Molly Fryatt

Optimising Elliptic Curve Cryptography over Binary Finite Fields in Julia

Computer Science Tripos – Part II

St John's College

April 26, 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 [date]

Proforma

Candidate Number:	**2407G**
Project Title:	**Optimising Elliptic Curve Cryptography over Binary Finite Fields in Julia**
Examination:	**Computer Science Tripos – Part II, July 2021**
Dissertation Word Count:	**10,977**[1]
Software Line Count:	**TODO**
Project Originator:	Dr Markus Kuhn
Project Supervisor:	Dr Markus Kuhn

Original aims of the project

Work completed

Special difficulties

None.

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

Contents

List of Figures

Chapter 1

Introduction

This dissertation describes the development of an elliptic curve cryptography and binary fields package for Julia [1], a high performance language launched in 2012. My main aim for this package, called BinaryECC, was to explore the different algorithms available for each of the key operations, comparing their performance on varying sizes and representations of curves. In this chapter, I discuss the motivation behind elliptic curve cryptography (ECC) and give an overview of the pre-existing Julia packages relevant to this area.

1.1 Motivation

Many cryptographic protocols, such as Diffie-Hellman key exchange ([2], 1976), depend upon the assumption that is computationally difficult to solve the Discrete Logarithm Problem (DLP). This is the problem of finding x, given $y = g^x \in \mathbb{G}$ (where g and $\mathbb{G}$ are publicly known and fixed within the system). There are several general-purpose algorithms for solving this problem in any suitable 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 subexponential 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 ("Guide To Elliptic Curve Cryptography", [3]).

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

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

1.2 Related work

This project tackles the problem of producing a Julia package for elliptic curves over binary fields which allows the user a large amount of flexibility (by allowing custom curves and fields to be used and the 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.

The package "GaloisFields.jl", by Kluck 2018 [5], adds support for Galois fields to Julia. 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. Julia also has a computer algebra package, "Nemo.jl" [6], which 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 [7]. Currently, Julia does not currently have any packages which support elliptic curves over binary fields. However, OpenSSL [8] is a general-purpose cryptography library, written primarily in C, which supports elliptic curves defined over both prime and binary fields.

1.3 Challenges

Undertaking this project presented me with two key challenges: firstly, I had decided to use Julia, a language that was new to me and which only released a stable version in 2018; and secondly, much of the mathematics required to understand elliptic curve cryptography is beyond the scope of material taught in the Tripos. Due to these challenges, and the constraint of this being a one-year project, I decided to focus more on implementing an efficient package than on security aspects such as resistance to side channel attacks (e.g. power analysis and fault attacks, as discussed in [9]), although some consideration is given to timing attack resistance.

1.4 Results

I have successfully built a package that can perform arithmetic in elliptic curve groups, offering several different elliptic curve point representations and alternative algorithms the key operation (scalar multiplication), some 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 representation of binary field elements which can be customised (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 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.

Chapter 2

Preparation

2.1 Background

Elliptic curve cryptography uses elliptic curve groups, which are formed from the set of points in a field that are on a given curve. In cryptography, this field is either a binary finite field, GF(2^m), or a prime order field, GF(p), and it is the first of these two which I focus on in this project. 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

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

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

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

The neutral elements of this field are $0_F = 0$ and $1_F = 1$. Such fields also have a reduction polynomial, which is an irreducible binary polynomial of degree m 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). For example:

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

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

Multiplication

Similarly, multiplication of binary field elements, $a \cdot b$ is performed as multiplication of the two binary polynomials. However, this may produce a polynomial of degree greater than $m-1$ (at most, it will be degree $2m-2$), which is not an element of GF(2^m). As a result, the product of the polynomials must be reduced modulo $f(x)$ to produce the corresponding field element.

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

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

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

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

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

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

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

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

Reduction

To convert an arbitrary binary polynomial into an element of the binary field GF(2^m), we need to reduce it modulo $f(x)$, where $f(x)$ is the reduction polynomial for that field. Although fields of the same order are isomorphic to one another, the performance of binary fields depends on the choice of $f(x)$. Standards organisations, such as NIST [10] and SECG [11], provide recommended reduction polynomials for commonly used field orders.

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

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

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

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

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

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 $W(n-1) \leq m' < Wn$, 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 GF(2^{163}) much more efficiently than with the reduction standard method.

Inversion

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

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

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

2.1.2 Elliptic curve groups

Elliptic curves are defined by the Weierstrass equation:

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

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

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

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

For 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 $P_3 = (x_3, y_3)$ where

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

Figure 2.1: Illustration of the cord 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.

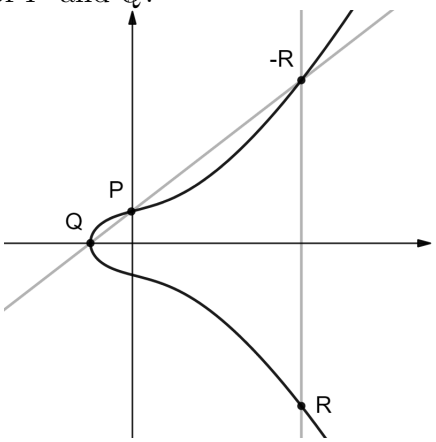
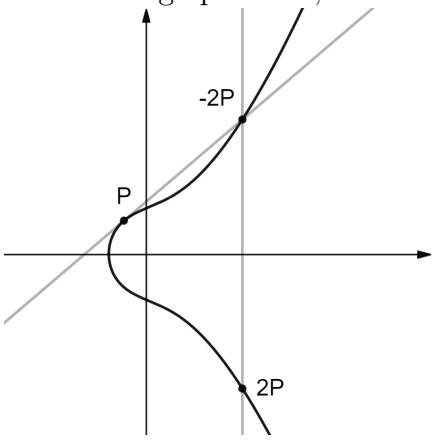


Figure 2.2: Illustration of the tangent rule, used to double the point P. First, a tangent to the curve is drawn at point P. This line intersects the curve again at $-2P$, and so we reflect about the x-axis to find the result of the doubling operation, $2P$.



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

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

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

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

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

Scalar multiplication

Scalar multiplication, $P \cdot k$, is the addition of the curve point P to itself, k times. For cryptographic values of k (that is, where $\log_2 k \approx 100$), this naive approach is computationally infeasible.

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

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

Binary NAF Because the inverse of a point is cheap to compute (as $-P = (x, x + y)$, requiring just one addition in the underlying field), it can also be useful to represent k in a binary non-adjacent form (NAF), in which $k_i \in \{-1, 0, 1\}$. This representation is computed in a similar way to a binary representation, except that when we set $k_i \neq 0$, we choose whether $k_i = 1$ or $k_i = -1$ in such a way that the next digit of the representation (k_{i-1}) is zero. Therefore the non-adjacent form of an integer has the property that there are no adjacent 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 a non-adjacent representation can be generalised to a have a window size of w, where representation of k uses values $k_i \in \{-2^{w-1} + 1, \ldots, 2^{w-1} - 1\}$. This representation is even more sparse, having an average density of non-zero digits of $1/(w + 1)$ [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 performs iterates over the bits in a binary representation of k, keeping track of two elliptic curve points, P_1 and P_2, and performing exactly one double and one add per iteration to maintain the invariant $P_2 - P_1 = P$. This method performs a fixed number of elliptic curve point operations for all values of k of the same length, and so it less vulnerable to timing attacks.

