# Elliptic Curve Cryptography: Weierstrass, Montgomery, and Edwards Forms

Miranda Wood

**The University of St Andrews**

**Mathematics M.Sc.**

Year of Submission: 2019
Supervisor: Prof. Peter Cameron

**Abstract**

Elliptic curve cryptography has all but replaced other encryption methods used today. However, the underlying curves are still generally implemented in their originally introduced Weierstrass form. This is despite literature from the 1970s to the present day demonstrating the advantages of Montgomery and Edwards forms. What follows is an explanation and simulation of elliptic curve cryptography in all three forms to test such claims, connecting group-theoretic concepts to the security afforded by the elliptic curve discrete logarithm problem. The equivalences between forms of elliptic curve are shown to be birational or isomorphic with a functional demonstration of those mappings in Python. This code further implements the group operations of large elliptic curves to derive cryptographic keys for real-world use. It is applied to the central aim of this project; to find which form of elliptic curve is the most efficient and secure. The report additionally includes a novel implementation of the Tonelli-Shanks algorithm for square roots and an original method for deriving cube roots in finite fields.

# Contents

# Chapter 1

# Introduction

## Preface

I certify that this dissertation report has been written by me, is a record of work carried out by me, and is essentially different from work undertaken for any other purpose or assessment.

Please note that the paper *Secure elliptic curves and their performance* [19] was written in September 2018 but published and discovered by myself in April 2019, months after work began on this dissertation. Though similar in its aims, this project is essentially different and took no inspiration from it.

The reader is expected to have at least undergraduate level knowledge of group theory and algebra (e.g. group axioms). This project is a continuation of work done in *Mathematics of Bitcoin* [53] but is its own standalone paper where no background in cryptography is required.

## 1.1 Elliptic Curve Cryptography

People have exchanged messages for aeons, and have wanted to exchange them secretly for nearly as long. The Spartans used a rod of standardised size to wrap lengths of parchment with seemingly scrambled letters around, revealing the hidden message along the rod. Ciphers, codes produced by substituting letters or symbols according to certain rules, have supposedly been used since Caesar's reign. They were written about in *Casanova* and were the basis of the Enigma code [23].

Ciphers require a secret key which defines the rules for encoding and decoding secret messages. In the age of letters, correspondents would meet in person to decide on the key for future communication. What if they cannot meet in person? What if the secret messages must be sent across vast distances, as technology allows us to do today? A 20th century invention, or discovery depending on how one looks at it, which addressed

this problem is public key cryptography.

Public key cryptography is used for encrypting data over public or otherwise insecure channels. It allows two people, conventionally Alice and Bob, to share secret messages without having to meet [23, p. 87]. Each person has their own *public key*, which may be published to anyone, and a *private key*, which is never shared [50, p. 2].

Essentially, the public key $P$ is the encryption key and the private key $d$ is the decryption key. Encrypting a message, or *plaintext*, with $P$ converts it into unreadable *ciphertext*, which can only be deciphered by applying the secret key $d$ [23, p. 55]. The idea for each person having two keys as opposed to one arose in the 1970s, famously by Diffie and Hellman in 1976, and less famously by mathematicians Ellis and Cocks at GCHQ in 1970 [35, p. 147, 53, p. 7]. A functioning public key *cryptosystem*, or family of transformations allowing for encryption and decryption, was published in 1976 by Rivest, Shamir, and Adleman: the well-known RSA system. Again, Ellis and Cocks had independently discovered the system in 1973.

Public key cryptosystems differ from classical *symmetric* cryptosystems which use a single key for encryption and decryption. A more formal definition of a symmetric system states that if it is broken, i.e. if the single key is leaked, decryption becomes just as easy as encryption [23, p. 88]. Therefore if two people wanted to exchange secret messages with a symmetric cryptosystem, they must find a way to privately agree on an encryption key first. Before the advent of *asymmetric* (a key for each computation) cryptography, key exchange was a widespread logistical problem [53, p. 9].

How, then, can the private key $d$ decrypt ciphertext encrypted by $P$ without being discoverable? The keys must be connected in such a way to allow decryption but not that $d$ could be calculated from $P$. The solution calculates $P$ from a secretly chosen $d$ in such a way that it is *computationally infeasible*, or near impossible with current methods, to find $d$ from $P$.

The first[1] method to produce a public and private key pair came from the founders of the asymmetric cryptosystem, Diffie and Hellman [35, p. 147]. To describe their protocol we first introduce some concepts in group theory.

