Molly Fryatt

Optimising Elliptic Curve Cryptography over Binary Finite Fields in Julia

Computer Science Tripos – Part II

St John's College

2021

Declaration of originality

I, Molly 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 14 May 2021

Proforma

Candidate Number:	**2407G**
Project Title:	**Optimising Elliptic Curve Cryptography over Binary Finite Fields in Julia**
Examination:	**Computer Science Tripos – Part II, 2021**
Dissertation Word Count:	**11,889**[1]
Software Line Count:	**3,492**[2]
Project Originator:	Dr Markus Kuhn
Project Supervisor:	Dr Markus Kuhn

Original aims of the project

This project aims to analyse and compare the performance of a range of different optimisation techniques for arithmetic in both binary fields and elliptic curve groups. As a part of this process, the algorithms and data structures are to be implemented in the language Julia, and then collated together into a high performance package for Elliptic Curve Cryptography (ECC).

Work completed

The original aims of this project were fully met, with the resulting package, BinaryECC, offering a wide range of algorithms for key operations (each offering high performance or important security properties), as well as a flexible and efficient representation of mathematical objects. As a result of the analysis and consideration that went into its development, BinaryECC outperforms pre-existing open-source implementations in much of its functionality.

Special difficulties

None.

[1]This word count was computed by `texcount`
[2]This line count was computed by `cloc`

Contents

List of Figures

1 Introduction

In this dissertation, I use the Julia [1] language to explore and evaluate a range of techniques for the optimisation of binary field arithmetic and elliptic curve group arithmetic. The results of this analysis are used to create a high performance Julia package for Elliptic Curve Cryptography (ECC).

1.1 Motivation

Many cryptographic protocols, such as Diffie-Hellman key exchange ([2], 1976), depend upon the assumption that it 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 finding discrete logarithms any suitable cyclic group, but they all have exponential time complexity. However, for the cyclic group $\mathbb{Z}_p^*$, the Index Calculus Algorithm is able to compute discrete logarithms in sub-exponential time. Therefore to achieve 80-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 [3].

An alternative is to use elliptic curve groups, in which the group operation involves "bouncing" a point around an elliptic 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). As the Index Calculus algorithm is not applicable to this group and there are no (known) sub-exponential algorithms for solving the ECDLP, this problem is thought to be computationally harder (Silverman, [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 (and therefore reducing the storage and transmission requirements).

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 order 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.

For this project, Julia has two key advantages. Firstly, it is a high performance language (its core is implemented in a mixture of C and itself), with benchmarks showing that its execution times for a selection of algorithms are within a factor of two of their C implementations [1]. Secondly, Julia is a high level language with several useful features, such as powerful macros (allowing specialised functions to be generated at runtime) and parametric types.

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

1.2 Related work

This project compares and analyses the performance of a variety of algorithms and techniques for ECC over binary fields. The results are then used to produce a Julia package which allows the user a large amount of flexibility (by allowing custom curves and fields to be used and their underlying representation to be altered), 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.

Related to this area is the book "Guide to Elliptic Curve Cryptography" (Hankerson, Menezes and Vanstone [3]), which describes many algorithms and attacks for ECC, as well as other papers cited throughout this dissertation which provide additional mathematical background and timing-attack secure algorithms. Other key works are the standards documents produced by bodies such as NIST [5] and SECG [6], which detail cryptographic primitives and standard values.

Within the Julia ecosystem, the package "GaloisFields.jl", by Kluck 2018 [7], provides support for Galois fields. However, it is designed to offer many fields ($\mathrm{GF}(p^m)$, for $m \geq 1$ and arbitrary prime p), and so it is not optimised for binary fields in particular. There is also a computer algebra package, "Nemo.jl" [8], that offers many features including finite field arithmetic (both prime and prime power order). Rather than implementing this arithmetic directly in Julia, Nemo wraps a C library for number theory called Flint [9]. Currently, Julia does not have any packages that support elliptic curves over binary fields. However, OpenSSL [10] is a general-purpose cryptography library, written primarily in C, that does support elliptic curves defined over both prime and binary fields.

1.3 Challenges

Undertaking this project presented me with two key challenges: firstly, the use of Julia, a language that was new to me and which only released a stable version in 2018; and secondly, the fact that much of the mathematics required to understand ECC is beyond the scope of material taught in the Tripos. Due to these challenges, and the constraint of this being a one-year project, the focus is more on implementing an efficient package than on security aspects such as resistance to side-channel attacks (e.g. power analysis and fault attacks, as discussed in [11]), although some consideration is given to timing attack resistance.

1.4 Results

As a result of applying and comparing a range of different optimisation techniques, I have successfully built a package that can perform arithmetic in elliptic curve groups, offering several point representations and alternative algorithms to perform the key operations, some of which are designed to provide high performance and others designed for resistance to timing attacks. It also supports binary field arithmetic, offering both predefined standard fields and the ability to create new fields, as well as a customisable representation of binary field elements (e.g. to suit the system's word size or collect extra information). In my evaluation, I find that this package achieves significantly faster binary field arithmetic than any other pre-existing open-source Julia implementation. In addition to this, BinaryECC contains example implementations of several cryptographic primitives with performance in the same range as OpenSSL, a commonly used C library for cryptography.

2 Preparation

2.1 Background

ECC uses elliptic curve groups, which are formed from the set of points in a field that are on a given curve. In cryptography, this field is either a binary finite field, $\mathrm{GF}(2^m)$, or a prime order field, $\mathrm{GF}(p)$, and it is the first of these two which this project focusses on. In this section, I begin by describing binary fields and then move on to elliptic curve groups, before outlining several cryptographic protocols 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.

Fields with a finite number of elements are known as Galois fields, and are written as $\mathrm{GF}(p^m)$ where p is a prime and $m \geq 1$. The order of the field, $\#\mathrm{GF}(p^m) = p^m$, is the number of elements that it contains.

This project uses binary Galois fields constructed with a polynomial basis representation. A binary field $\mathrm{GF}(2^m)$ 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 that I will refer to as $f(x)$.

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). Since addition and subtraction are equivalent in the field $\mathbb{F}_2$ (both can be seen as exclusive-or), they are also equivalent in binary fields. Below is an example of addition.

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

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 In 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.

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

Comb method For this method, we assume that the field elements have been stored as an 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 may be cheaper to calculate $a(x)x^i$ (for $0 \leq i < W$) 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 (i.e., $(a(x)+b(x))^2 = a(x)^2 + b(x)^2$) and can therefore be computed with a faster specialised 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 [5] and SECG [6], provide recommended reduction polynomials for commonly used field orders.

Because a reduction polynomial for $\mathrm{GF}(2^m)$ 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 $x^m \equiv f'(x) \mod f(x)$. 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:

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

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. This process is iterated until it yields a polynomial with degree less than m.

Fast reduction Tri- and pentanomial reduction polynomials, especially those whose terms are close together, can be used to perform reduction more efficiently. For example, the field $\mathrm{GF}(2^{163})$ has the recommended reduction polynomial $f(x) = x^{163} + x^7 + x^6 + x^3 + 1$, which gives us the following congruences modulo $f(x)$:

$$\begin{array}{lllll} x^{Wn} & \equiv x^{Wn-163+7} & +x^{Wn-163+6} & +x^{Wn-163+3} & +x^{Wn-163} \\ x^{Wn+1} & \equiv x^{Wn+1-163+7} & +x^{Wn+1-163+6} & +x^{Wn+1-163+3} & +x^{Wn+1-163} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ x^{W(n+1)-1} & \equiv x^{W(n+1)-1-163+7} & +x^{W(n+1)-1-163+6} & +x^{W(n+1)-1-163+3} & +x^{W(n+1)-1-163} \end{array}$$

By considering the columns of the above congruences as sums, we can then note that for any binary polynomial $a(x)$ with coefficients a_i and degree m' such that $W(n-1) \le m' < Wn$, we have the following congruence modulo $f(x)$:

$$\begin{aligned} \sum_{i=0}^{W-1} a_{Wn+i} x^{Wn+i} \equiv & \sum_{i=0}^{W-1} a_{Wn+i} x^{Wn-163+7+i} + \sum_{i=0}^{W-1} a_{Wn+i} x^{Wn-163+6+i} \\ & + \sum_{i=0}^{W-1} a_{Wn+i} x^{Wn-163+3+i} + \sum_{i=0}^{W-1} a_{Wn+i} x^{Wn-163+i} \end{aligned}$$

Therefore, by adding all five sums to $a(x)$ we obtain a new polynomial $a'(x)$ which has degree $Wn-1$ or less, such that $a(x) \equiv a'(x) \pmod{f(x)}$. By repeatedly applying this technique to a polynomial $a(x)$, reducing its degree by W on each iteration with only five binary field additions, we can convert $a(x)$ into an element of $\mathrm{GF}(2^{163})$ much more efficiently than with the reduction standard method.

Inversion

Every element of a binary field, except zero, 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, written as $\#E(\mathbb{K}) = nh$, where n is the largest prime factor and h is its cofactor. The security level of the group is $\frac{1}{2}\log_2 n$, and so we prefer to use curves which are cyclic (i.e. $h = 1$) or almost cyclic ($h \in \{2, 4\}$).

For curves defined over binary fields, we use the fields $\mathbb{L} = \mathbb{K} = \mathrm{GF}(2^m)$. 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 adds points together or doubles them, and every point has an inverse. For a curve with a binary underlying field, the inverse of a point $P = (x, y)$ is $-P = (x, x + y)$, and $\mathcal{O}$ is its own inverse.

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 $P_3 = (x_3, y_3)$, with:

$$\begin{aligned} x_3 &= \lambda^2 + \lambda + x_1 + a \\ y_3 &= \lambda(x_1 + x_3) + x_3 + y_1 \\ \lambda &= \frac{y_1 + y_2}{x_1 + x_2} \end{aligned} \tag{2.10}$$

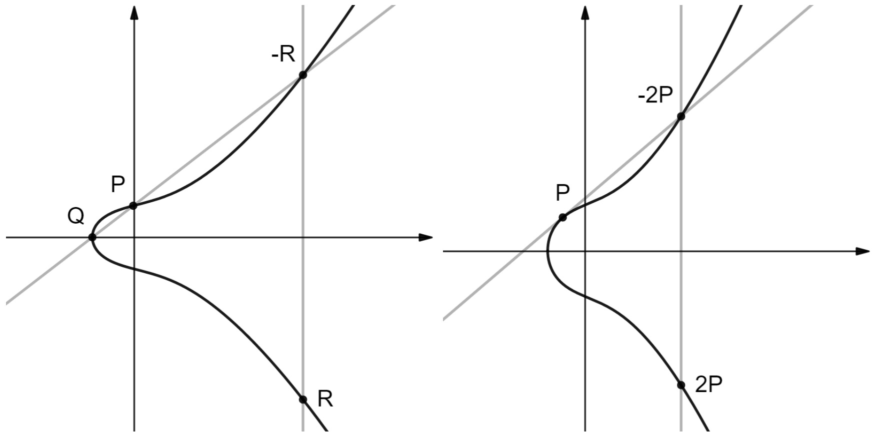


Figure 2.1: On the left is an illustration of the chord rule, performing the addition $P+Q=R$. First, a line is drawn through P and Q. This line intersects the curve again at $-R$, and so we reflect about the x-axis to find R, the sum of P and Q.
On the right is the tangent rule, used to double the point P. First, a tangent to the curve is drawn at point P. This line intersects the curve again at $-2P$, and so we reflect about the x-axis to find the result of the doubling operation, $2P$.

