Molly Fryatt

Optimising Elliptic Curve Cryptography over Binary Finite Fields in Julia

Computer Science Tripos – Part II

St John's College

April 6, 2021

Declaration of originality

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. I am content for my dissertation to be made available to the students and staff of the University.

Signed Molly Katherine Fryatt

Date [date]

Proforma

Candidate Number:	**TODO**
Project Title:	**Optimising Elliptic Curve Cryptography over Binary Finite Fields in Julia**
Examination:	**Computer Science Tripos – Part II, July 2021**
Dissertation Word Count:	**8,036**[1]
Software Line Count:	**TODO**
Project Originator:	Dr Markus Kuhn
Project Supervisor:	Dr Markus Kuhn

Original Aims of the Project

Work Completed

Special Difficulties

None.

[1] This word count was computed by `texcount`

Contents

List of Figures

Chapter 1

Introduction

This dissertation describes the development of an elliptic curve cryptography and binary fields package for Julia [**?**], 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 ([**?**], 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 all have exponential time complexity. 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", [**?**]).

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, [**?**]). 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 [**?**], 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 Nemo

1.2.3 ECC

This Julia package, Castano 2019 [**?**], provides functions for custom prime curves and one predefined curve (secp256k1, [**?**]), 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.

1.2.4 MIRACL

MIRACL (Multiprecision Integer and Rational Arithmetic Cryptographic Library) is a C software library for ECC ([**?**]). 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
- The distributive law holds

A finite field $\mathbb{F}$ is one which has a finite number of elements. Its order, written $\#\mathbb{F}$, is the number of elements it contains. Galois fields are a type of finite field that have prime or prime power order, written as $GF(p^m)$ for $m \geq 1$ and p prime.

In this project, I use binary Galois fields constructed with a polynomial basis representation. A binary field $GF(2^m)$ has order 2^m, and its 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, written as $f(x)$ here.

Addition and subtraction

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:

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

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.

Multiplication

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.

Shift and add This is the most straightforward method of polynomial multiplication. One polynomial is repeatedly multiplied by x (typically implemented as a left shift) and added to an accumulating result, as can be seen by this sum:

$$a(x) \cdot b(x) = \sum_{i=0}^{m-1} a(x) \cdot b_i x^i \tag{2.2}$$

Comb method For this method, we assume that the field elements have been stored as array of words, where the word length is W and the array length is t. We then rely on the assumption that for this representation, multiplying a polynomial $a(x)$ by x^W is fast: you simply append a zero word. Therefore it is cheaper to calculate $a(x)x^i$ once and then add zero words to produce each $a(x)x^{Wj+i}$ that is needed, than it is to calculate each $a(x)x^{Wj+i}$ from scratch.

$$a(x) \cdot b(x) = \sum_{i=0}^{W-1} \sum_{j=0}^{t-1} a(z) \cdot b_{Wj+i} x^{Wj+i} \tag{2.3}$$

Squaring For binary polynomials, squaring is a linear operation and can therefore be computed with a different, faster, technique. The square of the element $a_i x_i$ is $a_i x^{2i}$, and so the square of a general element $a(x)$ is:

$$a(x)^2 = \sum_{i=0}^{m-1} a_i x^{2i} \tag{2.4}$$

Windowing For each of the methods listed above, the technique of windowing can be applied, yielding performance gains at a cost of precomputing and storing 2^w extra elements (for a window size of w). For example, to multiply $a(x) \cdot b(x)$ we would precompute $c_i(x) = b(x) \cdot x^i$ for $i \in 0 \ldots 2^w - 1$, and then use them as follows, where $\{b_i \ldots b_{i+n}\}$ is the number $\sum_{j=0}^{n} 2^j b_{i+j}$:

$$a(x) \cdot b(x) = \sum_{i=0}^{\lfloor \frac{m-1}{w} \rfloor} a(x) \cdot c_{\{b_{wi} \ldots b_{w(i+1)-1}\}}(x) \cdot x^{wi} \tag{2.5}$$

Reduction

To convert an arbitrary binary polynomial into an element of the binary field $GF(2^m)$, we need to reduce it modulo $f(x)$, where $f(x)$ is the reduction polynomial for that field. Although fields of the same order are isomorphic to one another, the performance of binary fields depends on the choice of $f(x)$. Standards organisations, such as NIST [**?**] and SECG [**?**], provide recommended reduction polynomials for commonly used field orders.

Because a reduction polynomial for $GF(2^m)$ has degree m, we can rewrite it as $f(x) = x^m + f'(x)$, where $f'(x)$ has degree $m-1$ or less, and note that $x^m \equiv f'(x) \pmod{f(x)}$. Now, to reduce $a(x) = x^{m+n} + a'(x)$ (where $a'(x)$ has order strictly less than $m+n$), we can note the following congruences:

$$\begin{aligned} & x^{m+n} + a'(x) && \pmod{f(x)} \\ \equiv\ & x^{m+n} + a'(x) + f(x) \cdot x^n && \pmod{f(x)} \\ \equiv\ & x^{m+n} + a'(x) + x^{m+n} + f'(x) \cdot x^n && \pmod{f(x)} \\ \equiv\ & a'(x) + f'(x) \cdot x^n && \pmod{f(x)} \end{aligned}$$

Since both $a'(x)$ and $f'(x) \cdot x^n$ have degree less than $m+n$, we can conclude that $a'(x) + f'(x) \cdot x^n$ has degree less than $m+n$, and so adding $f(x) \cdot x^n$ successfully reduced the degree of the polynomial.