**Definition 1.1.** A set $(G, *)$ is an abelian group if it satisfies the properties of closure, associativity, identity, inverse, and commutativity under the operation $*$.

**Definition 1.2.** A set $(G, *)$ is cyclic if it is abelian under the operation $*$ and it is generated by a single element $g \in G$.

For instance $\mathbb{Z}_5^*$, the integers modulo 5 under multiplication, is a cyclic group. The

[1]There was actually an earlier protocol due to Merkle, *Merkle's Puzzles*, but it is less efficient and more complex than the Diffie Hellman method [35, p. 147]. Merkle is the founder of various methods used in cryptography, including *Merkle Trees* which allow for immutable data storage and are crucial to the security of Bitcoin [53, p. 30].

elements of the group can be generated by powers of 3: $3^1 = 3$, $3^2 = 9 = 4 \mod 5$, $3^3 = 27 = 2 \mod 5$, $3^4 = 81 = 1 \mod 5$. Clearly, $3^5 = 1 \cdot 3 = 3 \mod 5$, restarting the cycle. In this case, 3 is a generator of the group. We denote $\langle g \rangle$ as the group generated by the element $g$.

The Diffie-Hellman (DH) Key Exchange Protocol requires a finite cyclic group, $G$, and a generator of the group, $g$ [35, p. 147]. Say Alice and Bob want to use the DH method to generate public and private keys. They each secretly choose integers between 1 and $|G| - 1$; $d_A$ for Alice and $d_B$ for Bob. Their public keys are calculated using exponentiation:

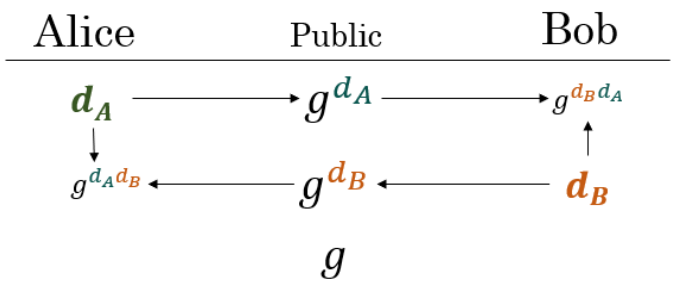



Figure 1.1: DH Key Exchange.

$$P_A = g^{d_A}, \qquad P_B = g^{d_A}.$$

Alice and Bob can exchange their public keys over an insecure channel and find a *shared secret*. Alice computes $P_B^{d_A} = g^{d_B d_A}$ and Bob computes $P_A^{d_B} = g^{d_A d_B}$. Since $g^{d_A d_B} = g^{d_B d_A}$ due to the commutativity of the group, they have the same secret group element without revealing their private decryption keys [23, p. 99]. Even knowing the public key $P_A$ and generator $g$, an adversary cannot extract the secret key $d_A$. Why? The adversary would have to calculate a *discrete logarithm* in a finite group. To explain why this is considered an infeasible calculation we first look at the kind of groups used in cryptography.

Perhaps more frequently, a finite field is used in place of the cyclic group in the DH protocol.

**Definition 1.3.** A finite field $(F, +, \cdot)$ is a set containing a finite number of elements which satisfies distributivity and all the properties of an abelian group under both the additive operation '+' and the multiplicative operation '$\cdot$'.

**Definition 1.4.** The characteristic of a finite field $F$ is the smallest positive integer $n$ such that $n \cdot 1 = 0$, where 1 is the multiplicative identity.

**Proposition 1.5.** *Every finite field has prime characteristic.*

**Proposition 1.6.** *The multiplicative group of every finite field has a generator [23, p. 34].*

Therefore every finite field is a special case of a cyclic group. In fact, it is not

necessary for $g$ to be a generator, though otherwise the private key will simply be a power of $g$ rather than any element of the specific cyclic group [23, p. 99].

The private keys are safe due to the Diffie-Hellman assumption [23, p. 99]:

**Definition 1.7.** The Diffie-Hellman Assumption (DHA) states that, for a suitable finite group $G$, an element $g \in G$, and integers $a$ and $b$, it is computationally infeasible to compute $g^{ab}$ knowing only $g^a$ and $g^b$.

