed and publicly known. For binary curves, these details, known as Elliptic Curve Domain Parameters, are stored in a septuple (SEC 1, 2009, [10]): $T = (m, f(x), a, b, G, n, h)$.

Element	**Purpose**
m	$\log_2$ of the underlying field order, i.e. $GF(2^m)$
$f(x)$	Irreducible polynomial of degree m, specifying the field $GF(2^m)$
a, b	Parameters for the curve equation $E : y^2 + xy = x^3 + ax^2 + b$
G	Generating point for a large prime-order subgroup in the curve
n	Prime order of point G
h	Cofactor of n, i.e. $\#h = E(GF(2^m))/n$

Another important primitive are Elliptic Curve Key Pairs, which are a tuple (d, Q) associated with a septuple T of curve domain parameters [10]. To generate a key pair, an integer $d \in \mathbb{Z}_n^*$ is chosen uniformly at random, and then the point $Q = G \cdot d$, where G is the generating point from T. In this key pair, d is known as the secret key and Q is the public key.

ECDSA

One scheme which uses these primitives is a the Elliptic Curve Digital Signature Algorithm (ECDSA), a variant of DSA, which allows users to sign messages and verify signatures produced by other users. In this scheme, each entity is assumed to have an elliptic curve key pair, where the public component is publicly known and trusted. So long as an attacker has not captured the corresponding private key, it is computationally infeasible for them to produce a “valid” signature, meaning the scheme provides existential unforgeability [10].

Given a message M, entity U can use their key pair (Q_U, d_U) to produce a signature $(r, s) \in \mathbb{Z}^2$, using the following algorithm [10]:

Algorithm 1: ECDSA Signing Operation

```
   Data: Message $M$ to be signed, key pair $(d_U, Q_U)$, parameters $T$
   Result: Signature $S = (r, s)$ on $M$, or “invalid”
1  Select ephemeral key pair $(k, R)$ associated with $T$;
2  $\overline{x_R} := \mathrm{int}(x_R)$;
3  $r := \overline{x_R} \mod n$;
4  if $r = 0$ then
5      Return to step 1;
6  end
7  $H := \mathrm{hash}(M)$;
8  $e := \mathrm{int}(H)$;
9  $s := k^{-1}(e + r d_U) \mod n$;
10 if $s = 0$ then
11     Return to step 1;
12 end
13 Output $S = (r, s)$;
```

Any other entity who knows U’s public key, Q_U, can verify the signature $S = (r, s)$

for message M, with the following algorithm [10]:

Algorithm 2: ECDSA Verifying Operation

```
Data: Message $M$ to be verified, public key $Q_U$, parameters $T$
Result: "valid" or "invalid"
if $r \notin [1, n-1]$ or $s \notin [1, n-1]$ then
    Output "invalid" and stop;
end
$H :=$ hash$(M)$;
$e :=$ int$(H)$;
$u_1 := es^{-1} \mod n$;
$u_2 := rs^{-1} \mod n$;
$R := u_1 G + U_2 Q_U$;
if $R = \mathcal{O}$ then
    Output "invalid" and stop;
end
$\overline{x_R} :=$ int$(x_R)$;
$v := \overline{x_R} \mod n$;
Output $S = (r, s)$;
if $r = v$ then
    Output "valid" and stop;
else
    Output "invalid" and stop;
end
```

ECDH

The Elliptic Curve Diffie-Hellman (ECDH) scheme allows two entities to agree on a symmetric key over an authenticated channel without any eavesdroppers also being able to derive the key. The security of this scheme relies on the computational complexity of a problem related to ECDLP, known as the Elliptic Curve Diffie-Hellman Problem (ECDHP): given $Q_1 = d_1G$ and $Q_2 = d_2G$, determine d_1d_2G ([10], B.2.3).

In the setup for the scheme, the entities A and B agree on a set of curve domain parameters, T. From the perspective of A, they generate an ephemeral key pair (d_A, Q_A) and send Q_A to be B over the authenticated channel. A then receives Q_B from B, and they multiply it to find $Q = Q_B \cdot d_A = G \cdot d_B \cdot d_A$. B follows a similar process to also obtain the shared point Q, but any eavesdropper would have to solve ECDHP to derive that same point.