Fast reduction Tri- and pentanomial reduction polynomials, such as those recommended by NIST [**?**], can be used to perform reduction more efficiently. For example, the field $GF(2^{163})$ has the recommended reduction polynomial $f(x) = x^{163} + x^7 + x^6 + x^3 + 1$, which gives us the following congruences modulo $f(x)$:

$$\begin{array}{lllll} x^{Wn} & \equiv x^{Wn-m+7} & +x^{Wn-m+6} & +x^{Wn-m+3} & +x^{Wn-m} \\ x^{Wn+1} & \equiv x^{Wn+1-m+7} & +x^{Wn+1-m+6} & +x^{Wn+1-m+3} & +x^{Wn+1-m} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ x^{W(n+1)-1} & \equiv x^{W(n+1)-1-m+7} & +x^{W(n+1)-1-m+6} & +x^{W(n+1)-1-m+3} & +x^{W(n+1)-1-m} \end{array}$$

Considering the columns, we can reduce the degree of a polynomial by at least W at a time, by adding five polynomials to it which each fit into a word: $\sum_{i=0}^{W-1} a_{Wn+i} x^{Wn+i}$, $\sum_{i=0}^{W-1} a_{Wn+i} x^{Wn-m+7+i}$, $\sum_{i=0}^{W-1} a_{Wn+i} x^{Wn-m+6+i}$, $\sum_{i=0}^{W-1} a_{Wn+i} x^{Wn-m+3+i}$, $\sum_{i=0}^{W-1} a_{Wn+i} x^{Wn-m+i}$.

Inversion

Every element of a binary field, except the additive identity 0, has a multiplicative inverse. This can be found using the polynomial version of Euclid's algorithm:

$$gcd(a(x), b(x)) = \begin{cases} b(x) & \text{if } a(x) = 0 \\ gcd(b(x), a(x)) & \text{if } deg(a(x)) > deg(b(x)) \\ gcd(b(x) - q(x) \cdot a(x), a(x)) & \text{otherwise} \end{cases} \tag{2.6}$$

This algorithm can be extended to find $g(x)$ and $h(x)$ such that $a(x) \cdot g(x) + b(x) \cdot h(x) = gcd(a(x), b(x))$. To find the multiplicative inverse of $a(x)$ modulo $f(x)$, we therefore call the extended Euclid's algorithm with $a(x)$ and $f(x)$, returning $g(x)$ as the result.

2.1.2 Elliptic Curve Groups

Elliptic curves are defined by the Weierstrass equation:

$$E : y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6 \text{ where } a_i \in \mathbb{K} \tag{2.7}$$

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:

$$E(\mathbb{L}) = \{(x, y) \in \mathbb{L}^2 \mid y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6\} \cup \{\mathcal{O}\} \tag{2.8}$$

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

$$E : y^2 + xy = x^3 + ax^2 + b \tag{2.9}$$

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 $\mathcal{O}$.

Addition For two general points P_1 and P_2 (that is, where $P_1 \neq P_2$ and $P_1 \neq -P_2$ and neither point is $\mathcal{O}$), $P_1 + P_2$ is calculated with 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

$$x_3 = \lambda^2 + \lambda + x_1 + a \tag{2.10}$$

$$y_3 = \lambda(x_1 + x_3) + x_3 + y_1 \tag{2.11}$$

$$\lambda = \frac{y_1 + y_2}{x_1 + x_2} \tag{2.12}$$

Doubling To calculate $P_1 + P_1 = 2P_1$, the tangent rule is used. 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

$$x_3 = \lambda^2 + \lambda + a = x_1^2 + \frac{b}{x_1^2} \tag{2.13}$$

$$y_3 = x_1^2 + \lambda x_3 + x_3 \tag{2.14}$$

$$\lambda = x_1 + \frac{y_1}{x_1} \tag{2.15}$$

Other cases 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.

Double and add Similar to the shift and add method of binary field multiplication, we can use the double-and-add method to multiply P by scalar k. This uses values $k_i \in \{0, 1\}$, where k_i is the i^{th} digit of the binary representation of k.

$$P \cdot k = \sum_i 2^i k_i P \text{ where } k_i \in \{0, 1\} \tag{2.16}$$

Binary NAF 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 choose whether $k_i = 1$ or $k_i = -1$ in such a way 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.

$$P \cdot k = \sum_i 2^i k_i P \text{ where } k_i \in \{-1, 0, 1\} \tag{2.17}$$

Width-w NAF The concept of a non-adjacent representation can be generalised to a have a window size of w, where representation of k uses values $k_i \in \{-2^{w-1} + 1, \ldots, 2^{w-1} - 1\}$. This representation is even more sparse, having an average density of non-zero digits of $1/(w + 1)$ [**?**]. However, we also need to precompute and store the result of $n \cdot P$ for $n \in \{2, \ldots, 2^{w-1} - 1\}$.

Montgomery powering ladder The previous methods for scalar multiplication perform different numbers of point doublings and additions for different values of k of the same length; for example, the double and add method will perform $\log_2 k$ doublings and one addition for every bit set to one in the binary representation of k. Information about k is therefore leaked by the time taken for calculate $k \cdot P$, which is undesirable from a security perspective. An alternative approach is Montgomery scalar multiplication (López and Dahab, 1999 [**?**]), which performs iterates over the bits in a binary representation of k, keeping track of two elliptic curve points, P_1 and P_2, and performing exactly one double and one add per iteration to maintain the invariant $P_2 - P_1 = P$. This method performs a fixed number of elliptic curve point operations for all values of k of the same length, and so it less vulnerable to timing attacks.

Affine Montgomery's method This method can be further refined to perform a lower, but still constant, number of calculations in each iteration. Rather than using the standard point doubling and addition routines, it only keeps track of the x coordinates of the points P_1 and P_2, meaning that less field arithmetic needs to be performed. After the main loop, the y coordinate of the result can be calculated (using P, the x coordinates of P_1 and P_2, and the invariant $P_2 - P_1 = P$).

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.

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

$$E : Y^2 + XYZ = X^3 + aX^2Z^2 + bZ^6 \tag{2.18}$$

meaning that all representatives of equivalence classes which are "on the curve E" will satisfy this new form of E.

López-Dahab coordinates This coordinate system uses the parameters $c = 1$ and $d = 2$, producing the following new form of the curve equation.

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

2.1.3 Prime Fields

Several cryptographic primitives, such as ECDSA, perform some arithmetic in prime fields. For the field Z_p, the elements are the numbers $\{0, 1, \ldots, p-1\}$, and the operations $+$ and $\cdot$ are addition modulo p and multiplication modulo p respectively. The additive inverse of a point x is $p - x$, and the multiplicative inverse can be found with the extended version of Euclid's algorithm.