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 $P_1 = (x_1, y_1)$, $P_3 = 2 \cdot P_1 = (x_3, y_3)$, with:

$$\begin{aligned} x_3 &= \lambda^2 + \lambda + a = x_1^2 + \frac{b}{x_1^2} \\ y_3 &= x_1^2 + \lambda x_3 + x_3 \\ \lambda &= x_1 + \frac{y_1}{x_1} \end{aligned} \tag{2.11}$$

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

- $P_1 + \mathcal{O} = P_1$, since $\mathcal{O}$ is the neutral element
- $P_1 + (-P_1) = \mathcal{O}$, 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, we can see 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.

$$P \cdot k = \sum_i 2^i k_i P \text{ where } k_i \in \{0, 1\} \tag{2.12}$$

Binary NAF Because the inverse of a point is cheap to compute (since $-P = (x, x + y)$, 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 non-zero values, producing a more sparse representation of k (the average density of zeros across NAFs of the same length is $\frac{1}{3}$, [3]). This allows us calculate $P \cdot k$ with fewer curve point additions.

$$P \cdot k = \sum_i 2^i k_i P \text{ where } k_i \in \{-1, 0, 1\} \tag{2.13}$$

Width-w NAF The concept of an NAF 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 $1/(w + 1)$ [3]. 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 [12]), which 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 equal-length values of k, and so is 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 is 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 $(x, y) \mapsto (x : y : 1)$, with inverse map $(X : Y : Z) \mapsto (X/Z^c, Y/Z^d)$ 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 can be expensive compared to multiplications.

There are two forms of projective coordinates which I explore in this project: Jacobian coordinates and López-Dahab coordinates.

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

$$E : Y^2 + XYZ = X^3 + aX^2Z^2 + bZ^6 \tag{2.14}$$

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.

$$E : Y^2 + XYZ = X^3Z + aX^2Z^2 + bZ^4 \tag{2.15}$$

2.1.3 Prime fields

Several cryptographic primitives, such as the Elliptic Curve Digital Signature Algorithm (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 Euclidean algorithm.

2.1.4 ECC

In elliptic curve groups, we have the ECDLP: given $P = G \cdot k$ (where the group $E(\mathrm{GF}(q))$ and the generating point G are known), find k. For carefully chosen curve parameters, there are currently no known algorithms to solve this problem in sub-exponential time[1], 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 curves over binary fields, these details, known as Elliptic Curve Domain Parameters, are stored in a septuple (SEC 1, [13]): $T = (m, f(x), a, b, G, n, h)$.

Element	**Purpose**
m	$\log_2$ of the underlying field order, i.e. $\#\mathrm{GF}(2^m)$
$f(x)$	Irreducible polynomial of degree m, specifying the field $\mathrm{GF}(2^m)$
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	Order of point G (prime)
h	Cofactor of n, i.e. $h = \#E(\mathrm{GF}(2^m))/n$

[1]There are faster algorithms for curves with specific properties, such as having an underlying field $\mathrm{GF}(2^m)$ where m is composite, or where the order is $\#E(\mathrm{GF}(2^m)) = 2^m$, or where the order is $\#E(\mathrm{GF}(q)) = nh$ with a large cofactor h.

Another important primitive are Elliptic Curve Key Pairs, which are a tuple (d, Q) associated with a septuple T of curve domain parameters [13]. 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 ECDSA, a variant of the Digital Signature Algorithm (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 [13].

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. 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.

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 the ECDLP, known as the Elliptic Curve Diffie-Hellman Problem (ECDHP): given $Q_1 = d_1G$ and $Q_2 = d_2G$, determine d_1d_2G [13].

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 the 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 a poor implementation.

- **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 [14] 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 and López-Dahab representations)**
  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.
- **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 [15] from SECG, and the equivalent document from NIST [16] 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 [13] from SECG, which lists and describes in detail all of the cryptographic primitives that I will need to implement. I will use test vectors [17] to verify the implementation of ECDSA.

2.3 Methods and tools

2.3.1 Software engineering principles

Due to the structure of the project, it is necessary to have a 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. 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.

Testing strategies

Testing will be integrated into the development process, with unit tests for each component. This will allow any alterations to be verified quickly and before they are incorporated into BinaryECC, reducing the amount of time spent debugging by making it easier to pinpoint the offending code. The unit tests will be built from test vectors, providing a good coverage of the expected inputs, as well as from edge cases which have known results (e.g. $\frac{x}{x} = 1$). Since each core component builds upon the previous ones (ECC relies on elliptic curve groups, which in turn rely upon binary finite fields), integration testing is performed by testing functions in the top layer, ECC, which necessarily interact with the layers below.

2.3.2 Choice of tools

This dissertation was written using LaTeX, using the TeXMaker editor [18]. BinaryECC was written in Julia 1.5, using Juno [19], an Integrated Development Environment, and it depends on two Julia packages:

1. StaticArrays [20]: to provide statically sized arrays, something which is not a built in feature of Julia, to enable certain performance improvements discussed in section 3.2.1

2. SHA [21]: to provide a hashing function, used in the implementation of cryptographic protocols

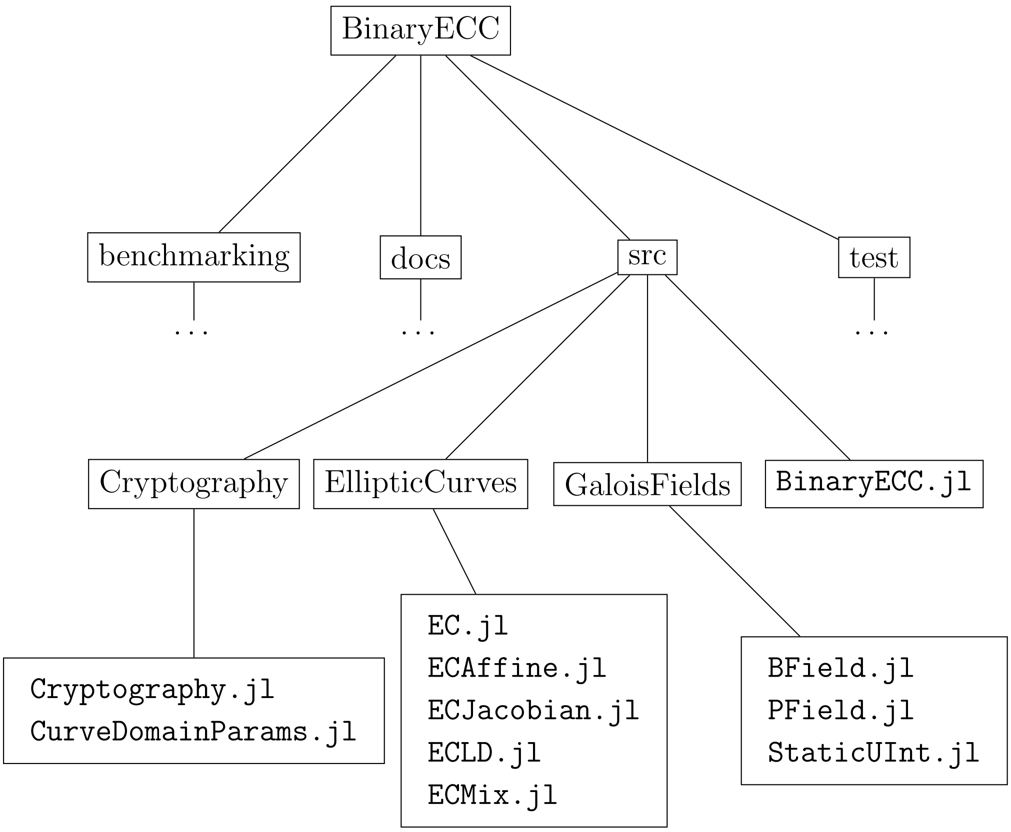
Julia was also used for peripheral activities such as data cleanup, testing, benchmarking, and for producing graphs and documentation. For these purposes, I used several other packages: Test, for creating automated unit tests; BenchmarkTools [22], to analyse the performance of different algorithms; Plots [23] and LaTeXStrings [24], to produce graphs visualising performance results using PGFPlots as a backend; and Documenter [25], 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.

3 Implementation

3.1 Repository overview

The code for BinaryECC, as shown in the diagram, was written from scratch. For the sake of conciseness, the contents of the "docs", "test" and "benchmarking" directories are not shown, although they too were written entirely for this project.



3.2 Binary fields

3.2.1 Representation

Given an element of a binary field, arithmetic routines need to know both the value of the element and the field that the element is in. Julia divides its types into primitives (e.g. booleans, characters, floating points and integers) and composites which contain a set of named fields (similar to structs in C), 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)`. The advantage of this is that there would be no redundancy: the field information is only 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 the 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. The field information can either be held as an explicit field within a struct, or as part of the object's type, as discussed later.

Another aspect to consider is how binary polynomials will be stored. The most straightforward approach here is to store them in unsigned 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 implementation of many arithmetic operations.

Field information

To uniquely identify a binary field $\mathrm{GF}(2^m)$, we need to know both $\#\mathrm{GF}(2^m) = 2^m$, the field order, and $f(x)$, an irreducible polynomial of degree m. One approach would be to store only the reduction polynomial, so that its degree would provide m, the logarithm of the field order. An alternative is to store m and only the lower terms of the reduction polynomial (that is, $f'(x)$ where $f(x) = x^m + f'(x)$). In cryptographic applications, the reduction polynomials, such as $f(x) = x^{163} + x^7 + x^6 + x^3 + 1$ for $\mathrm{GF}(2^{163})$ [15], tend to have a small number of low terms, and so the second approach makes a more efficient use of space (the $\mathrm{GF}(2^{163})$ given will use just two bytes: one byte to store each of m and $f'(x)$).

This information could be stored as a named value within the struct representing a binary field element, which would allow the reduction polynomial (which would be stored in an integer, for example using Julia's `BigInt` type) to be arbitrarily large.

However, I instead created a parametric composite type for the field elements, `BFieldElt{D,R,T,L}`, where `BFieldElt` is the name of the type, the parameters `{T,L}` are the word type and number of words that hold the polynomial representation of the element itself, and the integer parameters `{D,R}` identify the field $\mathrm{GF}(2^m)$. The parameter `R` is the lower terms of the reduction polynomial, $f'(x)$, and the other parameter `D` is $\deg(f(x)) = m$. Since elements of different fields will have different types, the compiler can check that arithmetic operations have been called on elements of the same field, rather than having to perform a runtime check, improving runtime performance. Furthermore, the field information will not use any memory at runtime, since it is not stored as a value within a struct.

One potential problem is that Julia will compile specialised functions for each different `BFieldElt{D,R,T,L}`, impacting the compile time. Even worse, if a new field is created during the running of a program, then first time a function is called with that new field type, a specialised version of that function will also need to be compiled, damaging runtime performance as well. 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 parametric types. A second problem is that only "bits" types (e.g. numbers, booleans, etc.) can be used as parameter values, and so we are restricted to at most a `UInt128` to store $f'(x)$, meaning it must be of degree 127 or lower. However, as noted above, reduction polynomials recommended for cryptographic use tend to have a low degree $f'(x)$, and so this is not overly restrictive.

Point information

The struct `BFieldElt{D,R,T,L}` contains an integer value representing a binary polynomial, often hundreds of bits long. The initial version of `BFieldElt` used objects of the pre-existing `BigInt` type (and therefore did not use the parameters `T` and `L` at all), allowing a working prototype of this component 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 means that the compiler is unable generate high-performance code. The alternative was to store binary polynomials in arrays of fixed-size unsigned integers (of length `L` and type `T`), for which I created a new type called `StaticUInt`.

StaticUInt

Creating a new type for statically-sized unsigned integers places a layer of abstraction between the representation of the element values and the arithmetic to manipulate those values. This made it much easier to iterate through several versions of the `StaticUInt` type, since the arithmetic routines themselves did not need to be altered to enable comparison between different representations.

Like `BFieldElt`, `StaticUInt` is also a parametric type, this time only with parameters `{L,T<:Unsigned}`. To enforce the unsigned aspect of `StaticUInt`, `T` must subtype `Unsigned`, Julia's abstract type for unsigned integers. The parameterisation of the word type adds flexibility, as the package does not need to assume any particular word size, and also opens up opportunities for future development by allowing, for example, side-channel attacks to be explored (by using an unsigned integer type with logging capabilities).

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` should be encountered (one which fits the polynomials of degree $m-1$ and one for degree $2m-2$).