2.2 Requirements analysis

The main components of the project have been listed, along with their priority, expected difficulty, and estimated impact on the project of not implementing them well.

- **Binary fields**
  priority: high, *difficulty: high*, *risk: high*
  - This component is very high priority and risk, because the arithmetic for curve point manipulations occur in binary fields. As a result, the difficulty is also high, because much of the work in to improve performance will need to occur at this level.
- **Prime fields**
  priority: low, *difficulty: medium*, *risk: low*
  - Prime field arithmetic is needed for the examples of cryptographic algorithms, such as ECDSA, but nowhere else. Therefore it has a much smaller impact on the project as a whole, and does not need to be optimised as much or implemented as urgently.
- **Elliptic curve groups (affine representation)**
  priority: high, *difficulty: high*, *risk: high*
  - This component is central to the project as it provides the standard version of elliptic curve arithmetic that will be used throughout other components. The difficulty and risk are also high because, as a major component, it needs to provide an efficient implementation of elliptic curves in order for the rest of the system to work well.
  - For this component, I will use test vectors [11] to verify that the implementation is correct (because cryptographic-sized elliptic curve groups are too large for their arithmetic to be verified by hand).
- **Elliptic curve groups (Jacobian representation)**
  priority: medium, *difficulty: high*, *risk: medium*
  - This module is lower priority and risk than the version with affine coordinates because it will not have any other modules depend on it, as it is simply an alternative representation of curve points that can be switched in.
  - I will test this implementation using the same test vectors as with the affine representation, but wrapped in conversion routines as the values provided are all in affine coordinates.
- **Elliptic curve groups (Lopez-Dahab representation)**
  priority: medium, *difficulty: high*, *risk: medium*
  - Same reasoning and tools as the Jacobian representation.
- **Standard binary fields and curve domain parameters**
  priority: medium, *difficulty: low*, *risk: medium*

  - This involves finding standard fields and curve domain parameters, and a suitable way to present them to users, which will be lower difficulty than other components. However, it is still medium priority because these standard values will be important for testing correctness, and therefore it also has a medium risk.
  - For this component, I will use the standards document SEC 2 (version 2.0, 2010, [12]) from the Standards for Efficient Cryptography Group (SECG, [7]), and the equivalent document [13] from NIST for additional detail or information where needed. However, for consistency in naming and format, I will only use curves and fields from SEC 2.

- **Cryptographic primitives**
  priority: low, *difficulty: medium*, *risk: low*
  - Implementing examples of ECC based schemes is low priority because it does not have any dependant components, and will not impact the rest of the system if implemented poorly.
  - For this component, I will use standards document SEC 1 [10] from SECG, which lists and describes in detail all of the cryptographic primitives that I will need to implement.

2.3 Methods and Tools

This section describes the methods and tools used in the development of the project.

2.3.1 Software engineering principles

Due to the structure of the project, it is necessary to have working prototype of each module as early as possible. This initial prototype is developed with a waterfall methodology, as the progress at this stage will be fairly linear and the basic requirements for each component are already known.

After this, the development will switch to an iterative methodology, in which each cycle consists of planning, implementing, testing and analysing new features. This flexible approach is necessary as it is not possible to know (before development begins) how best to refine components or which routines consume the most runtime. During the development process, unit tests for field and curve arithmetic will be used, allowing implementation errors to be discovered and fixed as early as possible. Abstractions will be used to allow each component to be re-implemented without its interface with other components needing to be changed, allowing alterations to be made more easily, or for the performance of alternative implementations to be compared.

2.3.2 Choice of Tools

This dissertation was written using LaTeX, using the Texmaker editor [14]. BinaryECC was written in Julia, using Juno [15], an Integrated Development Environment, and it depends on two Julia packages:

1. StaticArrays [16]: to provide statically sized arrays, something which is not a built in feature of Julia, to enable certain performance improvements
2. SHA [17]: to provide a hashing function, for the implementation of cryptographic protocols

Julia was also used for peripheral activities such as data cleanup, testing, benchmarking, and for producing graphs and documentation. For these purposes, I used several other packages: Test, for creating automated unit tests; BenchmarkTools [18], to analyse the performance (runtime and memory allocations) of different algorithms; Plots [19] and LaTeXStrings [20], to produce graphs visualising performance results using PGFPlots as a backend; and Documenter [21], to produce documentation hosted on GitHub.