Affine Montgomery's method This method can be further refined to perform a lower, but still constant, number of calculations in each iteration. Rather than using the standard point doubling and addition routines, it only keeps track of the x coordinates of the points P_1

and P_2, meaning that less field arithmetic needs to be performed. After the main loop, the y coordinate of the result can be calculated (using P, the x coordinates of P_1 and P_2, and the invariant $P_2 - P_1 = P$).

Projective coordinates

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

Projective coordinates allow the point doubling and point addition formulae to be rewritten without any inversions in the field $\mathbb{K}$, which, for $\mathbb{K} = \mathrm{GF}(2^m)$, can be expensive compared to multiplications. There are two forms of projective coordinates which I explore in this project: Jacobian coordinates and 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 ECDSA, perform some arithmetic in prime fields. For the field Z_p, the elements are the numbers $\{0, 1, \ldots, p-1\}$, and the operations $+$ and $\cdot$ are addition modulo p and multiplication modulo p respectively. The additive inverse of a point x is $p - x$, and the multiplicative inverse can be found with the extended version of Euclid's algorithm.

2.1.4 Elliptic curve cryptography

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

In such schemes, details of the group and generating point are fixed and publicly known. For curves over binary fields, these details, known as Elliptic Curve Domain Parameters, are stored in a septuple (SEC 1, 2009, [13]): $T = (m, f(x), a, b, G, n, h)$.

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

Another important primitive are Elliptic Curve Key Pairs, which are a tuple (d, Q) associated with a septuple T of curve domain parameters [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 a the Elliptic Curve Digital Signature Algorithm (ECDSA), a variant of DSA, which allows users to sign messages and verify signatures produced by other users. In this scheme, each entity is assumed to have an elliptic curve key pair, where the public component is publicly known and trusted. So long as an attacker has not captured the corresponding private key, it is computationally infeasible for them to produce a "valid" signature, meaning the scheme provides existential unforgeability [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 (appendix A, algorithm A.1). Any other entity who knows U's public key, Q_U, can verify the signature $S = (r, s)$ for message M, with the ECDSA Verifying Operation (appendix A, algorithm A.2).

ECDH

The Elliptic Curve Diffie-Hellman (ECDH) scheme allows two entities to agree on a symmetric key over an authenticated channel without any eavesdroppers also being able to derive the key. The security of this scheme relies on the computational complexity of a problem related to ECDLP, known as the Elliptic Curve Diffie-Hellman Problem (ECDHP): given $Q_1 = d_1 G$ and $Q_2 = d_2 G$, determine $d_1 d_2 G$ ([13], B.2.3).

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

2.2 Requirements analysis

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

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

  - This involves finding standard fields and curve domain parameters, and a suitable way to present them to users, which will be lower difficulty than other components. However, it is still medium priority because these standard values will be important for testing correctness, and therefore it also has a medium risk.
  - For this component, I will use the standards document SEC 2 (version 2.0, 2010, [15]) from the Standards for Efficient Cryptography Group (SECG, [11]), and the equivalent document [16] from NIST for additional detail or information where needed. However, for consistency in naming and format, I will only use curves and fields from SEC 2.

- **Cryptographic primitives**
  priority: low, *difficulty: medium*, *risk: low*

  - Implementing examples of ECC based schemes is low priority because it does not have any dependant components, and will not impact the rest of the system if implemented poorly.
  - For this component, I will use standards document SEC 1 [13] from SECG, which lists and describes in detail all of the cryptographic primitives that I will need to implement.

2.3 Methods and tools

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

2.3.1 Software engineering principles

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

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

2.3.2 Choice of tools

This dissertation was written using LaTeX, using the TeXMaker editor [17]. BinaryECC was written in Julia, using Juno [18], an Integrated Development Environment, and it depends on two Julia packages:

1. StaticArrays [19]: 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 [20]: 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 [21], to analyse the performance of different algorithms; Plots [22] and LaTeXStrings [23], to produce graphs visualising performance results using PGFPlots as a backend; and Documenter [24], to produce documentation hosted on GitHub.

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

2.4 Summary

This chapter described the mathematical background required to build a package for ECC, from the arithmetic in the underlying binary fields, to the definition of elliptic curve point addition. It also outlined several algorithms and representations that will be used in the next section, such as projective coordinates and the non-adjacent form of integers, before moving on to describe the project from a software engineering perspective. I broke down the project into its constituent parts and considered the difficult, risk, dependencies, etc., of each to help structure the implementation process.