2.1.4 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, [?]): $T = (m, f(x), a, b, G, n, h)$.

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 [?]. 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 [?].

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 ECDSA Signing Operation ([?], Appendix A). Any other entity who knows U's public key, Q_U, can verify the signature $S = (r, s)$ for message M, with the ECDSA Verifying Operation ([?], Appendix A).

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 ([**?**], 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 [**?**] 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 (López-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, [**?**]) from the Standards for Efficient Cryptography Group (SECG, [**?**]), and the equivalent document [**?**] 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 [**?**] 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.

Implementation Pipeline

2.3.2 Choice of Tools

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

1. StaticArrays [**?**]: to provide statically sized arrays, something which is not a built in feature of Julia, to enable certain performance improvements

2. SHA [**?**]: 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 [**?**], to analyse the performance (runtime and memory allocations) of different algorithms; Plots [**?**] and LaTeXStrings [**?**], to produce graphs visualising performance results using PGFPlots as a backend; and Documenter [**?**], 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

In this chapter, I describe and explain the process of creating a package for elliptic curve cryptography in Julia. First, I outline structure of the package's repository, and then I turn to the development process and the implementation decisions that were made.

3.1 Repository Overview

- BinaryECC
  - benchmarking
  - docs
  - src
    - Cryptography
      - `Crypto.jl`
      - `CurveDomainParams.jl`
    - EllipticCurves
      - `EC.jl`
      - `ECAffine.jl`
      - `ECJacobian.jl`
      - `ECLD.jl`
      - `ECMix.jl`
    - GaloisFields
      - `BField_fastreduce32.jl`
      - `BField_fastreduce64.jl`
      - `BField.jl`
      - `PField.jl`
      - `StaticUInt.jl`
    - `BinaryECC.jl`
    - `tests.jl`
  - testvectors
  - `Manifest.toml`

- `Project.toml`
- `README.md`

3.2 Binary Fields

3.2.1 Representation

Given an element of a binary field, arithmetic routines such as multiplication need to know both the value of the element (which is, in the case of a binary field, a binary polynomial) and the field that the element is in. Julia divides its types into primitives (booleans, characters, floating points and integers) and composites, which represent a collection of names fields [**?**], and it is this second option which I will use to create the types for BinaryECC.

One approach would be to create two separate types - one for fields and one for field elements - and then pass an instance of each to the arithmetic routines, so that $a \cdot b$ would become `*(a, b, field)`. A benefit of this idea is that there is no redundancy: the field information need only be provided once for multiple elements of the same field. However, this is at the cost of both usability, because programmers must remember to create and handle an additional object for field information, and also readability, because the implementation has converted an infix mathematical operation into a prefix one.

An alternative is for field information to be encapsulated within each instance of a field element, allowing the more intuitive notation `a * b` to be used. In my project proposal, one of the success criteria was for the package to be able to parse expressions with high level syntax close to that used in mathematics, and so I chose this latter representation.

Another aspect to consider is how how binary polynomials will be stored. The most straightforward approach here is to store them in integers, where the i^{th} bit of that integer's binary representation is the coefficient a_i. This representation is both space efficient and simplifies the implementations arithmetic operations.

Field information

To uniquely identify a binary field $GF(2^m)$, we need to know both $\#GF(2^m) = 2^m$, the field order, and $f(x)$, an irreducible polynomial of degree m. One approach would be to store only the reduction polynomial (where its degree would provide m, the logarithm of the field order), while an alternative is to store m and only the lower terms of the reduction polynomial (that is, $f'(x)$ where $f(x) = x^m + f'(x)$). Reduction polynomials of fields used for cryptographic purposes, such as $f(x) = x^{163} + x^7 + x^6 + x^3 + 1$ for $GF(2^{163})$ [**?**], tend to have a small number of low terms, and so the second approach makes more efficient use of space (the $GF(2^{163})$ given will use just 16 bits: 8 bits to store each of m and $f'(x)$).

This information could be stored as a field within a binary field point, which would allow the reduction polynomial, stored in an integer, to be arbitrarily large. This could be achieved using Julia's built-in `BigInt` type or by building my own integer type.

However, I instead created a parametric composite type for the field elements, `BFieldPoint{D,R}`, where `BFieldPoint` is its name and integer parameters `D,R` identify the field $GF(2^m)$. The parameter `R` holds the lower terms of the reduction polynomial, $f'(x)$, and `D`

holds its degree, m. This means that elements of different fields will have different types, allowing fields to be checked at compile time, rather than having to check at runtime and possibly throw an exception.

One potential problem is that Julia will compile specialised functions for each different `BFieldPoint{D,R}`, impacting the compile time. Even worse, if a new field is created during the running of a program, the first time a function is called with that new type, a specialised version of that function will also need to be compiled, damaging runtime performance. However, this situation is unlikely to occur because the choice of field is usually part of the core design of a cryptographic system, often based on hardware or processing constraints. As a result, the compiler will only need to create one specialised version of each function and new fields will not be encountered at runtime, minimising the impact of using a parametric type. A second problem is that only "bits" types (that is, numbers, booleans, etc., or structs without pointers to other objects) can be used as parameter values, meaning we are restricted to a `UInt128` to store $f'(x)$, meaning it must be of order less than 128. However, as noted above, reduction polynomials recommended for cryptographic use tend to have a low degree $f'(x)$, and so this should also not pose a problem.

The standard binary fields offered by BinaryECC were therefore provided as specialised field point types, such as `BFieldPoint163 = BFieldPoint{163, 2^7+2^6+2^3+1}`.

Point information

The type `BFieldPoint{D,R}` contains one field which is an integer (approximately 100s of bits long), representing a binary polynomial in the field specified by `{D,R}`. The initial version of this type used objects of the pre-existing `BigInt` type, allowing a working prototype to be developed quickly. However, this was not suitable for the final version because `BigInt` objects can change size during the execution of a program, and this uncertainty meant that the compiler was unable generate high-performance code. The alternative was to store binary polynomials in arrays of fixed-size integers, and so I created a new type called `StaticUInt`.