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

StaticArrays.jl Instead, I used the `StaticArrays` package [20], which builds upon Julia's tuples to provide statically-sized arrays, just as in languages like C. 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 binary field arithmetic routines, such as multiplication, involve iterating over the polynomial and conditionally updating a single coefficient in each iteration. To do this with immutable arrays would mean creating and destroying hundreds of `SVector` objects with only minor differences, which is highly inefficient. Mutable arrays, on the other hand, only

```
#generic field type and helper function
struct BFieldElt{D,R,T<:Unsigned,L}
    value::StaticUInt{L,T}
end
B(D, R, T) = BFieldElt{D, R, T, ceil(Int,D/bitsize(T))}

#predfined field and helper function
BFieldElt163{T<:Unsigned, L} = BFieldElt{163, UInt16(128+64+8+1), T, L}
B163(T) = BFieldElt163{T, ceil(Int,D/bitsize(T))}
```

Figure 3.1: Julia definitions for binary finite field types, including an example of a predefined field GF(2^{163}) with reduction polynomial $f(x) = x^{163} + x^7 + x^6 + x^3 + 1$.

require one copy at the start of an arithmetic routine (to prevent the result overwriting either of the operands), and from then on it can simply perform inexpensive bit flips and shifts on that one object.

Summary

In conclusion, field elements are represented by a composite type `BFieldElt{D,R,T,L}` (as shown in figure 3.1) where `{D,R}` specifies the binary field and `{L,T}` specifies their representation. The standard binary fields offered by BinaryECC are provided as specialised field element types, where the parameters for its element representation are left open, along with a function that allows the user to select the word type `T` and then returns a binary field type with a suitable number of words `L`. The package also allows users to easily define their own custom binary fields, by setting the `D` and `R` parameters themselves.

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 may not be an element of the field GF(2^m), and so may need to have its degree reduced.

Standard reduce The implementation of standard reduce, as described in section 2.1.1, is below. In this routine (figure 3.2), a deep copy of the input element's value is performed to create a new element, `b`, which is then repeatedly mutated within the loop[1], reducing the degree of `b`. The function `shiftedxor!(a, b, i)` was written to perform the operation `a ^ (b << i)` (corresponding to $a(x) + b(x) \cdot x^i$) all at once, since this is a very commonly used operation within the field arithmetic.

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 [15] to produce a significantly faster reduction routine. To reduce a

[1] In Julia, a function ends with an exclamation mark if it writes into preallocated memory, and so my functions `flipbit!` and `shiftedxor!` update the value of `b` rather than returning a new `StaticUInt`.

```
function reduce(a::BFieldElt{D,R,T,L}) where {D,R,T,L}
    b = copy(a.value)
    r = StaticUInt{ceil(Int,128/bitsize(T)),T}(R)
    for i in (2*D):-1:D
        if getbit(b, i)==1
            flipbit!(b, i)
            shiftedxor!(b, r, i-D)
        end
    end
    b = changelength(b, ceil(Int,D/bitsize(T)))
    return BFieldElt{D,R,T,ceil(Int,D/bitsize(T))}(b)
end
```

Figure 3.2: Julia code for standard binary field reduction.

binary polynomial of degree $2m-2$ to an equivalent polynomial of degree $m-1$, the standard reduction algorithm performs roughly $\frac{m-1}{2}$ `shiftedxor!` operations, compared to $\frac{m-1}{W}b$ of these operations (where b is the number of terms in $f(x)$ and W is the word size) for the fast reduction algorithm. As long as $\frac{b}{W} < \frac{1}{2}$, the fast reduce algorithm will outperform the standard version. For the recommended polynomials, this is always the case, as b is always five or less, while W is usually 32 or 64.

We can see this in figure 3.4, where fast reduce outperforms standard reduce for every word size, a performance gap that only increases as the word size grows. However, the constraint of $\frac{b}{W} < \frac{1}{2}$ implies that for 8-bit words with tri- or pentanomial $f(x)$, the fast and the standard reduction routines will achieve roughly the same performance. Despite this, we can still observe a difference between them. This is likely due to additional operations performed by the standard version, outside of the `shiftedxor!` operations calculated above. In particular, the standard reduction routine's main loop has a branch that is conditional on the value of a bit, which not only means that it has more logic to process, but also has implications at the hardware level for branch prediction. The fast reduction routine has no such conditional branching.

For each of the standard fields offered by BinaryECC, and for a both a 32- and 64-bit word size, I implemented fast reduction with hardcoded values. Then, as part of the iterative refinement process, I wrote a macro that could produce a fast reduction function for any given word size and binary field. Figure 3.3 shows the Julia code produced by this macro for GF(2^{113}) with reduction polynomial $f(x) = x^{113} + x^9 + 1$. This greatly improves the flexibility of the BinaryECC, as it can now offer fast reduction routines for any word size (for example, an 8-bit version for a micro-controller) and any binary field, allowing fields outside those listed by SECG to be used and still achieve good performance.

An additional benefit of the fast reduction algorithm is that it performs a fixed number of operations for a given reduction polynomial and input size, a property that helps protect against timing attacks.

3.2.3 Multiplication

Shift-and-add For the initial prototype of the binary field component, I wrote a basic right-to-left, shift-and-add implementation. In this version, I used an accumulator to hold the in-

```
julia> @fastreduce(113, 1<<9 + 1)

function reduce(a::BFieldElt{113,0x201,T,L}) where {T,L}
    b::StaticUInt{L,T} = copy(a.value)
    lastword = ceil(Int,113/bitsize(T))
    for i in L:-1:(1+lastword)
        t = StaticUInt{1,T}([b.value[i]])
        shiftedxor!(b, t, bitsize(T)*(i-1)-113)
        shiftedxor!(b, t, bitsize(T)*(i-1)-9)
        shiftedxor!(b, t, bitsize(T)*(i-1))
    end

    extra = 113 % bitsize(T)
    t = StaticUInt{1,T}(b.value[lastword]>>>extra)
    shiftedxor!(b, t, 113)
    shiftedxor!(b, t, 9)
    xor!(b, t)

    b.value[lastword] &= (T(1)<<extra)-1
    return BFieldElt{113,0x201,T,lastword}(changelength(b, lastword))
end
```

Figure 3.3: Julia code for fast binary field reduction, produced by the `@fastreduce` macro for GF(2^{113}) with reduction polynomial $f(x) = x^{113} + x^9 + 1$.

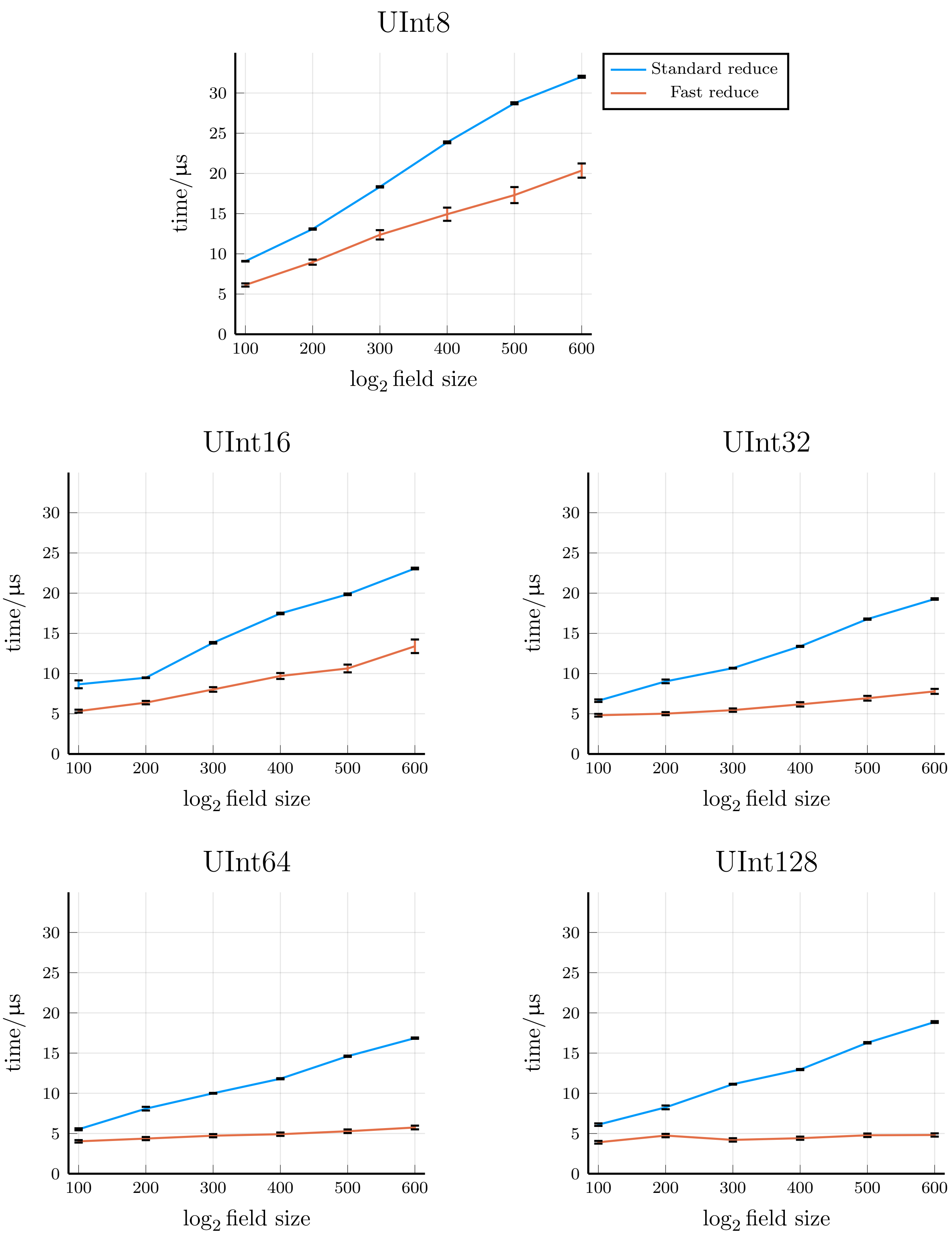


Figure 3.4: Comparison of the performance of binary field reduction methods as field order increases, for a selection of word sizes.

termediate result, which had to be twice as long as the two input polynomials (so that it could accommodate polynomials with degree up to $2m-2$). The main loop repeatedly shifted and added the second argument to this accumulator, depending on the coefficients of the first polynomial. The result was then reduced modulo $f(x)$ by a separate reduction routine.

I then wrote a second version of this method, in which the reduction was interleaved with the multiplication. For a right-to-left method, instead of calculating $b_i(x) = b(x) \cdot x^i$ on each iteration i, I multiplied 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, if it 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 accumulator ($\sum_i a_i b_i(x)$) always remained an element of the binary field GF(2^m), and so no reduction was required at the end. This has the advantage that the accumulator can be half as long as in the first version, but also means that we cannot make use of the fast reduction method discussed in section 3.2.2.

Comb method I also implemented both a left-to-right and a right-to-left version of the comb method, each using 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 $b(x)$. The comb method relies on the fact that shifting by a multiple of the word size is faster than shifting by some arbitrary amount, and so it rearranges the order in which the bits of $b(x)$ are accessed to allow left-shifts that are not aligned with word boundaries to be replaced with ones which are. On average, the shift-and-add method performs $\frac{m-1}{2}$ left-shifts which are (in general) not aligned with the word size, whereas the comb method replaces these with W non-aligned shifts and $\frac{m-1}{2}$ shifts by a multiple of the word size.

However, as we can see in figure 3.5 that for the larger word sizes, the comb method is actually less efficient. For the comb method to be useful, the time savings from replacing $\frac{m-1}{2}$ non-aligned shifts with aligned ones must be enough to offset the cost of the W additional non-aligned shifts that have to take place. For large values of W, these savings are simply not enough. For the smaller word sizes (such as $W = 8$) we can see that although this trade-off does pay off, it is still only for the larger fields where the value $\frac{m-1}{2}$ (and therefore also the potential savings) are large.

Windowing One of the refinements that I tested on the multiplication routines was the windowing. This technique 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, 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 the average cost for any $b(x) \in$ GF(2^m) is $c\frac{m}{2}$. When we instead use width-w windowing, the average cost is $2^w c\frac{w}{2} + c\frac{m}{w}(1 - 2^{-w})$. 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, each with $\frac{w}{2}$ terms on average. The second term comes from the main loop, which will have $\frac{m}{w}$ iterations that each have a $(1-2^{-w})$ probability (assuming that $b(x)$ is chosen uniformly at random) of having to perform one `shiftedxor!`.