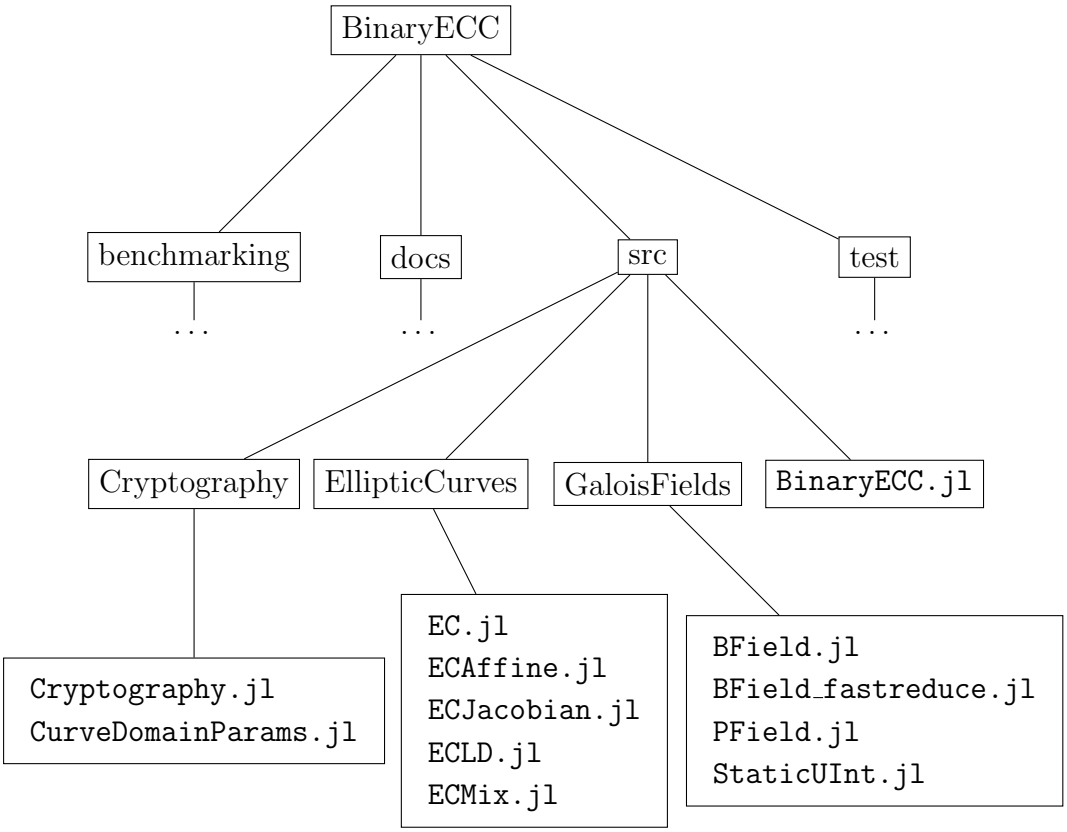
Chapter 3

Implementation

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

3.1 Repository overview

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 such as multiplication need to know both the value of the element (which is a binary polynomial) 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 be provided once for multiple elements of the same field. However, this is at the cost of both usability, because programmers must remember to create and handle 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 (either as an explicit field within a struct, or as part of the object's type), allowing the more intuitive notation `a * b` to be used. In my project proposal, one of the success criteria was for the package to be able to parse expressions with high level syntax close to that used in mathematics, and so I chose this latter representation.

Another aspect to consider is how 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 unsigned integer type (e.g. `UInt32`, `UInt64`, etc.) 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 its degree, m. This means that elements of different fields will have different

types, allowing the compiler to check that functions have been called on points from the same field, rather than having to check at runtime and possibly throw an exception. This also means that the field information does not use any memory in the struct, as it is a constant.

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, the first time a function is called with that new type, a specialised version of that function will also need to be compiled, damaging runtime performance. However, this situation is unlikely to occur because the choice of field is usually part of the core design of a cryptographic system, often based on hardware or processing constraints. As a result, the compiler will only need to create one specialised version of each function and new fields will not be encountered at runtime, minimising the impact of using a parametric type. A second problem is that only "bits" types (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 should also not pose a problem.

The standard binary fields offered by BinaryECC are therefore provided as specialised field element types, such as `BFieldElt163{T,L} = BFieldElt{163, 2^7+2^6+2^3+1, T,L}` (i.e the binary field with reduction polynomial $x^{163} + x^7 + x^6 + x^3 + 1$), along with a function that only requires the user to select the word type `T`, which then returns a binary field type with a suitable number of words `L`. This also allows users to easily define their own custom binary fields, by setting the `D` and `R` parameters themselves.

Point information

The struct `BFieldElt{D,R,T,L}` contains one field which is an integer (often hundreds of bits long), representing a binary polynomial in the field specified by `{D,R}`. The initial version of this type used objects of the pre-existing `BigInt` type (and so 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

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

Like the field type, `StaticUInt` is also a parametric type, this time with parameters `{L,T<:Unsigned}` representing the length (in words) of the array with `L`, and the type of the words with `T`, which must be an unsigned integer (i.e., it must subtype `Unsigned`, an abstract Julia type). This parameterisation of the word type adds flexibility, as package does not assume any particular word size. It 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. Providing the array length as a type parameter takes advantage of Julia's type dispatch by providing different implementations of various `StaticUInt` operations depending on the lengths of the given objects.

As before when using values as parameters, there is a risk of running into a new version of `StaticUInt{L,T}` at runtime and having to compile functions anew. Fortunately, this risk is mitigated, just as before, by the fact that only one field will be in use in a given system, and so only two lengths of `StaticUInt` 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 [19], which builds upon Julia's tuples to provide statically-sized arrays. From this package, I explored both the `SVector` and `MVector` types, which supported immutable and mutable arrays respectively. I found that `MVector` was best suited to the kinds of operations that the binary polynomials required. Several 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 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}`, where `D` and `R` specify the binary field, and the value of the element itself is held in an array of unsigned integers, of type `StaticUInt{L,T}`.

3.2.2 Reduction

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

Standard reduce This first method takes a polynomial $a(x)$ of degree $m + n$ and iterates over the coefficients a_i of the polynomial, from $i = m + n$ down to $i = m$, adding $f(x) \cdot x^i$ to it if $a_i = 1$. To save space, my implementation does not store the entire reduction polynomial $f(x)$ as a field element, instead only storing $f'(x)$ and adding $f'(x)$ to $a(x)$. This means that

at the end of the reduction routine there will still be many higher terms incorrectly left in $a(x)$, but these are easily be cleared in one operation.

Additionally, I wrote a specialised function `shiftedxor!(a, b, i)` (in Julia, a function ends with an exclamation mark if it writes into preallocated memory), which takes two `StaticUInt` objects and an integer `i`. The function then performs the operation `a ^ (b << i)` (corresponding to $a(x) + b(x) \cdot x^i$) all at once, writing the result into `a`. This greatly improves the memory usage of the reduction routine, because the second argument, `b`, holds $f'(x)$ which can typically fit into just one word. Performing the shift followed by the exclusive-or would necessitate storing the intermediate result `b := b<<i` in a `StaticUInt` with enough space to hold $\deg(f'(x)) + m + 1$ bits (because `i` may be up to $m - 1$), the vast majority of which will be zeros. For a word size of 64 and field $\mathrm{GF}(2^{571})$, this is the difference between storing `b` in nine words, or just one single word.

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 binary polynomial of degree $2m - 2$ (the most likely size, as it is the result of multiplying two degree $m-1$ polynomials) to an equivalent polynomial of degree $m-1$, the standard reduction algorithm performs roughly $\frac{m-1}{2}$ `shiftedxor!` operations. The fast reduce algorithm, on the other hand, performs roughly $\frac{m-1}{W}b$ of these operations, where b is the number of terms in $f(x)$ and W is the word size. 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 the graphs from figure 3.1, where fast reduce outperforms standard reduce for every word size, and that performance gap only increases as the words 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 discussed 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 this method 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. 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 microcontroller) 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.