StaticUInt

The purpose of creating a new type for statically-sized integers was to place a layer of abstraction between the unsigned integers representing binary polynomials, and the binary field arithmetic manipulating those polynomials. This allowed me to iterate several different implementations of the `StaticUInt` type and easily compare them against each other, without the need to alter the binary field arithmetic routines themselves.

Like the field type, `StaticUInt` is also a parametric type, this time with parameters `{L,T<:Unsigned}` representing the length (in words) of the array with `L`, and the type of the words with `T`, which must be an unsigned integer (i.e., it must subtype `Unsigned`, an abstract Julia type). In the current implementation, the package selects either `UInt32` or `UInt64` for the word type, depending on whether it was compiled on a 32 or 64 bit system. This parameterisation of the word type also opens up opportunities in future development by allowing, for example, an unsigned integer type with logging capabilities to be used to explore side channel attacks. Providing the array length as a type parameter also allows the package to take

advantage of type dispatch by providing different implementations of `StaticUInt` operations depending on the lengths of the given objects.

As before when using values as parameters, there is a risk of running into a new version of `StaticUInt{L,T}` at runtime and having to compile functions anew. Fortunately, this risk is mitigated just as before by the fact that only one field will be in use in a given system, and so only two lengths of `StaticUInt` (one which fits the polynomials of degree $m-1$ and one for degree $2m-2$) should be encountered.

Julia arrays The first implementation of `StaticUInt` used Julia's native arrays. However, as these are not statically-sized, the compiler cannot infer their length. As a result, they do not provide a significant advantage over the original `BigInt` implementation.

StaticArrays Instead, I used the `StaticArrays` package [**?**], which builds upon Julia's tuples to provide statically-sized arrays. From this package, I explored both the `SVector` and `MVector` types, which supported immutable and mutable arrays respectively. I found that `MVector` was best suited to the kinds of operations that the binary polynomials required. Several of the binary field arithmetic routines involve iterating over the polynomial, and for each term checking a condition and updating a single coefficient. To do this with immutable arrays would mean copying the polynomial hundreds of times with only small changes each time, which was highly inefficient. Mutable arrays, on the other hand, only required one copy at the start of an arithmetic routine (to prevent the result overwriting either of the operands), and from then on it could simply perform inexpensive bit flips and shifts.

Summary

In conclusion, field elements are represented by a composite type `BFieldPoint{D,R}`, where `D` and `R` specify the binary field, and the value of the element itself is held in an array of unsigned integers, of type `StaticUInt{L,T}`.

3.2.2 Reduction

After multiplying the two binary polynomials with degree at most $m-1$ the result is a polynomial with degree at most $2m-2$, which is a not an element of the field $GF(2^m)$, and so needs to have its degree reduced. During the development process, I implemented both reduction methods outlined in section 2.1.1.

Standard reduce This first method takes a polynomial $a(x)$ of degree $m+n$ and iterates over the coefficients a_i of the polynomial from $i = m+n$ down to $i = m$, adding $f(x) \cdot x^i$ to it if $a_i = 1$. To save space, my implementation does not store the entire reduction polynomial $f(x)$ as a field element, instead only storing $f'(x)$ and adding $f'(x)$ to $a(x)$. This means that at the end of the reduction routine there may still be many higher terms in $a(x)$, but these can easily be cleared in operation. Additionally, I wrote a specialised function `shiftedxor(a, b, i)` for `StaticUInt` objects which performs the operation `a ^ (b << i)` (corresponding to $a(x) + b(x) \cdot x^i$) all at once. If, on the other hand, the shift and exclusive-xor operations were to occur separately, the result $f'(x) \cdot x^i$ (which may be large,

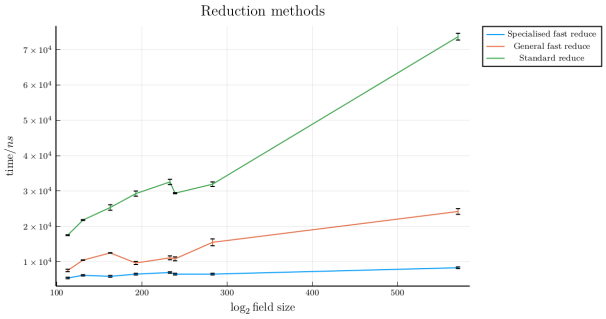
because its degree could be more than m) would need to be stored as an intermediate value.

Fast reduce The previous method works well if the reduction polynomial is an arbitrary polynomial of degree m. However, this second method is more efficient for cryptographic purposes, as it takes advantage of the low number and small spread of bits in the recommended reduction polynomials [**?**] to produce a significantly faster reduction routine. The standard reduction algorithm performs roughly $\frac{m-1}{2}$ `shiftedxor` operations, each of which performs roughly $\frac{m}{W}$ shifts and exclusive-or operations (where W is the word length). The fast reduce algorithm, on the other hand, performs roughly $\frac{m}{W}b$, where b is the number of terms in $f'(x)$. As long as $b < \frac{m-1}{2}$, which is always the case for the standard fields (as m falls in the range of 100 to 600, while b is less than five), the fast reduction algorithm should perform significantly fewer shifts and exclusive-ors, allowing it to achieve far better performance.

Additionally, the fast reduction algorithm performs a fixed number of operations for a given reduction polynomial and input size, a property that helps protect against timing attacks. For each of the standard fields offered by BinaryECC, and for a word size of both 32 and 64 bits, I implemented this method. Due to Julia's multiple dispatch and the fact that the parameters `D` and `R` are part of the type for field elements, calling the reduction function on `BFieldPoint{D,R}` will cause the fast reduce implementation to be automatically chosen if one exists for the field represented by `{D,R}`.

As a part of the iterative refinement process, I also implemented a generic version of the fast reduce algorithm which uses an arbitrary reduction polynomial. Although this version has a larger overhead (as it cannot use hardcoded values, calculated for the exact reduction polynomial), it still achieves greater performance than the standard reduction algorithm and ensures that the package's performance does not suffer too much when a new binary field is used. Not only does this provide some future-proofing for BinaryECC (as new standard fields may be recommended), but it also allowed the performance impact of different reduction polynomials to be explored.