If t denotes the cost of multiplying $a(x) \cdot b(x)$, then without windowing we have $p = \frac{m}{2}$, and with window width w we have $t = \frac{m}{2} - 2^w\frac{w}{2} - \frac{m}{w}(1 - 2^{-w})$ (where the constant c has been ignored, because we are only interested in the relative timings). By plotting t against m, as

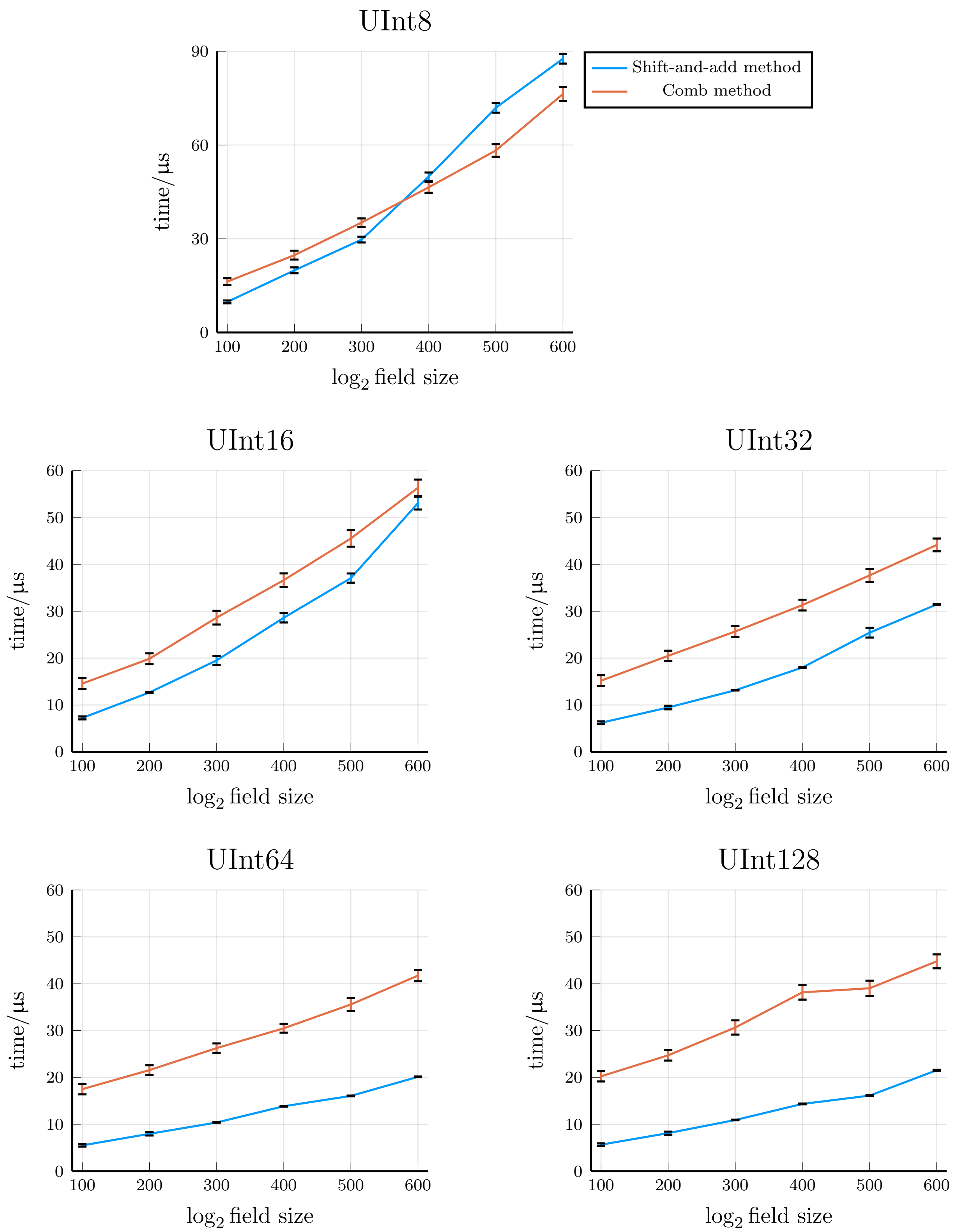


Figure 3.5: Comparison of the shift-and-add and comb methods for binary field multiplication, as field order increases and for a selection of different word sizes.

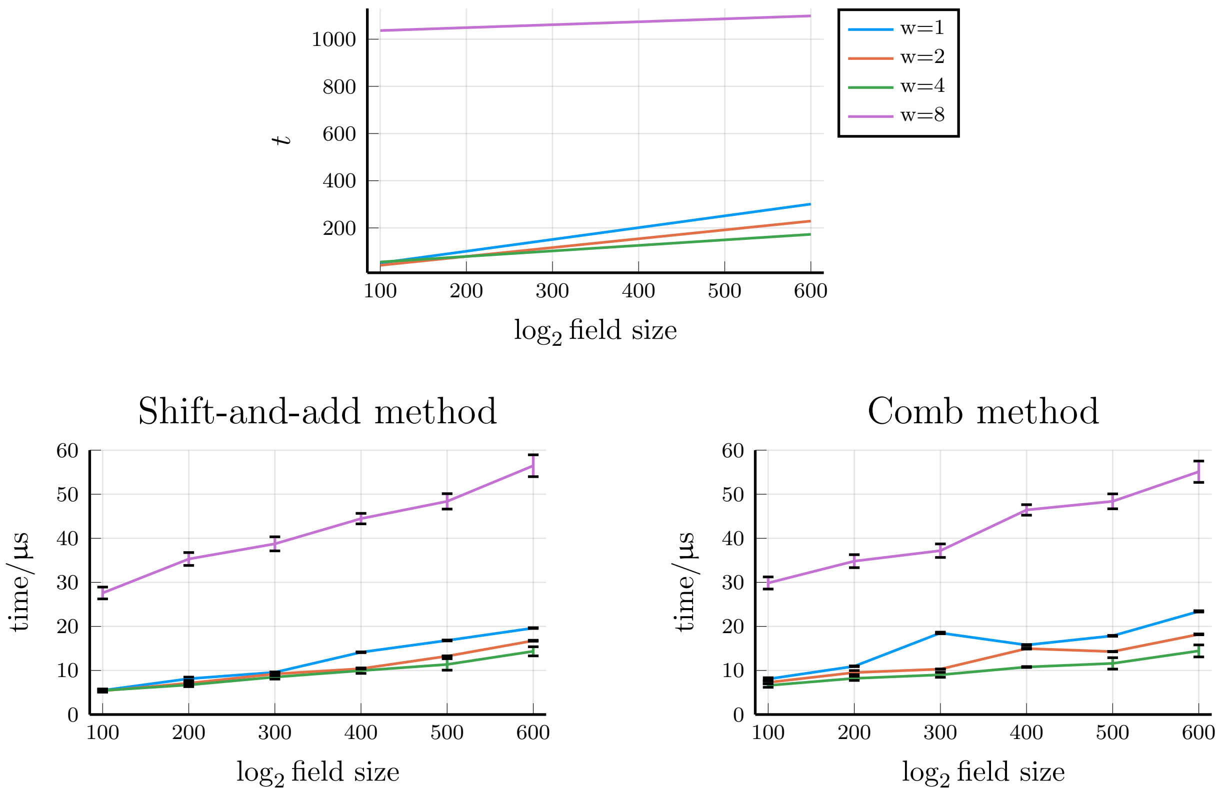


Figure 3.6: Effect of different window sizes on the shift-and-add and comb methods for binary field multiplication. The first figure shows the modelled times, using the formula $t = \frac{m}{2} - w2^{w-1} - \frac{m}{w}(1 - 2^{-w})$, and the second two show the actual timing results.

in figure 3.6 (which also shows timings for the comb method, since we can perform a similar analysis there), we see that $w = 4$ should produce the fastest version of shift-and-add for all the field sizes we are interested in, and $w = 2$ the second best. This is borne out by the actual data from the same figure. We also see, for both the comb and shift-and-adds methods, $w = 8$ worsens performance significantly. From the analysis above, it is clear that this is because the precomputation required is too great, and we can find that it only begins to pay off when m is over a thousand, i.e., well outside the range of field sizes being used for ECC today.

Multithreading From section 2.1.1, we have the fact that fields obey the distributive law, $a(x)\cdot(b_1(x)+b_2(x)) = a(x)\cdot b_1(x)+a(x)\cdot b_2(x)$, and also (from the paragraph on windowing above) that the time taken to multiply two polynomials is proportional to the number of terms in the second one. Furthermore, the two calculations $a(x)\cdot b_1(x)$ and $a(x)\cdot b_2(x)$ are independent of one another. Therefore I augmented the shift-and-add routine to make use of multithreading, by splitting $b(x)$ into $b_1(x) = b_{\lfloor m/2\rfloor}x^{\lfloor m/2\rfloor}+\cdots+b_0x^0$ and $b_2(x) = b_{m-1}x^{m-1}+\cdots+b_{\lfloor m/2\rfloor+1}x^{\lfloor m/2\rfloor+1}$, and then spawning two threads, one given the expression $a(x)\cdot b_1(x)$ to evaluate and the other $a(x)\cdot b_2(x)$. The augmented routine then "fetches" their results (that is, waits until each thread has completed and then retrieves the result of their evaluation), adds them together, reduces their sum modulo $f(x)$, and then finally returns the element of the field $c(x) = a(x)\cdot b(x)$. In figure 3.7, we can see that this thread-level parallelism does not improve the performance. However, for earlier iterations of BinaryECC where the shift-and-add implementation was more costly, this double-threaded approach did indeed improve performance for larger fields, where

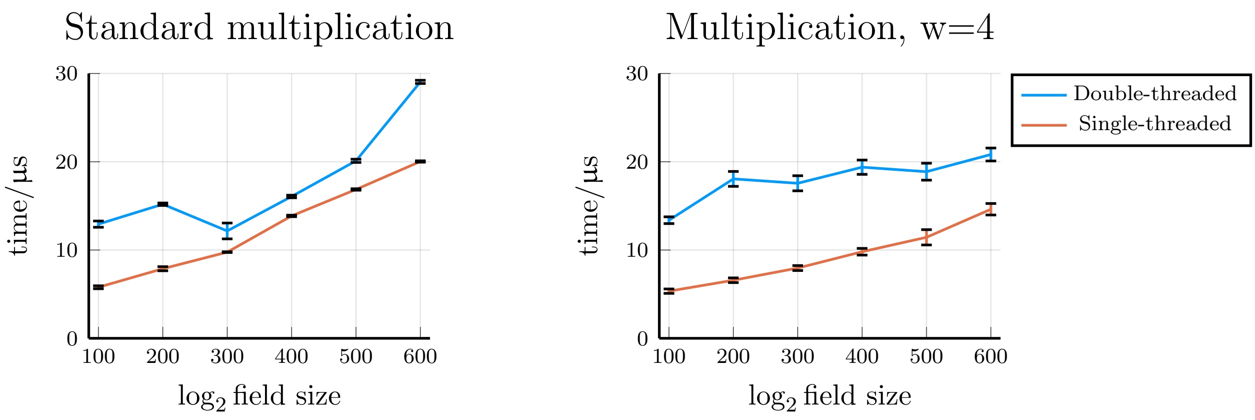


Figure 3.7: Effect of multithreading on the shift-and-add method for binary field multiplication. The left figure compares a double-threaded and single-threaded version of a standard shift-and-add routine, while the right shows the same for windowed shift-and-add routines. All data were collected on a dual-core processor.

each thread had at least 25µs of work. Therefore the lack of improvement in the current version of BinaryECC is likely due to the overheads involved in spawning threads (e.g. copying in values of variables) and fetching their results outweighing their benefits (which could, at best, be a two-times speedup).

However, the standard shift-and-add method is not the fastest (single-threaded) routine; instead, that is shift-and-add with a window size of four. I augmented this in a similar way, but found that there was, again, no improvement for any of the standard fields. This is because, just as had been found before, the double-threaded approach only improves performance when each thread had around 25µs of work. Since the windowed multiplication is faster than that for the full range of fields, multithreading can offer no improvement.