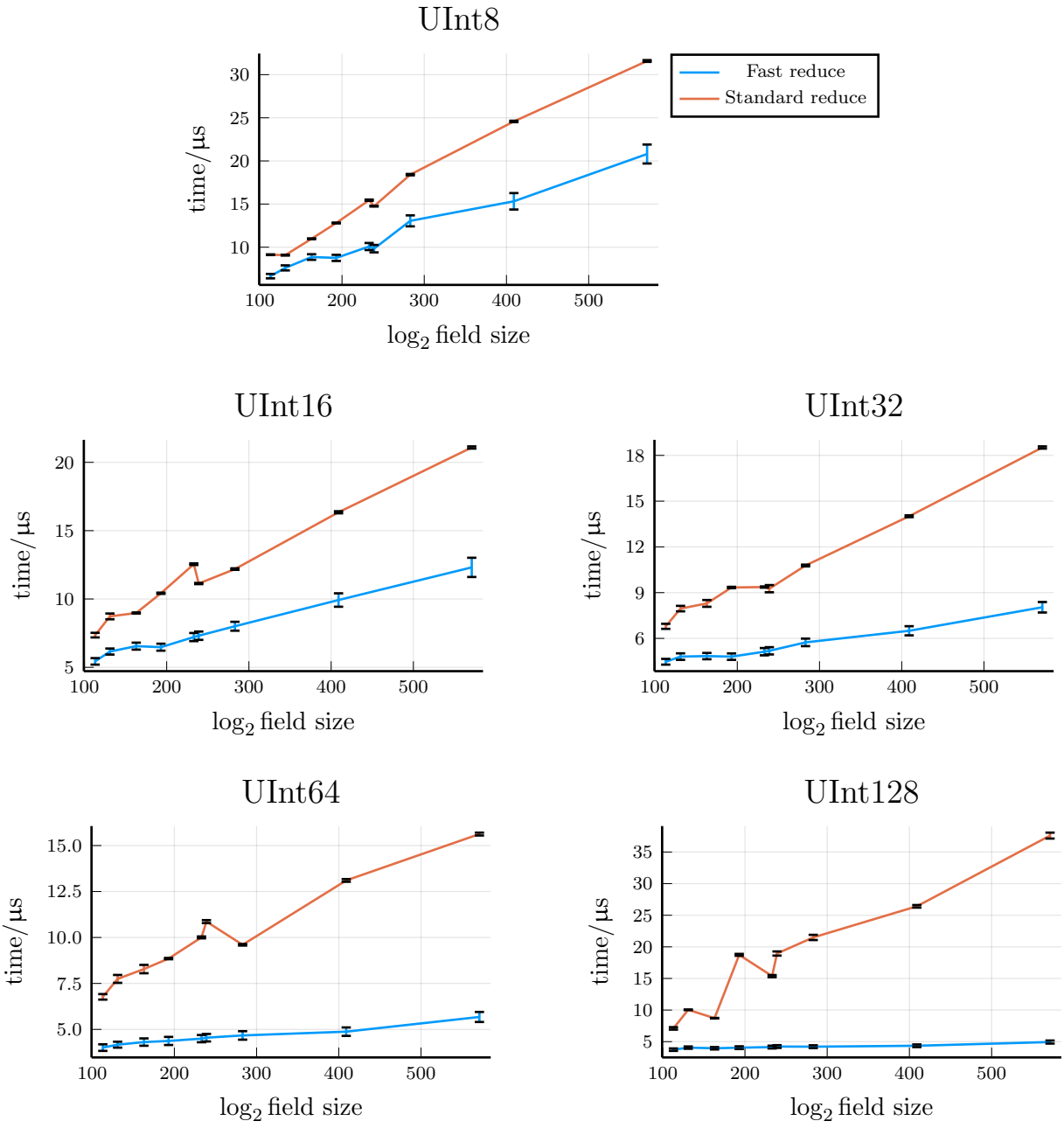


Figure 3.1: Comparison of the performance of binary field reduction methods as field order increases, for a selection of possible word sizes.

3.2.3 Multiplication

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

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 intermediate 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 performed at the same time as multiplication. For a right-to-left method, instead of calculating $b_i(x) = b(x) \cdot x^i$ on each iteration i, 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 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, we can see in figure 3.2 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 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 windowing (width w), 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.

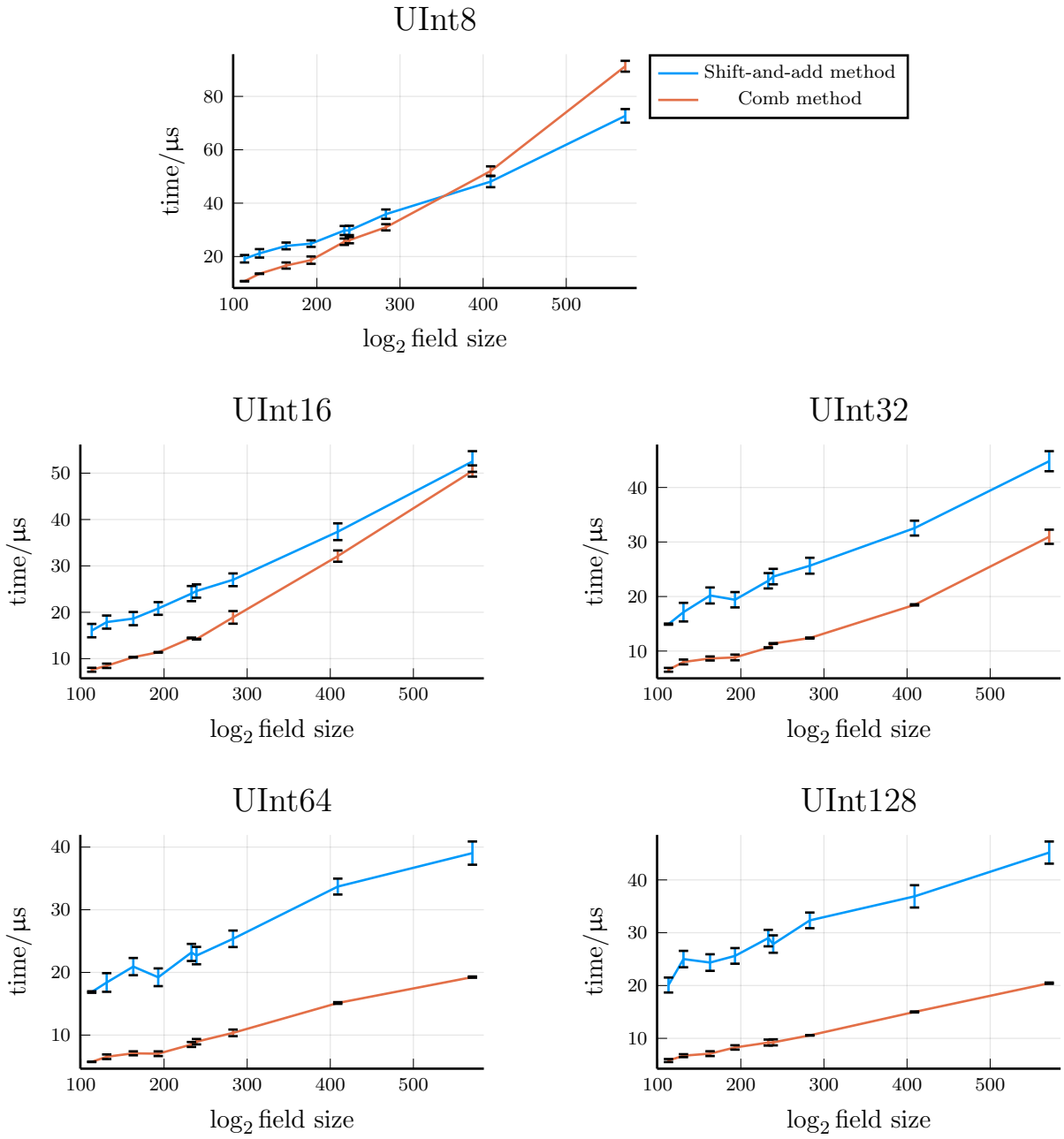


Figure 3.2: Comparison of the shift-and-add and comb methods for binary field multiplication, as field order increases.