The DHA is not conveniently plucked from thin air. It is considered to be true due to the perceived difficulty of 'reversing' exponentiation in some finite groups. Consider the example of $\mathbb{Z}_5^*$, where its generator 3 is multiplied by itself to 4, 2, then 1. Though a small example, the pattern the generator follows when 'moving' through each element is unpredictable. When the elements of $G$ follow this pattern and $|G|$ is very large, it is infeasible to check each possible exponent of $g$ to find the correct power. To find the exponent in general requires the solving of a discrete logarithm.

**Definition 1.8.** For a multiplicative group $G$, an element $a \in G$ of order $n$, and an element $y \in \langle a \rangle$ the Discrete Logarithm Problem (DLP) asks one to find the unique positive integer $x < n - 1$ in:

$$a^x = y. \tag{1.1}$$

It is considered that finding $x$ in Definition 1.8 above is computationally infeasible[2], so in other words $x$ is *intractable* [41, p. 129].

The DHA and the DLP do not apply for every group, such as $\mathbb{Z}_5^*$, where there are only four possible exponents for $g$ or $a$ which can each be checked quickly. However, it does apply to $\mathbb{Z}_p^*$ when $p$ is very large and prime.

This is because the term 'suitable', used in both definitions above, refers to how the group $G$ is *presented*, not just its size. For instance, the additive group $\mathbb{Z}_{p-1}^+$ is *isomorphic* to $\mathbb{Z}_p^*$. Informally this means the groups are the same set of elements since there is a reversible and rational one-to-one map between them (we shall define isomorphic formally in Section 2.2). However, it is easy to solve the DLP in $\mathbb{Z}_{p-1}^+$ because it is presented as an additive group, so we are finding $x$ in $xa = y \mod p - 1$. Finding the inverse of $a$ in $\mathbb{Z}_{p-1}^+$ can be done easily if $a$ and $p - 1$ are co-prime, using the Euclidean algorithm.

Otherwise a solution exists when $d$, the greatest common divisor of $a$ and $p - 1$, divides $y$. Then this $d$ can be written as $ua + y(p-1)$ where $y = vd$ for some $u, v \in \mathbb{Z}_{p-1}^+$. Therefore, $y = vd = vua \mod p - 1$, giving $x = vu \mod p - 1$. Since the DLP can be

[2]The DLP is believed to be in the class of *NP-Hard* problems, which informally means that no known algorithm exists to solve it in polynomial time [24, p. 43]. However, a correct solution can be checked in polynomial time. This makes it the foundation of many security systems, including other public key cryptosystems [41, p. 129].

solved, the DHA fails for this group. Hence, the DHA is considered at least as strong as the DLP.

However, for $\mathbb{Z}_p^*$, extracting $x$ in $a^x = y \mod p$ is difficult as long as checking every possibility manually is infeasible. This means that the private key $d_A$ is intractable from $g^{d_A} = P_A$ in such a multiplicative finite group.

One scheme which puts these keys to use, particularly in elliptic curve cryptography, is the Digital Signature Algorithm (DSA) [53, p. 10]. Key pairs, generated by the DH protocol, are used to sign messages in such a way that the recipient can verify the identity of the author using their public key.

Say Alice and Bob generate the keys as before and Alice wants to send Bob a signed message. She uses a one-way signing transformation[3] to apply $d_A$ to her message, then sends it to Bob. The signing transformation has a verifying transformation, akin to what decryption is to encryption but with the keys reversed. Bob uses Alice's *public* key $P_A$ to verify the message. The verification will work if and only if Alice's private key was used [8]. Unlike real signatures, which are susceptible to forgery, digital signatures can only be forged if the private key is stolen. Since $d_A$ is intractible, even knowing $P_A$, $G$, and $g$, this can only occur if the private key is found through other means.

Various groups for generating keys with the DH method have been proposed including large prime fields, imaginary quadratic groups, and elliptic curves over finite fields [35, p. 148].

This project focuses on the latter: elliptic curve cryptography (ECC), which was introduced by Neal Koblitz[4] in 1985 [22]. It was founded to have two main advantages over conventional groups for use in the DH protocol; more freedom in choosing the group itself (as there is only one group $\mathbb{Z}_p$ for each $p$ but many elliptic curves over each $\mathbb{Z}_p$) and better security [24, p. 131].

The public key is some point on an elliptic curve, $P$ with $x$ and $y$ coordinates. It is 'multiplied' by another person's integer private key, $d$, to calculate the shared secret, $S$, another point on the curve. This multiplication is in fact a group operation. ECC and how it is used in blockchain technology, particularly in Bitcoin as the keys used in signing transactions with the DSA, is explained in my previous thesis *Mathematics of Bitcoin* [53].