As we see in the digram below, the specialised reduction routines have the best performance, while the standard implementation of reduction is not only much slower, but also sees a faster growth in time as the field order increases.



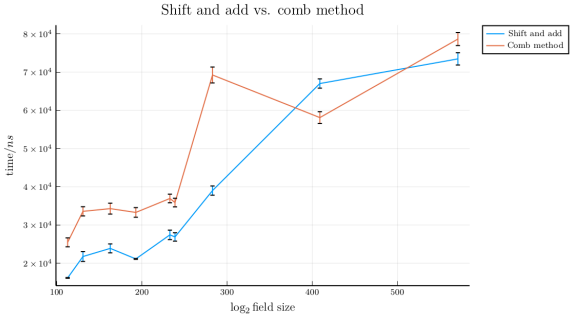
3.2.3 Multiplication

In order to find the most effective implementation for field multiplication, I explored several different algorithms and techniques, including those outlined in section 2.1.1.

Shift and add In the initial prototype of the binary field component, I used a basic right to left, shift and add algorithm, where an accumulator was set to be twice as long as the two input polynomials (so that it could accommodate polynomials with degree up to $2m - 2$), and the second polynomial was then added to it at various offsets, according to the coefficients of the first polynomial. The result was then reduced modulo $f(x)$ by a separate reduction routine.

A second version of this method was implemented, in which the reduction was performed at the same time as multiplication. For a right to left method, instead of calculating $b_i(x) = b(x) \cdot x^i$ on each iteration i, it is possible to simply multiply the value calculated on the previous iteration, $b_{i-1}(x)$ by x. If $b_{i-1}(x)$ is an element of the field (that is, has degree less than m), then by checking whether the m^{th} bit of $b_i(x)$ is one and adding $f(x)$ to it if it is, we can ensure that $b_i(x)$ is also an element of the field. Therefore the accumulating result ($\sum_i a_i b_i(x)$) will always be an element of the binary field $GF(2^m)$, and so will not require reduction at the end.

Comb method Both a left to right and a right to left version of the comb method were implemented. The comb method is implemented with two nested loops: the outer loop iterates over the bits of a word, and the inner loop iterates over the words in the representation of $b(x)$. As can be seen in the figure below, the comb method generally achieves poorer performance than shift and add, which may be attributed to the inner loop accessing a different word on every iteration, causing caching problems.

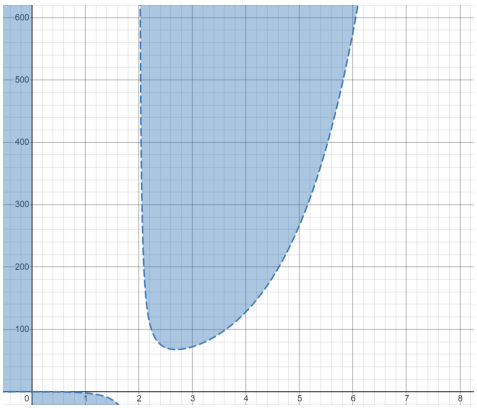


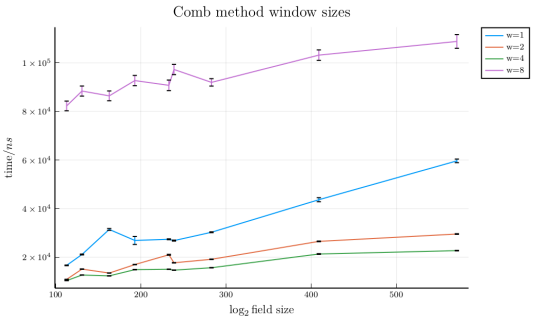
Windowing During an iterative refinement stages, this component was extended to include windowed versions of both the comb and the shift and add methods. Windowing makes a trade-off between the time taken to perform calculations in the main loop of the routine, and the time and space used for precomputation.

If we assign cost c to a single `shiftedxor` operation (which is the key operation in these multiplication implementations), we can say that multiplying $a(x) \cdot b(x)$, where $b(x)$ has t terms, costs roughly ct when we use the shift and add method. On average, elements of $GF(2^m)$ have $\frac{m}{2}$ terms, and so we can say that the average cost for any $b(x) \in GF(2^m)$ is $c\frac{m}{2}$. When we instead use windowing (width w), the average cost is $2^w c\frac{w}{2} + c\frac{m}{w}$. The first term comes from the cost of the precomputation, in which we perform a multiplication with every degree $w - 1$ binary polynomial, of which there are 2^w, with an $\frac{w}{2}$ terms on average. The second term comes from the main loop, which will have $\frac{m}{w}$ iterations that each perform one `shiftedxor`. In order for the windowing to achieve better performance than the unwindowed version, we need that $2^w c\frac{w}{2} + c\frac{m}{w} < c\frac{m}{2}$, and since $c > 0$, we can simplify this constraint to $2^w \frac{w}{2} + \frac{m}{w} < \frac{m}{2}$.

The first figure below shows this constraint visually, with m on the vertical axis and w on the horizontal. We can see that, for the standard field sizes offered by this package (that is, $m = 113$ to $m = 571$), the best window size is 4. This is borne out by the second figure,

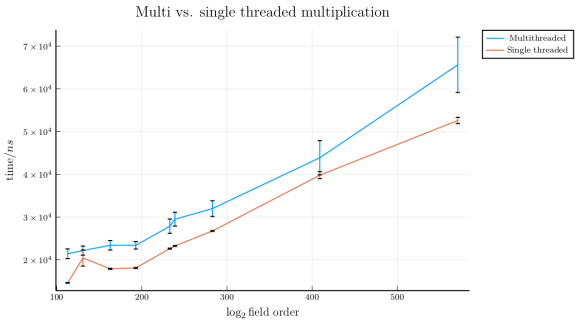
which plots the average time of the shift and add implementation against field size for varying window widths, where we can see that increasing window width improves performance up to $w = 4$, but for $w = 8$ the performance worsens significantly.