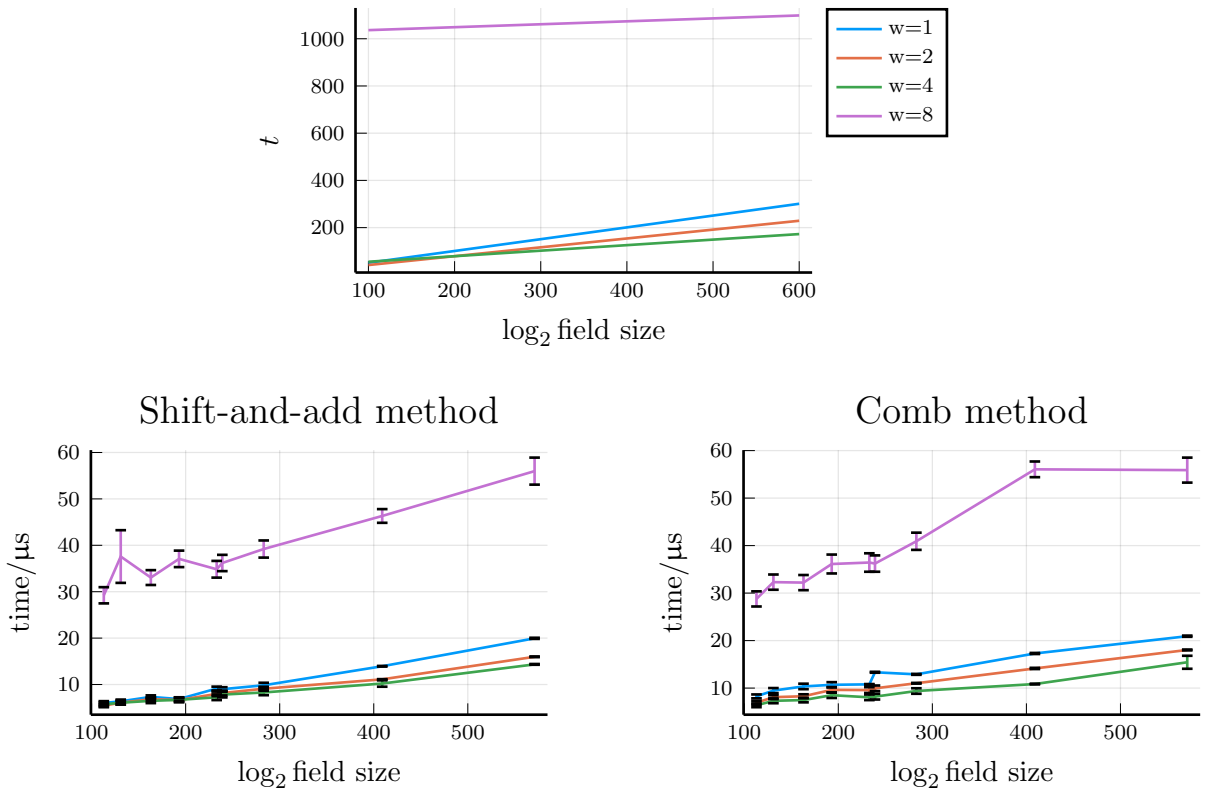


Figure 3.3: Effect of different window sizes on the shift-and-add and comb methods for binary field multiplication. The first figure shows the modelled times, using the formula $t = \frac{m}{2} - w2^{w-1} - \frac{m}{w}(1 - 2^{-w})$, and the second two show the actual timing results.

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 in figure 3.3 (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 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 (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

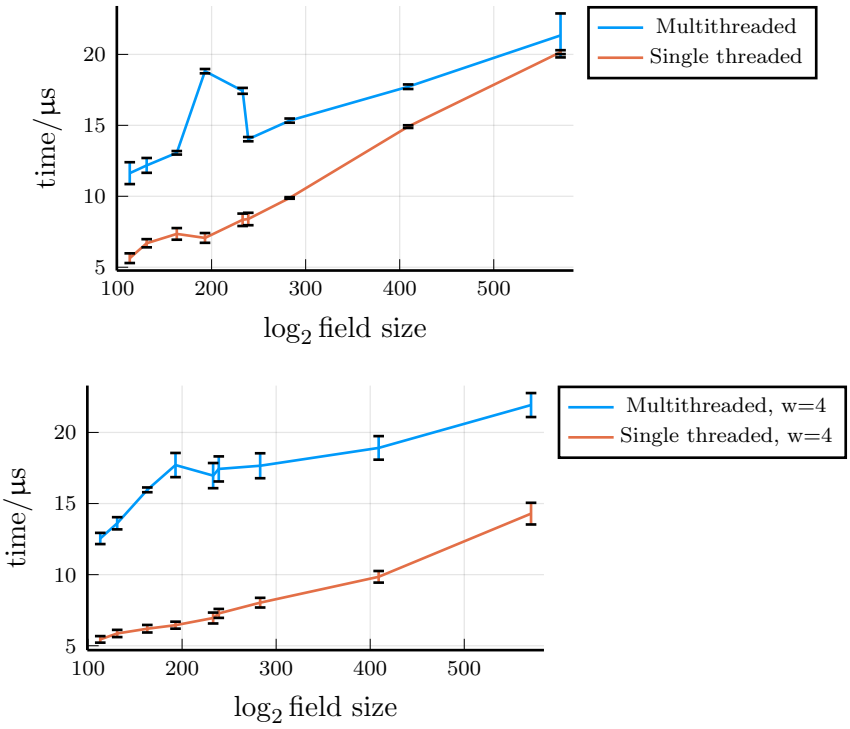


Figure 3.4: Effect of multithreading on the shift-and-add method for binary field multiplication. The first figure compares a two-threaded and single threaded version of a standard shift-and-add routine, while the second shows the same for windowed shift-and-add routines. All data were collected on a dual-core processor.

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 3.4, we can see that this thread-level parallelism does not improve the performance, although the trend shows that it could become worthwhile for larger fields (and indeed, for earlier iterations of BinaryECC where the shift-and-add implementation was more costly, multithreading did improve performance). This 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). For multithreading with more than two threads, this overhead cost only becomes greater.

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 no improvement for any of the standard fields. From the trend in figure 3.4 and the performance improvement from multithreading in the earlier version of BinaryECC, multithreading only outperforms a single thread once the single thread takes around 25µs to compute the result. 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, 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 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.

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. As we can see in algorithm 3.5, the main loop of inversion involves two multiplications (on lines 15 and 16), 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 can instead directly call `shiftedxor!(u, v, j)` to perform $u(x) := u(x) - v(x) \cdot x^j$ (and similar for line 16). 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, where the type of that object simply indicates which field the curve is defined over. Elements of an elliptic curve group then simply contain an `EC{B}` object that provides the group information that is needed for arithmetic.