Squaring Because squaring is a linear operation (described in section 2.1.1), it can be implemented much more efficiently than general multiplication. To square an element of the field, the bits of its representation are all spaced out by additional zeros, for example turning "1101" (i.e. x^3+x^2+1) into "1010001" (i.e. x^6+x^4+1). During the package's compilation, a dictionary mapping 4-bit strings to their 7-bit squares is computed, which is then used to square a field element, half a byte at a time. The result is then reduced by the reduction polynomial for the field, just as with general multiplication. This method is much faster than multiplication, because the dictionary does not depend on the arguments and can therefore be built just once.

However, this squaring routine must be explicitly called by the programmer, that is, `square(x)` or `x^2` rather than `x*x`. This is because the cost of explicitly checking inputs for equality every time the general multiplication routine is called outweighs the benefit of using a specialised squaring routine in the rare instance that it is suitable (since the odds of two random elements being equal is very small for cryptographically-sized fields).

Summary In conclusion, I used a single-threaded shift-and-add with a window size of four as the default binary field multiplication method, although the other implementations are also exported by BinaryECC for the sake of completeness. The programmer may also use a specialised squaring routine (with a window size of four), when they know this is suitable.

```
function inv(a::BFieldElt{D,R,T,L}) where {D,R,T,L}
    if iszero(a.value) throw(DivideError()) end

    u = copy(a.value)
    v = StaticUInt{L,T}(R)
    flipbit!(v, D)
    g1 = one(StaticUInt{L,T})
    g2 = zero(StaticUInt{L,T})

    while !isone(u)
        j = bits(u) - bits(v)
        if j<0
            u, v = v, u
            g1, g2 = g2, g1
            j = -j
        end
        shiftedxor!(u, v, j)
        shiftedxor!(g1, g2, j)
    end
    return BFieldElt{D,R,T,L}(g1)
end
```

Figure 3.8: Julia code for binary field inversion.

3.2.4 Inversion

The final operation required for the binary field is inversion, for which I used the polynomial version of the extended Euclidean algorithm as described in section 2.1.1. At the end of the inversion routine's main loop (figure 3.8), two field multiplications occur, implying that inversion will be many times more costly than multiplication. However, these multiplications are only ever by polynomials with a single term (that is, x^j) and so rather than use the general multiplication routines to perform these, I instead directly call `shiftedxor!(u, v, j)` to perform $u(x) := u(x) - v(x) \cdot x^j$. Therefore, although inversion is more costly than multiplication (it is around two to four times slower), the inversion to multiplication ratio is much lower than expected, which has implications for the usefulness of projective coordinates, as will be discussed in section 3.3.2.

3.3 Elliptic curve groups

3.3.1 Representation

Groups

An elliptic curve group is represented by the composite type `EC{B}`, where the parameter `B` is the type of the underlying binary field. Objects of the type `EC{B}` have the fields `a` and `b`, both of type `B`, which are the constants in the simplified equation for a characteristic-2 elliptic curve, $E : y^2 + xy = x^3 + ax^2 + b$.

Unlike binary field elements, where their field is encoded in the type of field elements, I chose to store information about elliptic groups in an object whose type simply indicates which field the curve is defined over. Elements of an elliptic curve group can then contain an `EC{B}` object that provides the group information that is needed for arithmetic. This is summarised in figure 3.9.

Although this prevents the compiler checking that operations only occur on elements of the same group at compile time (necessitating an extra type, `ECMismatchException`, subtyping `Exception`, which can be thrown at runtime), it is more suitable for the way elliptic curve groups are used. While the underlying field is not expected to be changed for a given system, the choice of elliptic curve group may be changed frequently, which (if the elliptic curve group were a part of the type system) would cause many different specialised versions of the same functions to be compiled. 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 type 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{B}` with three concrete subtypes `ECPointAffine{B}`, `ECPointJacobian{B}` and `ECPointLD{B}`. 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{B}` contains three fields: `x` and `y`, both of type `B`, to hold the value of the point, and a field `ec` of type `EC{B}` to hold the information about the group itself. The projective representations, `ECPointJacobian{B}` and `ECPointLD{B}` (Jacobian and López-Dahab respectively), contain `x, y, ec` and one additional field, `z` of type `BFieldElt{B}`. Note that although the projective representations have identical structures, they are given different types to allow different behaviour.

The point at infinity also needs to be represented, but it does not have any coordinates. However, 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 the point $(0, 0)$ (or $(0, 0, 0)$) can be redefined to be $\mathcal{O}$.

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, as outlined in section 2.1.2. However, since these formulae (2.10 for addition and 2.11 for doubling) are for affine coordinates, I derived the corresponding formulae for projective coordinates.

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 coordi-

```
struct ECMismatchException <: Exception end
abstract type AbstractECPoint{B} end

struct EC{B}          struct ECPointAffine{B} <: AbstractECPoint{B}
    a::B                  x::B
    b::B                  y::B
end                       ec::EC{B}
                      end

struct ECPointLD{B} <: AbstractECPoint{B} #ECPointJacobian has an
    x::B                                  #identical definition
    y::B
    z::B
    ec::EC{B}
end
```

Figure 3.9: Julia definitions of the key types associated with elliptic curve groups.

nates $(\frac{X}{Z^2}, \frac{Y}{Z^3})$. 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. 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:

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

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

The formulae for point doubling are:

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

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

Mixed point Using the same method as above, I also derived formulae for addition where the two input points are from different coordinate systems, for example to compute $R_{LD} = P_J + Q_{LD}$, in which Q_{LD} and R_{LD} are López-Dahab points, and P_J is a Jacobian point.

Performance When implementing each of these routines, I attempted to minimise the number of arithmetic operations that were used by storing intermediate results in temporary variables. In the underlying binary field, the three most costly operations are division, multiplication and squaring, with inversion being around three to five times as costly as multiplication (the division to multiplication ratio increases with field size), while squaring only takes around half the time of multiplication.

The idea behind the projective coordinates is that, although their addition and doubling formulae involve more operations overall, they may still be more faster than the affine versions since they contain no division. However, as we can see in table 3.1, the division to multiplication ratio would have to be at least six for doubling in López-Dahab coordinates to be faster than affine coordinates, and even greater for the others, before this trade-off between divisions and multiplications can pay off. As I had calculated the projective formulae for these routines by hand, I searched for alternatives that used fewer multiplications, but I did not find any which reduced the number of multiplications sufficiently for them to outperform the affine point routines. This means that due to the low division to multiplication ratio, the projective coordinate systems supported by BinaryECC do not offer any performance advantage.

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 $P = n \cdot G$ 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.

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

Table 3.1: Cost of point doubling and addition for each representation, measured in the number of binary field divisions, multiplications and squarings required.

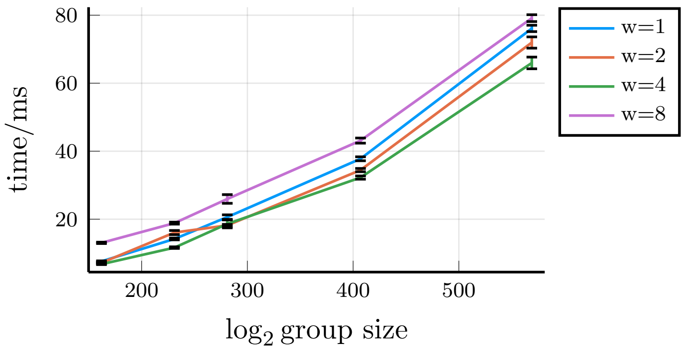


Figure 3.10: Comparison of window sizes when performing scalar multiplication with the double-and-add method.

Double-and-add

For the initial prototype of the elliptic curve group component, I implemented scalar multiplication with the double-and-add method (described in section 2.1.2). Because this method does not require any manipulation of the point's coordinates, I wrote it simply to take an instance of `AbstractECPoint`. The first time any of these functions are executed on a given concrete point type, the compiler will produce a specialised version.

Windowing As with the binary field multiplication, windowing can be applied to scalar point multiplication in order to calculate $k \cdot P$ more efficiently. As we can see in figure 3.10, a window size of $w = 4$ produces a slight performance advantage which becomes more significant as the group order increases. We can also note that, just as with binary field multiplication, the performance drops when $w = 8$, likely due to the precomputation requirement outweighing the reduction in point additions in the main loop. This method performs $2^w A + m(D + \frac{1}{w}(1 - 2^{-w})A)$ doublings and additions for a window size of w (where D is the cost of a doubling, A is the cost of an addition, and the underlying field is $\mathrm{GF}(2^m)$), compared to $m(D + \frac{1}{2}A)$ for the non-windowed version.

Non-adjacent form

The NAF of an integer is the representation that has the fewest non-zero digits of all signed digit representations, as outlined in section 2.1.2. This reduces the number of addition operations required in the main loop, when using a double-and-add style method, which can improve the running time.

```
function mult_bnaf(P::AbstractECPoint{B}, k::Integer) where B
    if k<0 return (-P)*(-k) end
    if iszero(P) return P end

    Q = zero(P)
    while k>0
        if k&1==1
            digit = 2 - (k%4)
            k -= digit
            if digit==1 Q+=P
            else Q-=P
            end
        end
        P = double(P)
        k >>= 1
    end
    return Q
end
```

Figure 3.11: Julia code for elliptic curve multiplication with binary NAF.

Binary NAF The most straightforward version of this representation is binary NAF, which represents the integer k with at most $\lfloor \log_2 k \rfloor + 2$ digits k_i (where $k_i \in \{-1, 0, 1\}$), reducing the average density of non-zero digits to $\frac{1}{3}$ (compared to $\frac{1}{2}$ for a standard binary representation).

Since the NAF of k is calculated in a right-to-left manner, I wrote a scalar multiplication function which interleaves the calculation of the NAF with the point arithmetic itself, rather than computing the NAF upfront and storing it (which could take up twice as much space as k itself). In this algorithm, as shown in figure 3.11, the digits of the NAF of k are computed at the point where they are needed, and not stored before or after that iteration. This method uses $m(D + \frac{1}{3}A)$ point doublings and additions.

Windowing For methods using a NAF, there are two different ways in which windowing can be applied. The first is similar to its usage in the double-and-add method, where each iteration of the main loop is modified to look at w digits of the multiplier's representation, rather than just one. The alternative is to apply the concept of windowing to the NAF itself, allowing the digits of the representation to have a magnitude of up to 2^{w-1}.

From figure 3.12, we can see that all window sizes for the first method (binary NAF) have very similar performance, although, $w = 8$ has a slight advantage. For the second method, we can see in figure 3.13 that a width-six NAF performs best, again by only a small margin.

Multithreading When we calculate $P \cdot k$, the iterations of the main loop corresponding to non-zero digits in k must perform both an point doubling and addition, and so it may be possible to improve performance by calculating these two operations simultaneously with multithreading. For multiplication performed in a left-to-right manner, these operations are `Q := double(Q); Q := Q + P`, whereas for the right-to-left version, it is `Q := Q + P; P := double(P)`. Clearly, we can only apply multithreading to right-to-left multiplication, in which

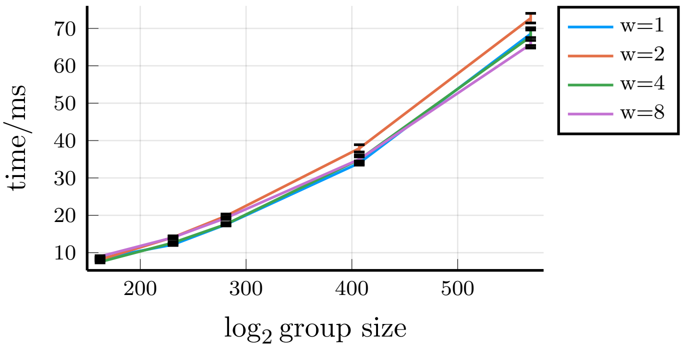


Figure 3.12: Comparison of window sizes when performing scalar multiplication with binary NAF representation.