Multithreading Julia allows `for` loops to be created, in which a scheduler splits the iteration space between the available threads. This construct can be used within the shift and add method, by noting that the iterations are all independent. Rather than executing every iteration on just one thread with one accumulator, we can have multiple threads each with their own accumulators, splitting the work between themselves. After the main loop has executed, these n accumulators can be added together (with $n - 1$ additions) to produce the final result.

However, as can be seen in the figure below, this does not improve the performance of binary field multiplication.



Squaring

3.2.4 Inversion

3.3 Elliptic Curve Groups

3.3.1 Representation

Groups

An elliptic curve group is represented by the composite type `EC{D,R}`, where `D` and `R` are integer parameters that represent the binary field which the curve is defined over, as described in section 3.2.1. Objects of the type `EC{D,R}` have the fields `a` and `b`, both of type `BFieldPoint{D,R}`, which are the constants in equation for a characteristic-2 elliptic curve, $E : y^2+xy = x^3+ax^2+b$.

Unlike binary field elements, whose type determines the field that they come from, elements of an elliptic curve group contain an object that represents that group. Although this prevents the compiler checking that operations only occur on elements of the same group at compile time (necessitating an extra type, `ECMismatchException`, that inherits from `Exception` and can be thrown at runtime), it is more suitable for the way elliptic curve groups are used. While the binary field is not expected to be changed for a given system, the choice of elliptic curve group may change frequently, leading to many different specialised versions of the same functions to be compiled if the elliptic curve group were a part of the type system. Additionally, the values of `a` and `b` are likely to be hundreds of bits long (as they belong to a binary field of order 2^m, where m is in the hundreds), and so cannot be simply stored in a bits type, which is a requirement for parameter values. This issue could be avoided by allowing only Koblitz curves (in which $a, b \in \{0, 1\}$), but this would make the package unnecessarily restrictive.

Elements

For the representation of elements, we have the abstract type `AbstractECPoint{D,R}` with three concrete subtypes `ECPointAffine{D,R}`, `ECPointJacobian{D,R}` and `ECPointLD{D,R}`. This allows high-level routines, such as the Montgomery powering ladder (section 2.1.2), to have one implementation that can be used for any of the three point representations, while lower level routines, such as point doubling, can still be specialised for each.

The type `ECPointAffine{D,R}` contains three fields: `x` and `y`, both of type `BFieldPoint{D,R}`, to hold the value of the point, and a field `ec` of type `EC{D,R}` to hold the information about the group itself. The projective representations, `ECPointJacobian{D,R}` and `ECPointLD{D,R}`, are similar apart from one additional field, `z` of type `BFieldPoint{D,R}`.

Point at infinity Each elliptic curve group must also contain the identity, known as the point at infinity and written in mathematical notation as $\mathcal{O}$. One possible implementation of $\mathcal{O}$ is to create a type for it, `ECPointO{D,R}`, with just one field, `ec`, storing information about the elliptic curve group it belongs to. It would therefore need to be a concrete type (as Julia only allows abstract types to have behaviour, and not state), and for it to fit into the existing type system it could subtype `AbstractECPoint{D,R}`. However, this would mean that certain functions, such as point addition, would suffer from type instability, in that the types of the function's arguments would not provide the compiler with enough information to deduce the (concrete) type of the result. For example, performing the addition `p1 + p2` (where `p1` and `p2` are both instances of the `ECPointAffine{D,R}` type) could return either an `ECPointAffine{D,R}` (in the general case) or an `ECPointO{D,R}` (if `p1 == -p2`). The presence of type instability limits the performance of the code that can be produced by compiler, because if it cannot know at compile time what the type of an object is, it has to use dynamic dispatch and work out which functions to call at runtime.

Instead, we can note that in all three point representations, the point at the origin will never satisfy the elliptic curve equation (for non-supersingular curves, that is, where $b \neq 0$). Therefore we can redefine this point to be $\mathcal{O}$, allowing it to have the same type as every other point on the curve and avoiding the issue of type instability.

Summary

In conclusion, elliptic curve groups are represented by the type `EC{D,R}`, which has fields `a` and `b`, both of type `BFieldPoint{D,R}`. Points on an elliptic curve have a concrete type of either `ECPointAffine{D,R}`, `ECPointJacobian{D,R}`, or `ECPointLD{D,R}`. They have fields `x`, `y` and (for the latter two only) `z`, which store the coordinates of the point, or zeroes if the point is $\mathcal{O}$. They also have the field `ec`, of type `EC{D,R}`, to store the group information. These three point types are all subtypes of `AbstractECPoint{D,R}`, and operations involving multiple points will throw an `ECMismatchException` if the points belong to different groups.

3.3.2 Point addition and doubling

The two basic operations that can be performed on elements in an elliptic curve group are addition and doubling, outlined in section 2.1.2. Here, in this section, I derive formulae for addition and doubling of projective points, and then I discuss their implications.

Jacobian coordinates Jacobian coordinates, as discussed in section 2.1.2, have parameters $c = 2$ and $d = 3$, meaning that the Jacobian coordinates (X, Y, Z) map to the affine coordinates $(\frac{X}{Z^2}, \frac{Y}{Z^3})$. New formulae for point addition and doubling can therefore be produced by substituting $x = \frac{X}{Z^2}$ and $y = \frac{Y}{Z^3}$ into the formulae for affine coordinates (section 2.1.2). For addition, this gives:

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

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

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

$$\begin{aligned} X_3 &= A^2 + AB + C^3 + aB^2) \cdot Z_1^4 \\ Y_3 &= A(X_1 Z_3^2 + X_3 Z_1^2) + C Z_2 (X_3 Z_1^3 + Y_1 Z_3^2) \\ Z_3 &= B Z_1^2 \end{aligned}$$

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

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

$$\begin{aligned} X_3 &= (A^2 + AB + aB^2) \cdot Z_1^8 \\ Y_3 &= B X_1^2 Z_3^2 + (A + B) X_3 Z_1^4 \\ Z_3 &= B Z_1^4 \end{aligned}$$

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

López-Dahab coordinates For this projective coordinate system, the parameters are $c = 1$ and $d = 2$, and the formulae for point addition (derived through the same method as above) are:

$$\begin{aligned} X_3 &= A^2 + AB + BC^2 + aB^2 \\ Y_3 &= ACZ_2(X_1Z_3 + X_3Z_1) + C^2Z_2^2(X_3Z_1^2 + Y_1Z_3) \\ Z_3 &= B^2 \end{aligned}$$