Here we omit the longer proofs and explanations of how and why ECC works as they can be found in [53]. We instead summarise the method of key generation using an elliptic curve in Weierstrass form over a finite field and its security due to the DLP.

[3]Informally, the transformation is computationally infeasible to reverse. A more detailed algorithm can be found in [53, p. 21].

[4]Independently by Victor Miller in the same year [24, 27].

### 1.1.1 Elliptic Curve Arithmetic and Finite Fields

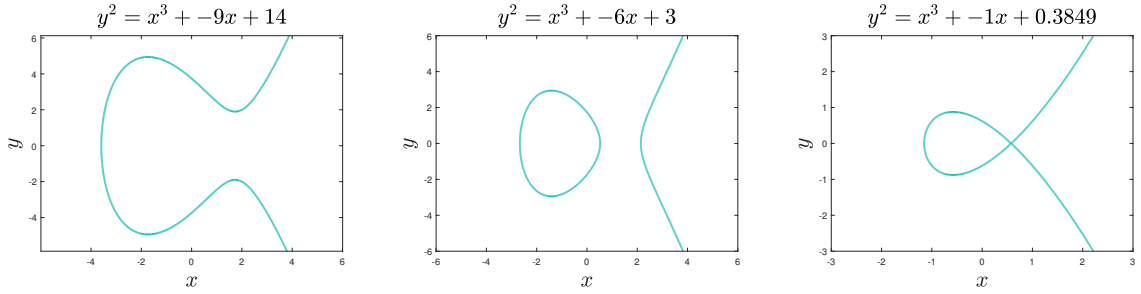



Figure 1.2: Elliptic curves with negative, positive, and zero discriminant [53].

A Weierstrass normal form (WNF) elliptic curve over the real numbers is defined as [53]:

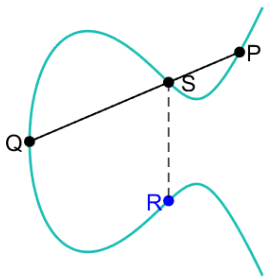
**Definition 1.9.** An elliptic curve $E$ where $a, b \in \mathbb{R}$ are chosen constants which satisfy $4a^3 + 27b^2 \neq 0$ is the set of points such that:

$$E = \{(x, y) \in \mathbb{R}^2 \,|\, y^2 = x^3 + ax + b\} \cup \{O\}. \tag{1.2}$$

If $4a^3 + 27b^2 = 0$ then the discriminant, $\Delta = -16(4a^3 + 27b^2) = 0$. As long as $\Delta \neq 0$, we exclude cubics with repeated roots and therefore singular curves, unsuitable for cryptography [22]. The right hand side of the equation of $E$ will have repeated roots if and only if it shares a factor with its derivative, $3x^2 + a = 0$, which occurs when $\sqrt{-a/3}$ is a root. This by substitution implies $b^2 = -4a^3/27$, hence $-16(4a^3 + 27b^2) = \Delta = 0$.

The elliptic curve $E$ is in fact an abelian group, with points as elements. To use it in the DH protocol we require a group operation which maps two points on the curve to a third. The below is a short summary of point addition largely from [53].

Figure 1.3: Group Operation.




We call the binary operation '+', as it 'adds' two points $P$ and $Q$ in $E$ by constructing a line intersecting both points. Three points in a line sum up to the additive zero element, which we define as the point at infinity $O$. So from the left hand figure, $Q + S + P = O$. In an elliptic curve as in Definition 1.9, the line intersecting $P$ and $Q$ will intersect at a third point, $S$. This is due to a special case of Bezout's Theorem [26]:

**Theorem 1.10** (Bezout)**.** *Let $f$ and $g$ be homogeneous polynomials of degree $m$ and $n$ respectively over an algebraically closed field $F$. If $f$ and $g$ share no polynomial factor, the curves intersect at $mn$ points.*

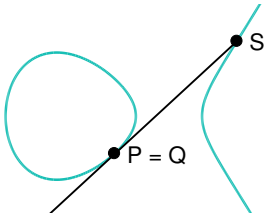
This applies as $\mathbb{R}$ is a field and a straight line, $y = \lambda x + c$, will not share roots with an elliptic curve unless $y - \lambda x - c$ is a factor of $y^2 - x^3 - ax - b$. In this case, the curve equation would be equivalent to $(\lambda x + c)^2 - x^3 - ax - b = 0$.