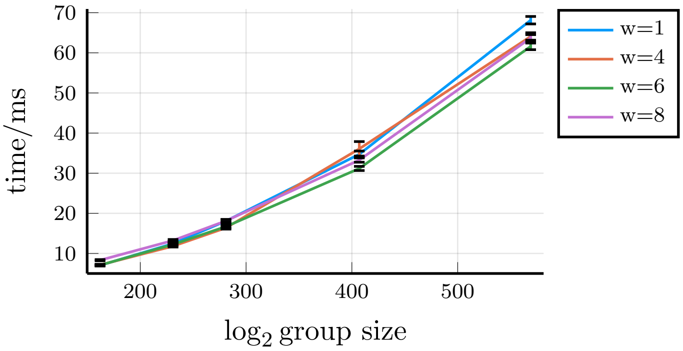


Figure 3.13: Comparison of different widths of NAF for scalar multiplication.

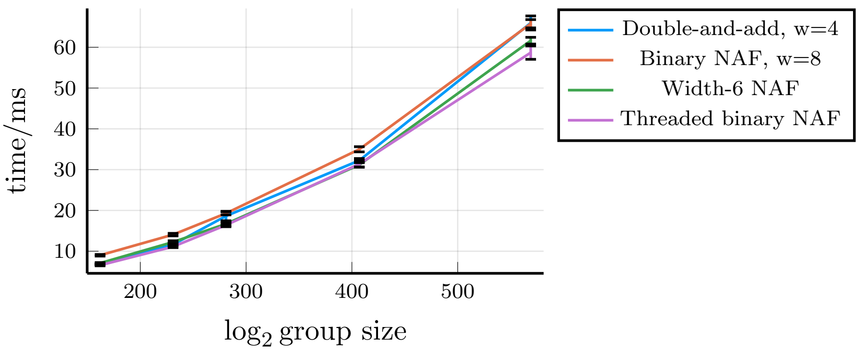


Figure 3.14: Comparison of different methods (each with their optimal parameterisation) for scalar point multiplication.

the second operation does not depend on the result of the first. This does, however, restrict which multiplication routines we can parallelise, because routines which perform precomputation (i.e., the windowed methods) must be left-to-right, otherwise they would need to double every precomputed element (which take the place of P in the operations described above) on every iteration, which would be inefficient. Therefore we must make a choice between windowing and multithreading.

Figure 3.14 compares the best parameterisations of each of the scalar multiplication methods. Here, we can see that the best performance is achieved by a width-six NAF and a double-threaded binary NAF (without windowing), with the double-threaded routine having a slight advantage for larger groups. The default scalar multiplication method in BinaryECC is therefore the double-threaded binary NAF.

Montgomery powering ladder

The Montgomery powering ladder (section 2.1.2) computes $k \cdot P$ by keeping track of two points, P_1 and P_2, as it iterates over the bits of k, always maintaining the invariant $P_2 - P_1 = P$. As this algorithm performs one point doubling and one point addition per iteration, it will take the same amount of time to perform point multiplication for two scalars of the same length, preventing attackers from using the computation time to derive any information about the scalar multiplier. However, this resistance to timing attacks comes at a cost, because there are now m point additions (for a curve with underlying field $\mathrm{GF}(2^m)$) rather than an average of $\frac{m}{2}$.

Optimised version For this reason, I also implemented a version of the Montgomery powering ladder that is specific to affine coordinates, as mentioned in section 2.1.2. This method allows the point $Q = k \cdot P$ to be derived from just the x-coordinates of P_1 and P_2, P, and the fact that $P = P_2 - P_1$. Therefore only the x-coordinates of P_1 and P_2 must be computed in the main loop, saving some arithmetic in the binary field (only two divisions and two squarings are required per iteration, compared to the two divisions, two multiplications and three squarings that would otherwise be used to perform a point doubling and addition). This accelerated version of Montgomery's powering ladder still has a constant running time for scalar multipliers of equal length, and so still offers protection against timing attacks. As we can see from figure 3.15, it is around a third faster than the general version, but still slower than the

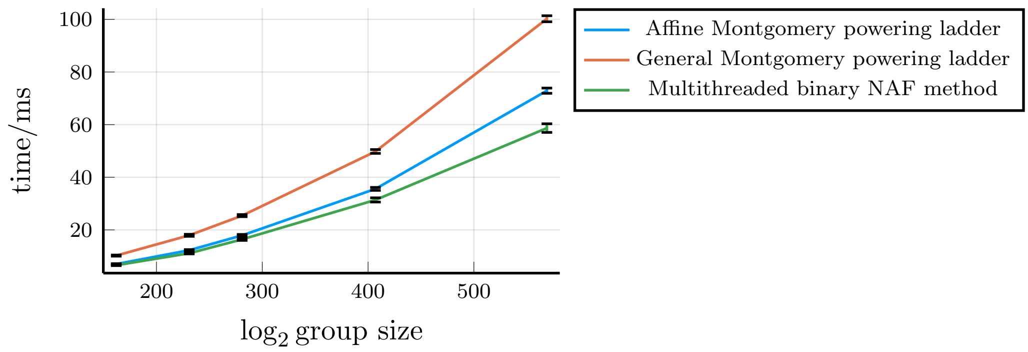


Figure 3.15: Comparison of the general and specialised (affine) version of Montgomery's ladder, alongside the default scalar multiplication routine.

fastest scalar multiplication method, showing that a trade-off must be made between security and performance.

3.4 Cryptographic primitives

BinaryECC contains example implementations of several cryptographic protocols in order to demonstrate how the package can be used. This section discusses these algorithms and the additional structures that I have defined to support them.

3.4.1 Prime fields

Since modular arithmetic is used in several cryptographic protocols, such as ECDSA, I wrote a component for prime field arithmetic. Elements of a prime-order field, GF(p), are held in objects of type `PFieldElt`, which contains `value`, a `BigInt` storing the value of the element, and `p`, the order of the field.

However, the execution time of ECC algorithms using curves defined over binary fields is dominated by binary field arithmetic, with comparatively little time spent on prime field arithmetic (hundreds of operations vs. fewer than ten), meaning that it was more effective to spend time optimising `BFieldElt` arithmetic rather than `PFieldElt`. As a result, I implemented this component in a very straightforward manner, with values stored in Julia's `BigInt` type.

3.4.2 Curve domain parameters

For the Elliptic Curve Domain Parameters, as described in section 2.1.4, I create the type `CurveDomainParams{B}` to hold three key pieces of information: the generating point, `G`; the order, `n`, of `G`; and the cofactor, `h`, of `n`. Although the curve domain parameters are defined to be a septuple in SEC 1 [13], these three values are enough to provide all the information, due to the representations used by BinaryECC. The type parameter `B` provides the binary field (GF(2^m)) and the logarithm of its order (m), while the field point `G` contains the curve parameters `a` and `b`. Therefore it would be redundant to include these elements of the septuple directly in `CurveDomainParams` objects.

Five standard curve domain parameters are provided by BinaryECC (with values taken from SEC 2 [15]), in the form of functions that take a word type and return a `CurveDomainParams` object. For example, calling `SECT163K1(UInt64)` will return the septuple for the curve "sect163k1", represented by arrays of `UInt64`s. This allows custom word types (such as one which logs information to simulate information leaks) to be easily used with the package.

3.4.3 Key pairs

Another type provided by the cryptography layer of the package is `ECKeyPair{B}`, which stores the public (point `Q`) and private (integer `d`) components of an asymmetric key, as used in ECDH and ECDSA. There are also functions for the generation and validation of such keys, following the procedures set out in SEC 1 [13].

Memoisation

The technique of caching previously computed results for later use, under the assumption that the same inputs are likely to be seen again, is known as memoisation. When performing scalar multiplication $P \cdot k$ with a right-to-left method, the bulk of the calculation is to repeatedly double P, computing $2^i \cdot P$ for every $i \leq \log_2 k$. For another arbitrary point Q, this work cannot be reused unless Q's chain of doubles happens to coincide with P's chain at some point. When performing arbitrary arithmetic, elliptic curve groups are large enough that this is unlikely to be the case.

However, in the context of cryptography, it is common to perform scalar multiplication with the same input point (the generating point, G, from the curve domain parameter septuple). For example, key pair generation must calculate $G \cdot k$ (where $k \in_R \mathbb{Z}_n^*$ and n is the order of G, also provided in the septuple), and this is a subroutine used in both ECDSA and ECDH. Therefore memoisation can be used to store the chain of doubles of G, allowing the bulk of work in generating a key pair to be reduced to a simple dictionary lookup and greatly decreasing the time taken to perform the multiplication. It also reduces the time spent in garbage collection, since an ordinary doubling operation creates and disposes of several field elements as part of its calculation, and scalar multiplication without memoisation performs hundreds of point doublings.

Memoisation cannot be applied as effectively to left-to-right methods, since the point being doubled in that case is the accumulator, which must depend on k. On the i^{th} iteration, the point being doubled is $\{k_{\log_2 k} \ldots k_{(\log_2 k)-i+1}\} \cdot G$ (where $\{k_{a+b} \ldots k_a\}$ is the number represented by the a^{th} to $(a+b)^{\text{th}}$ digits of k). This will only be in the dictionary if a multiplication has previously been performed where the scalar multiplier has the same i most significant bits, which is very unlikely for a random k, for all but the smallest values of i. Therefore the windowed multiplication methods, which must use a left-to-right method (discussed in section 3.3.3) do not see any significant improvements from memoisation.

Despite being unable to combine memoisation with windowing, it still outperforms all other scalar multiplication methods when the input point has been seen before. However, using it with an arbitrary point will unnecessarily bloat the dictionary by filling it with results that are unlikely to be needed again. Instead, the default scalar multiplication in BinaryECC does not use memoisation and it is left to the programmer to identify cases where they are multiplying by a common point and explicitly call the memoised scalar multiplication function.

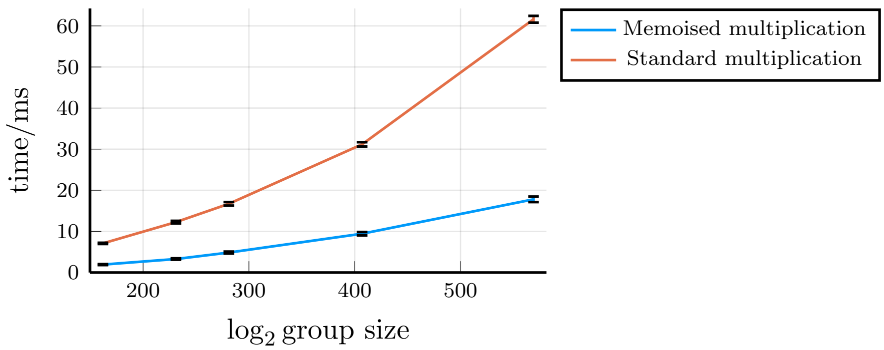


Figure 3.16: Comparison of the fastest memoised scalar multiplication routine (i.e. using binary NAF) with the fastest standard multiplication method (width-6 NAF).

3.4.4 Cryptographic protocols

The two example protocols implemented by BinaryECC are ECDSA and ECDH (sections 2.1.4 and 2.1.4 respectively), following the procedures in SEC 1. Where hashing algorithms are required, the function `sha256` from the Julia package `SHA` [21] is used instead of custom implementation, since this package is designed to focus on ECC. For ECDSA, the type `ECDSASignature` was created to allow signatures to be easily handled for generation and verification.

3.5 Testing

Unit tests, consisting of both test vectors (for elliptic curve arithmetic [14] and ECDSA [17]) and algebraic identities (such as $\frac{x}{x} = 1_F$), formed a continuous part of the development process. This enabled the quick detection of bugs in the code, which was especially important as routines were incrementally refined.

3.6 Documentation

To improve the usability of the package, I have written documentation on the exported functions and types, using the Julia package `Documenter` [25], hosted alongside the BinaryECC source code on GitHub.

3.7 Summary

This package has, at its core, three layers. At the bottom is the binary finite field implementation, with associated type `BFieldElt`, on which the elliptic curve layer (offering several types, primarily `ECPointAffine`) depends. The top layer then uses this elliptic curve arithmetic to offer several cryptographic functions and types, such as `CurveDomainParams` and ECDSA signatures. During the process of implementation, I have strived to consider not only performance, but also usability, by offering standard values and routines for quick use as well as ways to define custom fields and groups, and alternative routines with more timing attack resistance.