Although this prevents the compiler checking that operations only occur on elements of the same group at compile time (necessitating an extra type, `ECMismatchException`, that inherits

Input: Point $a(x)$ in field with reduction polynomial $f(x)$
Output: Point $g_1(x)$ such that $a(x) \cdot g_1(x) \equiv 1 \pmod{f(x)}$

```
1  if a(x) = 0 then
2  |  Throw DivideError and stop;
3  end
4  u(x) := a(x);
5  v(x) := f(x);
6  g1(x) := 1;
7  g2(x) := 0;
8  while u(x) ≠ 1 do
9  |  j := deg(u) − deg(v);
10 |  if j < 0 then
11 |  |  swap(u(x), v(x));
12 |  |  swap(g1(x), g2(x));
13 |  |  j := −j;
14 |  end
15 |  u(x) := u(x) − v(x) · x^j;
16 |  g1(x) := g1(x) − g2(x) · x^j;
17 end
18 Output g1(x) and stop;
```

Figure 3.5: Binary field inversion

from `Exception` and 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 would cause many different specialised versions of the same functions to be compiled, if the elliptic curve group were a part of the type system. Additionally, the values of `a` and `b` are likely to be hundreds of bits long (as they belong to a binary field of order 2^m, where m is in the hundreds), and so cannot be simply stored in a bits type, which is a requirement for parameter values. This issue could be avoided by allowing only Koblitz curves (in which $a, b \in \{0, 1\}$), but this would make the package unnecessarily restrictive.

Elements

For the representation of elements, we have the abstract type `AbstractECPoint{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}`, are similar apart from one additional field, `z` of type `BFieldElt{B}`.

Point at infinity Each elliptic curve group must also contain the identity, known as the point at infinity and written in mathematical notation as $\mathcal{O}$. One possible implementation of $\mathcal{O}$ is to create a type for it, `ECPoint0{B}`, with just one field, `ec`, storing information about the elliptic curve group it belongs to. It would therefore need to be a concrete type (as Julia only allows abstract types to have behaviour, and not state), and for it to fit into the existing type system it could subtype `AbstractECPoint{B}`. However, this would mean that certain functions, such as point addition, would suffer from type instability, in that the types of the function's arguments would not provide the compiler with enough information to deduce the (concrete) type of the result. For example, performing the addition `p1 + p2` (where `p1` and `p2` are both instances of the `ECPointAffine{B}` type) could return either an `ECPointAffine{B}` (in the general case) or an `ECPoint0{B}` (if `p1 == -p2`). The presence of type instability limits the performance of the code that can be produced by compiler, because if it cannot know at compile time what the type of an object is, it has to use dynamic dispatch and work out which functions to call at runtime.

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

Summary

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

3.3.2 Point addition and doubling

The two basic operations that can be performed on elements in an elliptic curve group are addition and doubling, 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 coordinates $(\frac{X}{Z^2}, \frac{Y}{Z^3})$. New formulae for point addition and doubling can therefore be produced by substituting $x = \frac{X}{Z^2}$ and $y = \frac{Y}{Z^3}$ into the formulae for affine coordinates. 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 $A = Y_1 Z_2^3 + Y_2 Z_1^3$ $C = X_1 Z_2^2 + X_2 Z_1^2$ $B = Z_1 Z_2 C$

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 $A = X_1^2 + Y_1 Z_1$ $B = X_1 Z_1^2$

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:

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

where $A = Y_1 Z_2^2 + Y_2 Z_1^2$ $C = X_1 Z_2 + X_2 Z_1$ $B = Z_1 Z_2 C$

The formulae for point doubling are:

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

where $A = X_1^2 + Y_1$ $B = Z_1 X_1$

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.

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

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

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.

Double-and-add

For the initial prototype of the elliptic curve group module, 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 to simply 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 the figure below, 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 $2^w A + m(D + \frac{1}{w}(1 - 2^{-w})A)$

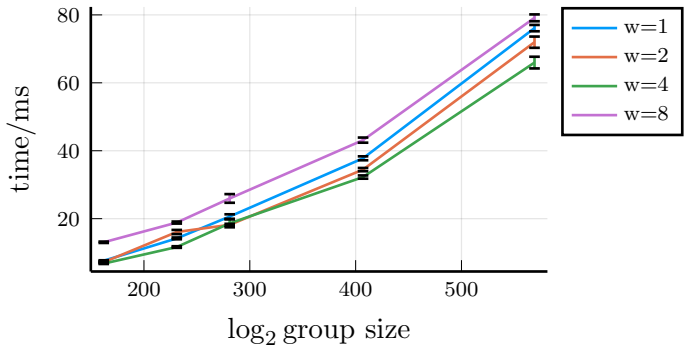


Figure 3.6: Comparison of window sizes when performing scalar multiplication with the double-and-add method.

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 non-adjacent form (NAF) of an integer is the representation that has the fewest nonzero 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.

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, shown in algorithm 3.7, 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 3.8, 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 3.9 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 nonzero digits in k must perform both an point doubling and addition, and so it may

Input: Elliptic curve point P and integer k
Output: Elliptic curve point $Q = k \cdot P$

if $k < 0$ **then**
  Output $(-k) \cdot (-P)$ and stop;
end
if $P = \mathcal{O}$ **then**
  Output $\mathcal{O}$ and stop;
end
$Q := \mathcal{O}$;
while $k > 0$ **do**
  if $k \equiv 1 \pmod 2$ **then**
    $t := 2 - (k \mod 4)$;
    $k := k - t$;
    if $t = 1$ **then**
      $Q := Q + P$;
    else
      $Q := Q - P$;
    end
  end
  $P :=$ double(P);
  $k := k/2$;
end
Output Q and stop;

Figure 3.7: Scalar point multiplication with binary NAF.

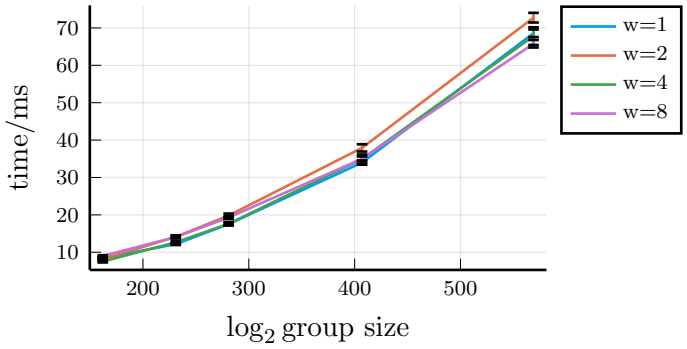


Figure 3.8: Comparison of window sizes when performing scalar multiplication with binary NAF representation.

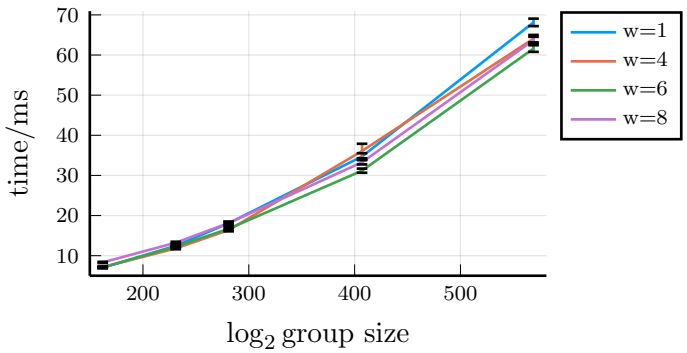


Figure 3.9: Comparison of different widths of NAF for scalar multiplication.