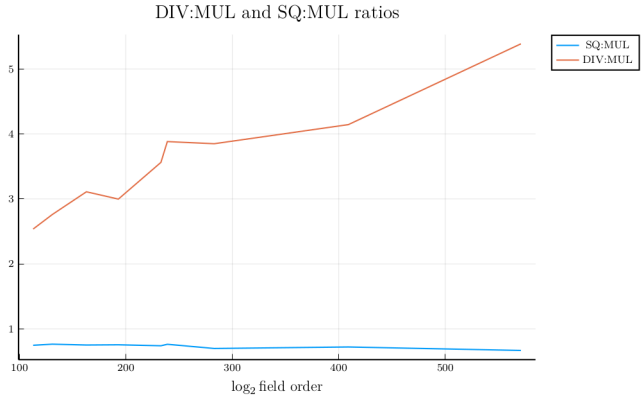
$$\text{where} \qquad A = Y_1Z_2^2 + Y_2Z_1^2 \qquad C = X_1Z_2 + X_2Z_1 \qquad B = Z_1Z_2C$$

The formulae for point doubling are:

$$\begin{aligned} X_3 &= A(A + B) + aB^2 \\ Y_3 &= X_1^4Z_3 + X_3B(A + B) \\ Z_3 &= B^2 \end{aligned}$$

$$\text{where} \qquad A = X_1^2 + Y_1 \qquad B = Z_1X_1$$

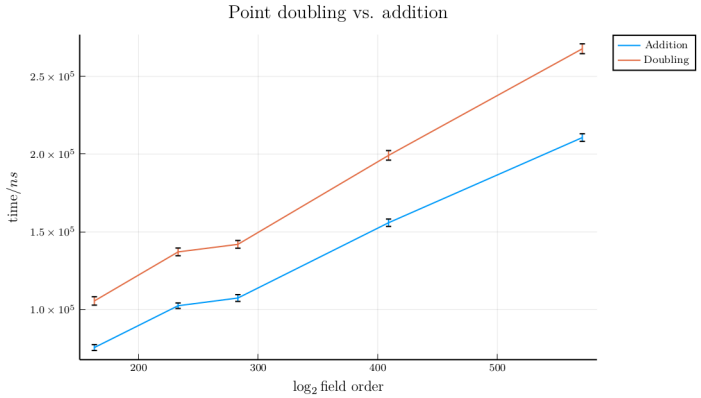
Performance When implementing each of these routines, I attempted to minimise the number of arithmetic operations that were used. In the underlying binary field, the three most costly operations are division, multiplication and squaring. From the figure below, showing the ratios of division and squaring times to multiplication times as field sizes grow, we can see that division takes roughly three to five times as long as multiplication, while squaring only takes roughly half the time.

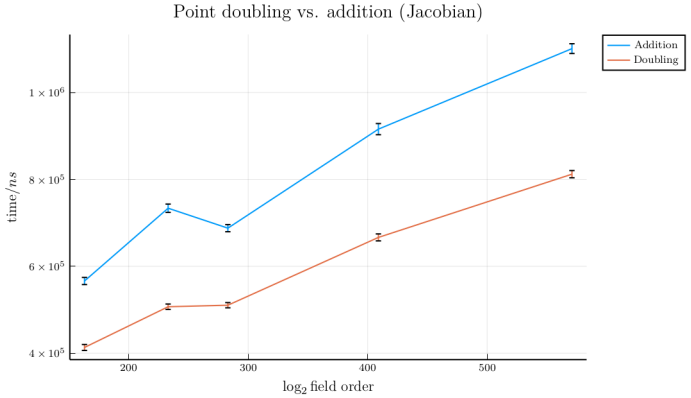


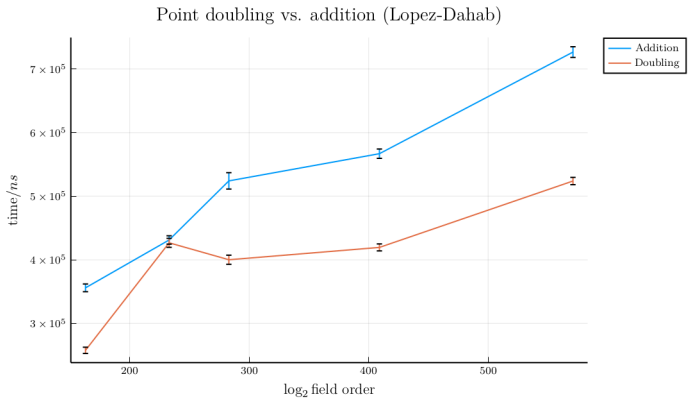
For this reason, it is desirable to reduce the number of divisions performed, and so to compare the different point representations more clearly, the table above shows the costs of both point doubling and addition for each. Although we can see that the projective representations (Jacobian and López-Dahab) can perform these routines without any divisions, it is unfortunately at the cost of many more multiplications and squarings, which would require a much higher ratio of division time to multiplication time for it to be worthwhile. As I had calculated the formulae for these routines by hand, I searched for other implementations of projective point doubling and addition that used fewer multiplications. However, I did not find any which reduced the number of multiplications sufficiently for them to outperform the affine point routines.

Representation	Addition			Doubling		
	D	M	S	D	M	S
Affine	1	1	1	1	1	2
Jacobian	0	6	20	0	10	6
López-Dahab	0	16	4	0	6	3

The performance of the doubling and addition routines for each of the representations is below, and we can see that







3.3.3 Scalar multiplication

Multiplication of an elliptic curve point by a scalar is the key operation for elliptic curve cryptography, as it allows you to calculate the point $P = nG$ to be used in ECDLP. In this section, I outline several different methods for scalar multiplication and discuss the different trade-offs that they make.