Both the development of the Julia package and the writing of this dissertation were carried out on my personal machine, with Git for revision control and backups stored on my University OneDrive and on GitHub.

2.4 Summary

Chapter 3

Implementation

3.1 Structure

3.1.1 Repository overview

3.1.2 Types

3.2 Binary Fields

3.2.1 Representation

3.2.2 Reduction

3.2.3 Multiplication

3.2.4 Inversion

3.3 Elliptic Curve Groups

3.3.1 Representation

3.3.2 Point addition and doubling

3.3.3 Scalar multiplication

3.4 Cryptographic primitives

3.4.1 Prime fields

3.4.2 Key pairs

3.4.3 ECDSA

3.4.4 ECDH

3.5 Testing

3.6 Limitations

Chapter 4

Evaluation

4.1 Performance

4.1.1 Comparison with Julia

4.1.2 Comparison with C

4.2 Requirements met

4.3 Limitations

Chapter 5

Conclusion

Bibliography

[1] J. Bezanson, S. Karpinski, V. B. Shah, *et al.*, “The Julia programming language.” Available: `https://julialang.org/`, 2020.

[2] W. Diffie and M. Hellman, “New directions in cryptography,” *IEEE Trans. Inf. Theor.*, vol. 22, p. 644–654, Sept. 2006.

[3] D. Hankerson, A. Menezes, and S. Vanstone, *Guide to Elliptic Curve Cryptography*. Springer-Verlag New York, 2004.

[4] J. H. Silverman and J. Suzuki, “Elliptic curve discrete logarithms and the index calculus,” in *Advances in Cryptology — ASIACRYPT’98* (K. Ohta and D. Pei, eds.), (Berlin, Heidelberg), pp. 110–125, Springer Berlin Heidelberg, 1998.

[5] T. Kluck, “GaloisFields.jl.” Available: `https://github.com/tkluck/GaloisFields.jl`, 2018.

[6] S. Castano, “ECC.jl.” Available: `https://github.com/roshii/ECC.jl`, 2019.

[7] SECG, “Standards for Efficient Cryptography Group.” Available: `https://www.secg.org/`.

[8] MIRACL, “MIRACL.” Available: `https://github.com/miracl/MIRACL`, 2021.

[9] U.S. Department of Commerce, “National Institute of Standards and Technology.” Available: `https://www.nist.gov/`.

[10] D. R. L. Brown, “Sec 1: Elliptic curve cryptography,” 2009.

[11] B. Poettering, “Test vectors for the NIST elliptic curves.” Available: `http://point-at-infinity.org/ecc/nisttv`, 2007.

[12] D. R. L. Brown, “Sec 2: Recommended elliptic curve domain parameters,” 2010.

[13] L. Chen, D. Moody, A. Regenscheid, and K. Randall, “Recommendations for discrete logarithm-based cryptography: Elliptic curve domain parameters,” 2019.

[14] P. Brachet, “Texmaker.” Available: `https://www.xm1math.net/texmaker/`, 2003.

[15] J. Bezanson, S. Karpinski, V. B. Shah, *et al.*, “Juno IDE.” Available: `https://junolab.org/`, 2003.

[16] A. Ferris, "StaticArrays.jl." Available: `https://github.com/JuliaArrays/StaticArrays.jl`, 2016.

[17] J. Bezanson, S. Karpinski, V. B. Shah, *et al.*, "SHA.jl." Available: `https://docs.julialang.org/en/v1/stdlib/SHA/`, 2020.

[18] J. Revels, "BenchmarkTools.jl." Available: `https://github.com/JuliaCI/BenchmarkTools.jl`, 2015.

[19] T. Breloff, "Plots.jl." Available: `https://github.com/JuliaPlots/Plots.jl`, 2015.

[20] S. G. Johnson, "LaTeXStrings.jl." Available: `https://github.com/stevengj/LaTeXStrings.jl`, 2014.

[21] M. Hatherly, "Documenter.jl." Available: `https://github.com/JuliaDocs/Documenter.jl`, 2016.

Appendix A

Project Proposal