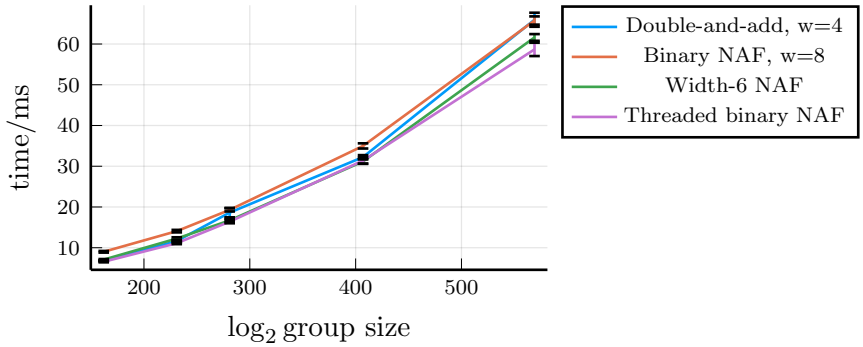


Figure 3.10: Comparison of different methods (each with their optimal parameterisation) for scalar point multiplication.

be possible to improve performance by calculating these two operations simultaneously with multithreading. For multiplication performed in a left-to-right manner, the 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, where 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) on every iteration, which would be inefficient. Therefore we must make the choice between windowing and multithreading.

Figure 3.10 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 multithreaded binary NAF (without windowing), with the multithreaded routine having a slight advantage for larger groups. The default scalar multiplication method in BinaryECC is therefore the multithreaded 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

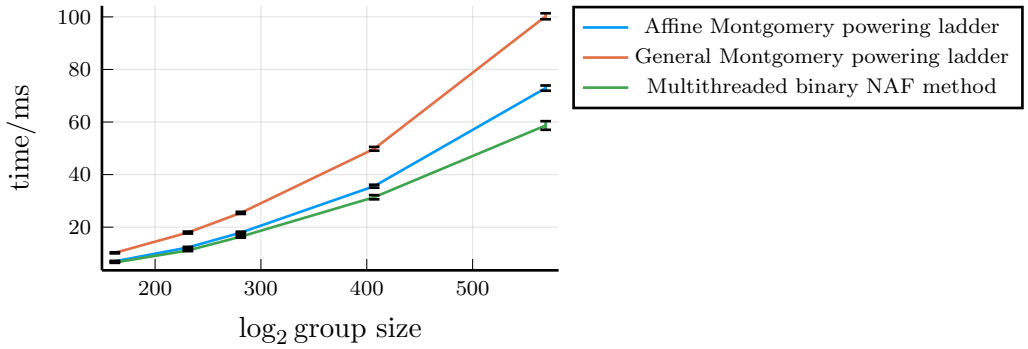


Figure 3.11: Comparison of the general and specialised (affine) version of Montgomery's ladder, alongside the default scalar multiplication routine.

this algorithm performs one point doubling and one point addition per iteration, it will take roughly 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 3.11, it is around a third faster than the general version, but still slower than the fastest scalar multiplication method, showing that a trade-off must be made between security and performance.

3.4 Cryptographic primitives

BinaryECC has example implementations of several cryptographic protocols, to demonstrate how the package could 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, $\mathrm{GF}(p)$, 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.

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 for me 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 of `G`, `n`; and the cofactor of `n`, `h`. 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, since the type parameter `B` provides the binary field GF(2^m) and the logarithm of its order (m), and 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 size and return a `CurveDomainParams` object. For example, `SECT163K1(UInt64)` will return the septuple for the curve "sect163k1", in which a word size of 64 has been used. 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 storing 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, therefore 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 $\sum_{j=0}^{i} k_{(\log_2 k)-i} \cdot G$ (where k_i is the i^{th} digit of k), and 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. 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 use windowing with memoisation, it still outperforms all other scalar multiplication methods when the input point has been seen before. However, using it with arbitrary point will unnecessarily bloat the dictionary storing the doubles (by filling it with results that are unlikely to be needed again), and so the default scalar multiplication in BinaryECC does not use memoisation. Instead, it is left to the programmer to identify cases where they are multiplying by a common point.

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), again following the procedures in SEC 1. Where hashing algorithms are required, the function `sha256` from the Julia package `SHA` [20] 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 [25]) and algebraic identities (such as $\frac{x}{x} = 1_F$), formed a continuous part of the development process. This enabled me to quickly detect 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` [24], hosted alongside the BinaryECC source code on GitHub [1].

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

[1] Documentation: `mkfryatt.github.io/BinaryECC/`

offer several cryptographic functions and types, such as `CurveDomainParams` and ECDSA signatures. During the process of implementation, I have striven 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.