4 Evaluation

4.1 Success criteria

In the original project proposal, I listed the following features which the package must offer in order to be considered a success, all of which have since been met by BinaryECC:

- *Binary finite field arithmetic*
  Binary field arithmetic, including addition, subtraction, multiplication, inversion, division, reduction, exponentiation and square roots have been implemented as part of BinaryECC. I have created a type for elements, `BFieldElt{D,R,T,L}`, which encodes the field information (`D` and `R`) and representation (`T` and `L`), along with helper functions to automatically choose a suitable representation for custom fields, and predefined nine standard fields from SEC 2 [15].

- *Several algorithms for elliptic curve arithmetic:*

  – *An implementation using affine coordinates*
    The package allows affine points to be added together, doubled, and multiplied by a scalar value. There are several other utility functions which allow points to be created from their coordinates, selected uniformly at random from a given group, verified to be element of a given group, checked for equality and whether they are the identity, and printed in a human-readable format.

  – *Implementations for several types of projective coordinates*
    Both Jacobian and López-Dahab projective coordinates are provided by the package, also with their own addition and doubling routines, and sharing the scalar multiplication routines (as these were written for generic elliptic curve points rather than any specific representation). They also have the same utility functions as affine points, in addition to mixed representation implementations of addition, and conversion routines to allow points to be mapped to their equivalents in each representation.

  – *An implementation which is resistant to timing attacks*
    Two scalar multiplication methods are offered which provide resistance to timing attacks. The general version of Montgomery's powering ladder can be applied to any point representation, while there is also a specialised affine version that has higher performance.

- *Standard curve parameters*
  Custom curves
  BinaryECC offers a type for elliptic curve groups, called `EC`, which allows new groups to be defined with just the a and b parameters from the curve equation. It also provides the `CurveDomainParams` type, which represents the elliptic curve domain parameter septuple defined by SECG [6], with eleven different standard curve parameters from SEC 2 [15] predefined. It also offers functions to create and validate custom curve domain parameters.

- *Functions to perform ECDSA*
  Functions to perform ECDH
  There is a type, `ECKeyPair`, to store asymmetric elliptic curve keys, although with a routine to generate keys uniformly at random and to validate them. ECDH is implemented through a collection of key agreement and calculation routines. ECDSA signatures are implemented as objects of the type `ECDSASignature`, with routines to sign messages and verify signatures.

- *The ability to parse high level expressions involving elliptic curve arithmetic and produce implementations of them*
  The syntax used by BinaryECC is as close to mathematical notation as possible, so that users do not need to familiarise themselves with new operation names. For example, the expression $(P + 2Q) \cdot d$ can be written as `(P + 2Q) · d` or as `(P + 2Q)*d`. The points involved in such expressions may be different representations (e.g., `P` is Jacobian and `Q` is López-Dahab) without needing to be explicitly converted.

4.2 Evaluation of components

In this section I review the three main components of the project, testing each one and evaluating their performance against pre-existing tools where appropriate.

4.2.1 Binary fields

This component provides a high performance implementation of binary field arithmetic, which was achieved by analysing and comparing the costs associated with different possible representations of field elements, as well as how best to perform key operations on those elements. It contains eight implementations of multiplication and two for squaring (using techniques such as windowing, multithreading, interleaved reduction, etc., as well as combinations of those ideas), with their relative performances analysed in section 3.2.3:

`mult_comb_rtl`	`mult_comb_ltr`	`mult_comb_window`
`mult_shiftandadd`	`mult_shiftandadd_window`	`mult_ownreduce`
`mult_threaded`	`mult_threaded_window`	
`square_standard`	`square_window`	

There are also specialised reduction routines for each of the standard fields and a macro to produce similarly specialised routines for custom fields.

The arithmetic in this component is indirectly verified by the test vectors used for the elliptic curve group arithmetic, since that component depends upon this one. To verify the field arithmetic for the edge cases, I wrote sets of tests checking results such as $\frac{x}{x} = 1$, $x \cdot y = y \cdot x$ and $x + x = 0$ for random elements. These checks were performed for every arithmetic implementation (not just the ones which were chosen to be the default), for every standard field and for five different word sizes (8-bit to 128-bit), with all 155 tests passing without errors.

To evaluate the performance of the arithmetic, I compared it to two pre-existing open-source Julia packages, GaloisFields.jl [7] and Nemo.jl [8]. Inversion, multiplication and squaring are the three most costly operations for binary field arithmetic (since addition is simply exclusive-or), meaning they have the greatest impact on the overall performance of a package. As can be seen

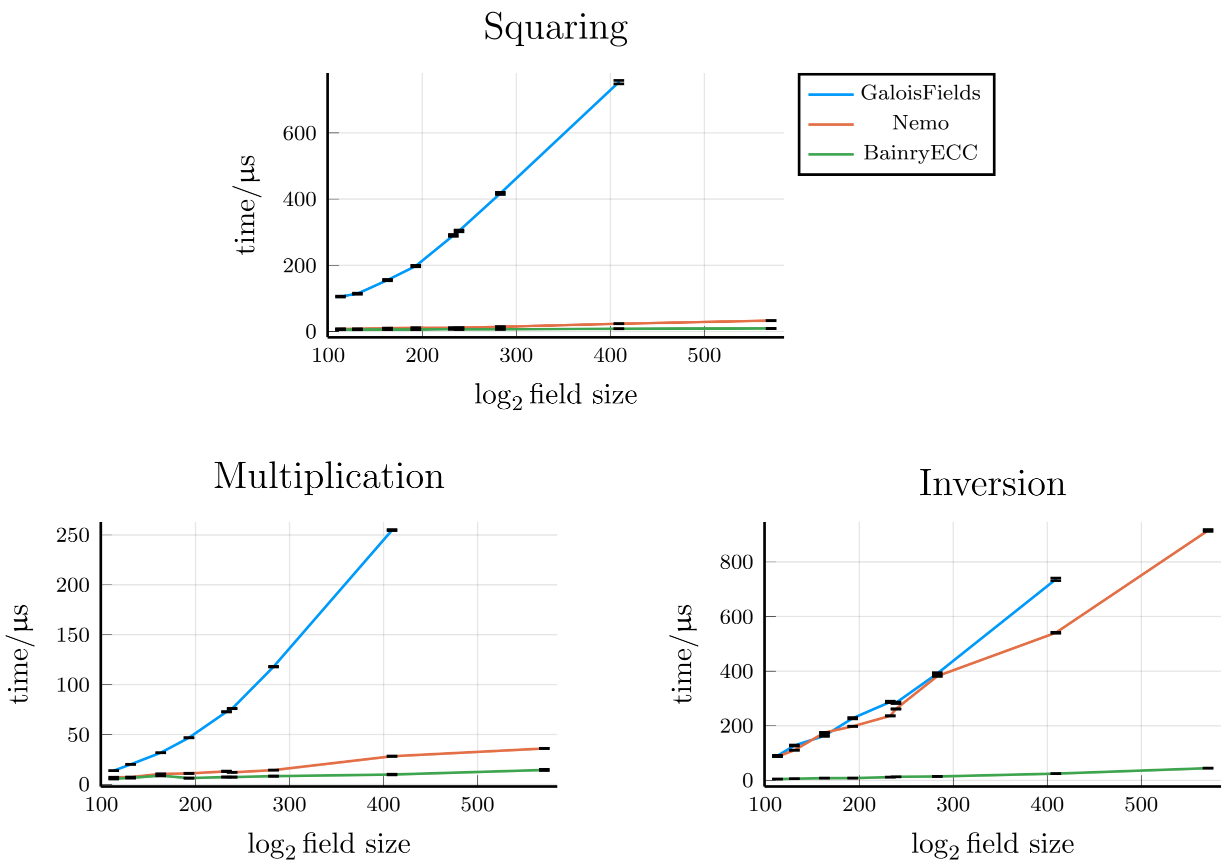


Figure 4.1: Comparison of timings for binary field operations, using the packages BinaryECC, Nemo and GaloisFields (for a smaller range of field sizes, as it does not support the full range of standard fields in BinaryECC).

in figure 4.1, BinaryECC achieves higher performance than both GaloisFields and Nemo. For GaloisFields (which implements the arithmetic directly in Julia, just as BinaryECC does), this performance gap is very significant and the rate of growth for execution times is much higher than for BinaryECC. My implementation also outperforms Nemo, a Julia wrapper around the C library Flint [9], albeit to a lesser extent for multiplication and squaring.

4.2.2 Elliptic curve groups

For this component, I used test vectors [14] containing scalar multiplication results for around 50 different multipliers, for each standard elliptic curve group. Additionally, I wrote tests to verify the results of various edge cases (such as $G \cdot n = \mathcal{O}$ for generating point G of order n), all three point representations, and for each alternative scalar multiplication and point doubling routine. The routines are as follows:

`mult_standard_rtl`	`mult_standard_ltr`	`mult_window`
`mult_bnaf`	`mult_bnaf_threaded`	`mult_bnaf_window`
`mult_wnaf`	`mult_memo`	
`mult_mont_general`	`mult_mont_affine`	

In total, I produced 89 tests that comprehensively covered the functionality in this component, all of which passed without error.

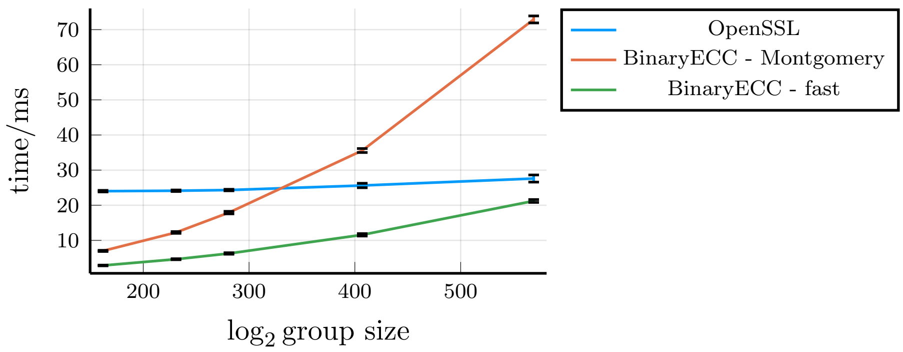


Figure 4.2: Comparison of key generation timings with OpenSSL, for both the high performance and timing attack resistant methods offered by BinaryECC.

At the time of writing, I am not aware of any other Julia packages which support elliptic curves defined over binary fields, and so I am unable to evaluate the performance of BinaryECC against pre-existing implementations.

4.2.3 Elliptic curve cryptography

This component builds upon the previous two by offering several primitives for ECC, such as ECDSA and key pair generation routines. To verify the correctness of the ECDSA implementation, I used a set of test vectors [17] with an optional parameter in the signing routine (which allowed the integer $k \in \mathbb{Z}_n$, used in the ephemeral key pair, to be passed in to the function rather than chosen uniformly at random). For ECDH, I wrote a set of integration tests to check that two entities executing these routines would generate the same result (the shared private key), and that the process produces a different shared key each time. Key pair generation is mostly scalar point multiplication, verified in section 4.2.2, and so I simply wrote tests to check that a different key pair is generated each time. For the prime field arithmetic, I tested basic arithmetic rules held, such as $x + p \equiv x$ for GF(p). In total, there were 107 tests, all of which passed without error.

As noted in section 4.2.2, there are no Julia packages which support elliptic curves over binary fields, and therefore also none which support ECC with binary fields. Instead, I evaluated this component against the performance of OpenSSL [10] (written primarily in C), as called from within Julia. Figure 4.2 shows that while my high performance scalar multiplication implementation is faster than OpenSSL for generating key pairs, for scalar multiplication with timing attack resistance (using the affine version of Montgomery's powering ladder) my implementation is only faster for the smaller field sizes. For ECC however, a 256-bit key is thought to provide protection until the year 2040 [13], which is well within the range where BinaryECC's implementation of a timing attack resistant method outperforms OpenSSL, meaning that it can still be useful for applications where two decades of protection is acceptable.