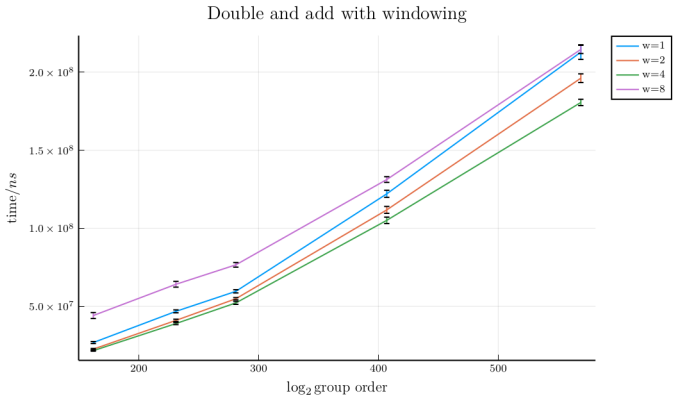
Double and add

This is the most basic method (described in section **??**), and was used to implement the elliptic curve group prototype. As this method requires no manipulation of the point coordinates, it was initially written to simply take an instance of `AbstractECPoint`.

However, this meant that the point (passed in as an argument) had an abstract type at runtime, meaning that when a point doubling or addition routine is called on it, its concrete type must first be found causing a hit to performance. Therefore it was rewritten to take an object of any type `T` which subtypes `AbstractECPoint`. This subtle difference means that the first time the function is called with, for example, a point of type `ECPointAffine`, a specialised version of that function is precompiled with the affine versions of point addition and doubling.

Therefore the first call to the double and add function will incur a higher overhead, but all subsequent calls are able to execute faster. An alternative way to avoid this issue of unknown types at runtime would be to have three almost identical versions of the function, with the only difference between them being their signatures: each would take a different (concrete) point type. Each version would have the same body, but would take a different concrete type of point. However, this would be unnecessarily expand the source code and make it more tedious to maintain.

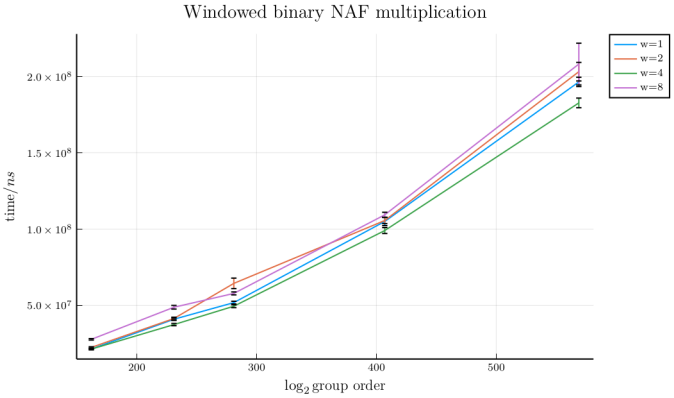
Windowing As with the binary field multiplication, windowing can also be applied to scalar multiplication in order to calculate $n \cdot P$ more efficiently. As we can see in the figure below, a window size of $w = 4$ produces a slight performance advantage, which is more noticeable as the group order increases. We can also note that, just as with binary field multiplication, the performance drops significantly when $w = 8$. This is likely because the amount of precomputation required ($2^8 - 2 = 254$ additions) is the greater than the number of additions we would expect to occur in the main loop in the absence of windowing (that is, $\frac{1}{2} \log_2 n$ additions). Therefore we are simply replacing the work in the main loop with a larger amount of work in the precomputation stage, leading to poor performance, especially for the groups with a lower order.



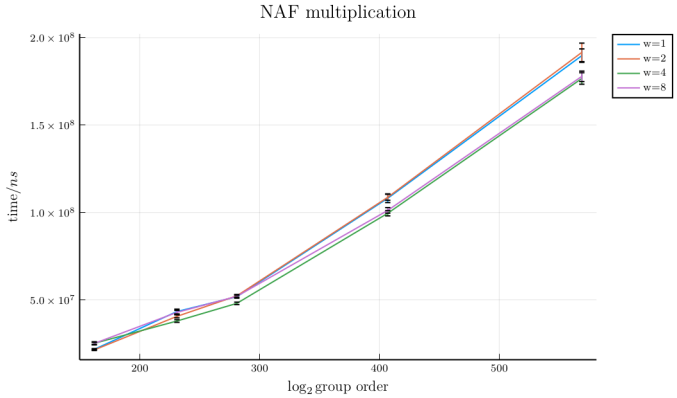
Non-Adjacent Form

The non-adjacent form of an integer is the representation that has the fewest nonzero digits of all signed digit representations, as outlined in section **??**. This allows them to reduce the number of double and add operations used in the main loop, at the cost of the extra time and space required to compute and store the NAF of the scalar multiplier.

Binary NAF figure



Width-w NAF figure



Windowing

Montgomery Powering Ladder

General version

Affine version

3.4 Cryptographic primitives

3.4.1 Prime fields

3.4.2 Curve Domain Parameters

3.4.3 Key pairs

3.4.4 ECDSA

3.4.5 ECDH

3.5 Fine tuning

3.6 Testing

3.7 Documentation

3.8 Limitations

3.9 Summary

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

Conclusions

Appendix A

Algorithms

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 $\overline{x_R} := \text{int}(x_R)$;
3 $r := \overline{x_R} \mod n$;
4 **if** $r = 0$ **then**
5 | Return to step 1;
6 **end**
7 $H := \text{hash}(M)$;
8 $e := \text{int}(H)$;
9 $s := k^{-1}(e + rd_U) \mod n$;
10 **if** $s = 0$ **then**
11 | Return to step 1;
12 **end**
13 Output $S = (r, s)$;

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
$H := \text{hash}(M)$;
$e := \text{int}(H)$;
$u_1 := es^{-1} \mod n$;
$u_2 := rs^{-1} \mod n$;
$R := u_1 G + U_2 Q_U$;
if $R = \mathcal{O}$ **then**
  Output "invalid" and stop;
end
$\overline{x_R} := \text{int}(x_R)$;
$v := \overline{x_R} \mod n$;
Output $S = (r, s)$;
if $r = v$ **then**
  Output "valid" and stop;
else
  Output "invalid" and stop;
end

Appendix B

Project Proposal