Chapter 4

Evaluation

4.1 Performance

4.1.1 Comparison with Julia

GaloisFields.jl

Nemo.jl

4.1.2 Comparison with C

4.2 Requirements met

4.3 Limitations

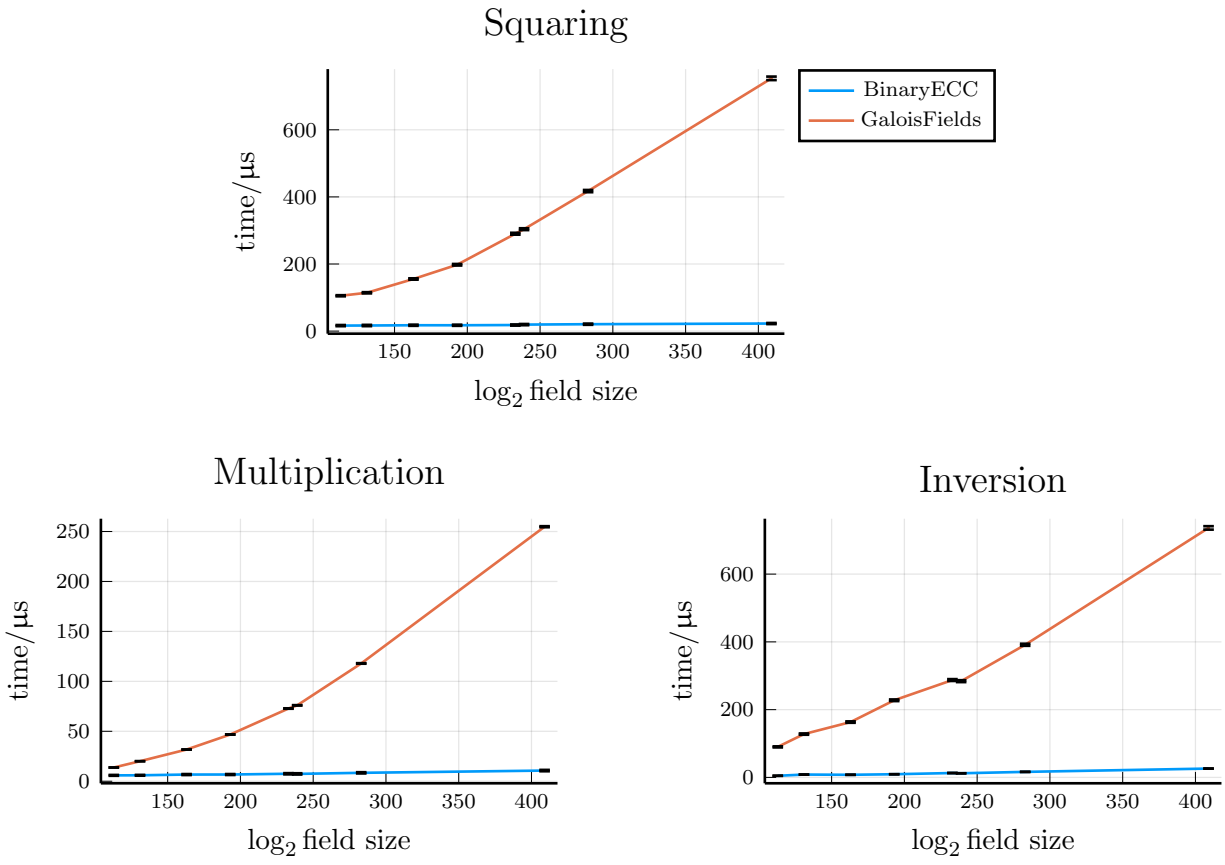


Figure 4.1: blah blah

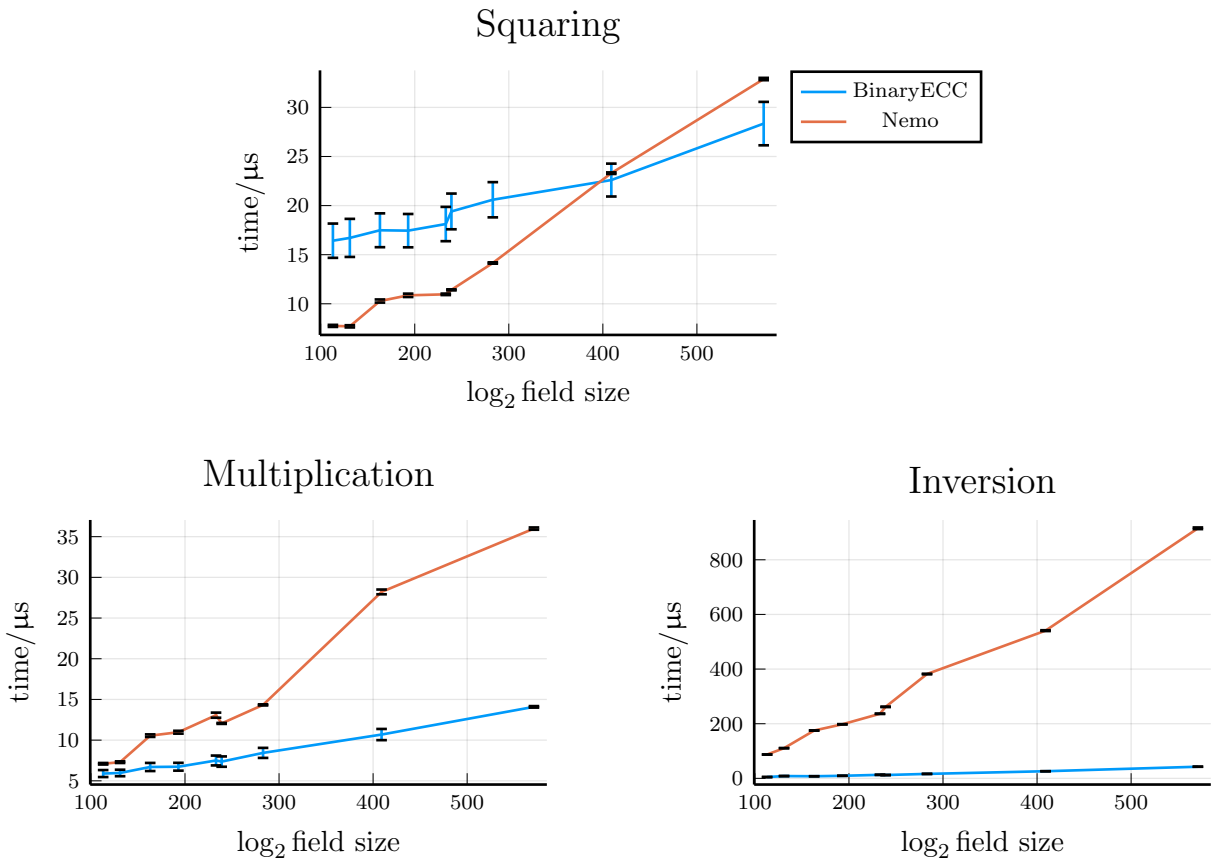


Figure 4.2: blah blah

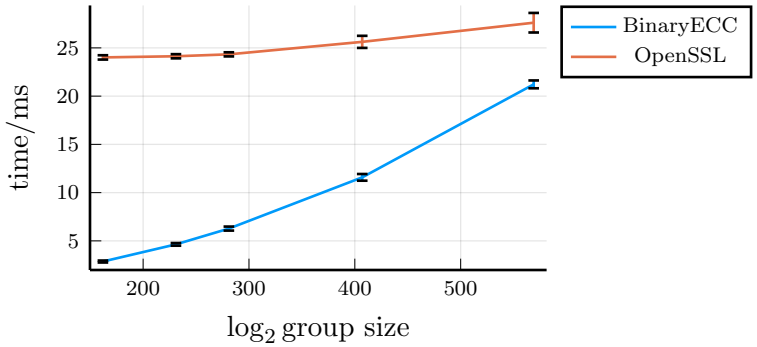


Figure 4.3: blah blah

Chapter 5

Conclusions

Bibliography

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

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

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

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

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

[6] 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.

[7] 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`.

[8] OpenSSL Software Foundation, "OpenSSL." Available: `https://www.openssl.org/`, 1999-2018.

[9] 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.

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

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

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

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

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

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

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

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

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

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

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

Appendix A

Algorithms

Input: Message M to be signed, key pair (d_U, Q_U), parameters T
Output: Signature $S = (r, s)$ on M, or "invalid"

1 Select ephemeral key pair (k, R) associated with T;
2 $\overline{x_R} := \text{int}(x_R)$;
3 $r := \overline{x_R} \mod n$;
4 **if** $r = 0$ **then**
5 | Return to step 1;
6 **end**
7 $H := \text{hash}(M)$;
8 $e := \text{int}(H)$;
9 $s := k^{-1}(e + rd_U) \mod n$;
10 **if** $s = 0$ **then**
11 | Return to step 1;
12 **end**
13 Output $S = (r, s)$;

Figure A.1: ECDSA Signing Operation

Input: Message M to be verified, public key Q_U, parameters T
Output: "valid" or "invalid"

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

Figure A.2: ECDSA Verifying Operation

Appendix B

Project Proposal