5 Conclusions

In this dissertation I explored both mathematical ideas, such as NAF and projective coordinates, as well as programming techniques, such as memoisation and windowing, to create a high performance implementation of ECC. A major aim of this project was to fill in a gap in the Julia ecosystem by producing a package for ECC over binary fields, which was successfully achieved by analysing the performance of different representations and implementations of key data types and arithmetic operations. The final package, BinaryECC, outperforms pre-existing Julia packages where there is an overlap in functionality (that is, in the binary field component) as well as achieving comparable, and for many key sizes better, execution times to using OpenSSL from within Julia.

5.1 Lessons learnt

Over the course of this project, I have learnt a great deal about both the mathematics behind ECC and also the more practical problems involved in its implementation, as well as the language Julia which was new to me when I began.

I have also explored some promising techniques that did not deliver speed improvements and in particular, I found that these tended to be the techniques that relied upon the underlying components having specific characteristics. For example, projective coordinates aim to reduce point addition and multiplication times by replacing each inversion in the binary field with several multiplications. However, this assumes that the inversion:multiplication ratio of the underlying field is actually fairly high; this is an assumption that tends to hold true for prime order fields, but not so much for binary fields (BinaryECC has a ratio of around two to four, depending on field size).

5.2 Future extensions

While this dissertation focussed on ECC, there is significant overlap with the area of Pairing-Based Cryptography (PBC), since two of the currently known efficiently computable pairings make use of elliptic curves [26]. In the future, BinaryECC could be extended to also support PBC, allowing identity-based or attribute-based encryption schemes to be implemented. This could be achieved by adding a data type and point addition / doubling routines for supersingular curves, alongside an implementation of a bilinear pairing.

Another possible area for development is side-channel attacks (outside of the timing attacks already considered in this project) such as attacks based on power or fault analysis [11]. This could make use of the flexibility in the representation of binary field elements by using a custom word type with logging capabilities to simulate the leakage of information.

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

[6] SECG, "Standards for Efficient Cryptography Group." Available: `https://www.secg.org/`.

[7] T. Kluck, "GaloisFields.jl." Available: `https://github.com/tkluck/GaloisFields.jl`, 2018.

[8] C. Fieker, W. Hart, T. Hofmann, and F. Johansson, "Nemo/Hecke: Computer algebra and number theory packages for the Julia programming language," in *Proceedings of the 2017 ACM on International Symposium on Symbolic and Algebraic Computation*, ISSAC '17, (New York, NY, USA), pp. 157–164, ACM, 2017.

[9] W. B. Hart, "Fast Library for Number Theory: An introduction," in *Proceedings of the Third International Congress on Mathematical Software*, ICMS'10, (Berlin, Heidelberg), pp. 88–91, Springer-Verlag, 2010. `http://flintlib.org`.

[10] OpenSSL Software Foundation, "Openssl: Cryptography and SSL/TLS toolkit." Available: `https://www.openssl.org/`, 2018.

[11] J. Fan and I. Verbauwhede, *An Updated Survey on Secure ECC Implementations: Attacks, Countermeasures and Cost*, pp. 265–282. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012.

[12] J. López and R. Dahab, "Fast multiplication on elliptic curves over gf(2m) without precomputation," in *Cryptographic Hardware and Embedded Systems* (Ç. K. Koç and C. Paar, eds.), (Berlin, Heidelberg), pp. 316–327, Springer Berlin Heidelberg, 1999.

[13] D. R. L. Brown, "SEC 1: Elliptic curve cryptography," 2009. Available: `https://www.secg.org/sec1-v2.pdf`.

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

[15] D. R. L. Brown, "SEC 2: Recommended elliptic curve domain parameters," 2010. Available: https://www.secg.org/sec2-v2.pdf.

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

[17] T. Pornin, "Deterministic usage of the Digital Signature Algorithm (DSA) and Elliptic Curve Digital Signature Algorithm (ECDSA)," RFC 6979, RFC Editor, 08 2013.

[18] P. Brachet, "TeXMaker." Available: https://www.xm1math.net/texmaker/, 2003.

[19] S. Pfitzner, M. Innes, S. Kadowaki, *et al.*, "Juno IDE." Available: https://junolab.org/, 2003.

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

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

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

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

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

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

[26] A. Menezes, "An introduction to pairing-based cryptography," 2005.

[27] S. D. Galbraith and P. Gaudry, "Recent progress on the elliptic curve discrete logarithm problem," *Des. Codes Cryptography*, vol. 78, p. 51–72, Jan. 2016.

[28] J. Bezanson, S. Karpinski, V. B. Shah, *et al.*, "Julia 1.5 documentation." Available: https://docs.julialang.org/en/v1/, 2020.

[29] J. Bethencourt, A. Sahai, and B. Waters, "Ciphertext-policy attribute-based encryption," in *Proceedings of the 2007 IEEE Symposium on Security and Privacy*, SP '07, (USA), p. 321–334, IEEE Computer Society, 2007.

[30] Python Software Foundation, "Welcome to Python.org." Available: https://www.python.org/, 2020.

[31] D. Suo, "Crypto.jl." Available: https://github.com/danielsuo/Crypto.jl, 2014.

[32] S. Castano, "Elliptic Curve Cryptography in Julia." Available: https://gitlab.com/braneproject/ECC.jl, 2019.

A Project Proposal

Phase 3 Proposal: Optimising Elliptic Curve Cryptography over Binary Finite Fields in Julia

Molly Fryatt

22 October 2020

A.1 Introduction and Background

Many cryptographic functions rely on the assumption that it is computationally difficult to solve the Discrete Logarithm Problem (DLP); that is, to find k, given $g, g^k \in \mathbb{F}_p$. For example, the security of Diffie-Hellman (DH) key exchange is based on the difficulty of the Computational Diffie-Hellman problem, for which the only known solution involves computing a discrete logarithm (Galbraith 2015, [27]). The Elliptic Curve Discrete Log Problem (ECDLP), where the aim is to find k, given $P, kP \in E(\mathbb{F}_q)$, is the analogue of DLP for elliptic curve groups, giving rise to Elliptic Curve Diffie-Hellman (ECDH) key exchange. It is believed that ECDLP is harder than DLP (Silverman 1998, [4]), allowing shorter keys to be used with ECDH compared to DH.

I plan to create a Julia [1] library for Elliptic Curve Cryptography (ECC) which offers various curves and cryptographic primitives, as well as allowing users some freedom to create custom curves and applications. The library will use elliptic curves over binary fields. Currently, Julia's standard library [28] does not provide functions or types for ECC, and I have not found any other open-source Julia libraries which offer this functionality either.

A.2 Starting Point

I am using Julia because it has a dynamic type system while still offering high performance, as well as providing features, such as macros, that will help me implement various parts of my project's functionality. However, my experience with Julia is minimal, as I only began to learn it a month before this proposal was submitted.

As part of an internship (August to September 2020), I implemented a Pairing Based Cryptography scheme, called Ciphertext-Policy Attribute-Based Encryption [29], in Python [30]. This involved implementing arithmetic for a specific elliptic curve defined over a prime field. In addition to that, I have been reading a textbook (Guide to Elliptic Curve Cryptography, [3]) which covers elliptic curve arithmetic, binary and prime field arithmetic, and key cryptographic protocols.

I plan to create my project from scratch, rather than extending an existing implementation. During my research for this proposal, I found several open-source Julia packages that are in the this area, although neither of them provide quite the same functionality as I intend to create. One such package, by Daniel Suo [31], is designed to wrap OpenSSL [10], and it provides functions for ECDSA and arithmetic for finite polynomial fields. However, it is not optimised for binary fields in particular and it is not ready for production. Another package, by Simon Castano [32], provides functions for public key cryptography using a specific curve which is defined over a prime field ("secp256k1", [15]) rather than a binary field, and so it is less closely related to my project.

A.3 Substance and Structure

The core of my project will involve implementing arithmetic in GF(2^n) and elliptic curve arithmetic. My initial implementation will represent points on the curve with affine coordinates, and I will then expand this to allow other point representations (which should help find performance improvements, for example, by removing the need for costly field inversions). It will also offer an implementation of point arithmetic which is resistant to timing attacks. The core of the library should also offer users a set of standard curve parameters, such as those listed by the Standards for Efficient Cryptography Group (SECG) [15]. It should also allow them to define their own custom curves and perform basic validation, for example by ensuring that the generator point is on the given curve and of the specified order.

I will also implement Elliptic Curve Digital Signature Algorithm (ECDSA) and ECDH primitives as part of the core, and allow users to define their own cryptographic applications using high level expressions (which the library should parse and implement, ideally at compile time, e.g. with macros). The library should aim to find an efficient implementation of each expression, inserting conversions between different point representations where this would improve performance.

A possible extension for the library is to implement a point counting algorithm, which would allow users to create and use curves which they don't already know the order of. Another is to implement a pairing algorithm, allowing the library to also be used for Pairing Based Cryptography.

A.4 Success Criteria

In order to be considered a success, the library should offer the core features listed in section 3, i.e.:

- Binary finite field arithmetic
- Several algorithms for elliptic curve arithmetic:
  - An implementation using affine coordinates
  - Implementations for several different types of projective coordinates
  - An implementation which is resistant to timing attacks
- Standard curve parameters
- Custom curves
- Functions to perform ECDSA
- Functions to perform ECDH
- The ability to parse high level expressions involving elliptic curve point arithmetic and produce implementations of them

In addition to this, I plan to evaluate the library based on its performance, as measured against existing open source implementations in C / C++. I plan to find the run time of each of the functions that I write and compare those with the run times of corresponding functions in existing implementations.

A.5 Work Plan

I have split the time between the proposal submission deadline (23/10/2020) and the dissertation submission deadline (14/05/2021) into fifteen work packets:

1. **23/10 – 05/11**
   - 23/10: Proposal deadline
   - Prep work: research algorithms for the cryptographic primitives, research different point representations and how to perform point arithmetic with them
2. **06/11 – 19/11**
   - Prep work: research algorithms for point arithmetic that are resistant to timing attacks
3. **20/11 – 03/12**
   - Implement binary field arithmetic
   - Implement EC arithmetic (affine coordinates)
   - Implement standard and custom curve parameters
4. **04/12 – 17/12**
   - Implement point arithmetic for different representations
   - Implement timing attack resistant algorithm(s)
   - Evaluate performance of each representation on different types of operation (e.g. doubling and just adding two different points)
5. **18/12 – 24/12**
   - Automatically parse and evaluate high level expressions
   - Optimise that algorithm to use an efficient combination of point representations and conversions
6. **25/12 – 07/01**
   - Implement ECDSA
   - Implement ECDH
7. **08/01 – 21/01**
   - Measure the performance of core features against existing open-source implementations

8. **22/01 – 04/02**
   - Extension: implement pairing
   - Write progress report and prepare presentation
9. **05/02 – 18/02**
   - 05/02: Progress report deadline
   - Extension: research and implement point counting algorithm
   - 11/02: Progress report presentations
10. **19/02 – 04/03**
    - 19/02: "How to write a dissertation" lecture
    - Evaluate extension(s) to project
    - Small buffer to help keep the plan on track
11. **05/03 – 18/03**
    - Write draft Introduction and Preparation chapters for dissertation
    - Send draft chapters to supervisor
12. **19/03 – 01/04**
    - Write draft Implementation chapter
    - Send draft chapter to supervisor
    - Incorporate feedback on Introduction and Preparation chapters
13. **02/04 – 15/04**
    - Write draft Evaluation and Conclusion chapters
    - Send draft chapters to supervisor
    - Incorporate feedback on Implementation chapter
14. **16/04 – 29/04**
    - Incorporate feedback on Evaluation and Conclusion chapters
    - Make final revisions / checks of whole dissertation
15. **30/04 – 14/05**
    - Buffer week
    - 14/05: Dissertation deadline

A.6 Resource Declaration

I plan to use my own computer (2.5GHz CPU, 8GB RAM, 100GB SSD, 900GB HDD, Windows 10 OS). In the event that it fails, I would use MCS computers until I am able to replace/repair my machine. I will use GitHub for version control and backup of my code and dissertation, as well as backing them up in my filespace on the University OneDrive.

I accept full responsibility for this machine and I have made contingency plans to protect myself against hardware and/or software failure.