However, this means $y^2 - x^3 - ax - b = 0$ is identically satisfied with $-x^3 - \lambda^2 x^2 + (a - 2\lambda c)x + b - c^2 = 0$, which can only occur when $\lambda = c = 0$. This equates to the line $y = 0$. Since the discriminant of the elliptic curve must be non zero, $y = 0$ intersects it in either one or three distinct places. In three places, we have a sum of three points which satisfies addition. In one place, we do not consider the line since we only consider those which intersect at two points to find a third. If the curve and line do not share roots, they intersect at $mn = 3 \cdot 1 = 3$ places.

Given our points $P$ and $Q$, the line intersecting them must therefore intersect some third point $S$. The reflection of the $S$ in the $x$-axis, its inverse element in group theory terms, is our result; $P + Q = -S = R$ (see Figure 1.3). We defined $O$ as the additive identity, so that $P + O = O + P = P$. Note that elliptic curves are symmetric in the $x$-axis, so we have three cases for which to find the result of $P + Q$:

**Case 1:** $P = Q$

In the first case, $P = Q = (x, y)$. We assume $y \neq 0$, as this could be considered under the case $P = -Q$. As we cannot define a line 'between' the points, we count the point as two intersections and take the line as the tangent to $P$. The line has slope [53]:



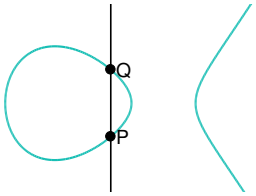

$$\lambda = \frac{3x^2 + a}{2y}.$$

The tangent intersects the curve $E$ at $(x, y)$ and will intersect it again at $(x_S, y_S)$. After some algebra (in [53]) we find:

$$S = (x_S, y_S) = (\lambda^2 - 2x, \lambda(x_S - x) + y),$$
$$P + P = -S = (\lambda^2 - 2x, \lambda(x - x_S) - y).$$

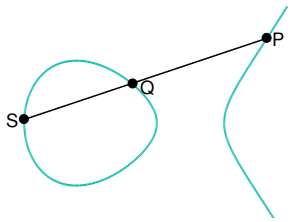
**Case 2:** $P = -Q$

When $P = -Q$, we have $P = (x, y)$ and $Q = (x, -y)$. Here Theorem 1.10 does not apply, as the 'line' is $x = t$ for some constant $t$. The points are additive inverses so, by our definition, $P + Q = O$. Another way of looking at this is to consider the line to intersect the curve at $P$, $Q$, and $O$.



**Case 3:** $P \neq \pm Q$

All other points fall under case 3. We find the slope for the line intersecting two distinct points $P = (x_1, y_1)$ and $Q = (x_2, y_2)$ to be:




$$\lambda = \frac{y_2 - y_1}{x_2 - x_1}.$$

Then our addition is:

$$P + Q = -S = (\lambda^2 - x_1 - x_2, \lambda(x_1 - x_S) - y_1).$$

**Proposition 1.11.** *The set $E$ as in Definition 1.2 is an abelian group under the operation $+$ as defined above.*

Proposition 1.11 shows that elliptic curves can be used to generate keys using the DH protocol. It is proven in [53]. To summarise, our operation $+$ is described:

$$\begin{aligned} P = (x_1, y_1), Q = (x_2, y_2) : \\ P + P &= (\lambda_d^2 - 2x_1, \lambda_d(x_1 - x_S) - y_1), \quad \lambda_d = \frac{3x_1^2 + a}{2y_1}, \\ P + Q &= (\lambda_a^2 - x_1 - x_2, \lambda_a(x_1 - x_S) - y_1), \quad \lambda_a = \frac{y_2 - y_1}{x_2 - x_1}, \\ P + (-P) &= O. \end{aligned}$$

Above it is stated that we can think of the line intersecting $P$ and $-P$ as further intersecting $O$. Therefore this line must satisfy $P + (-P) + O = O$ which gives $O + O = O$. So, doubling the identity element gives itself. With our geometric interpretation, this must also mean that the 'tangent' through $O$ also intersects the curve at $O$, giving a triple point on a line of its own[5].

We define the curves used in cryptography over a finite field, specifically the field $\mathbb{Z}_p$ with prime $p$ [25]. Definition 1.9 remains the same, but Equation 1.2 is modified:

$$E = \{(x, y) \in \mathbb{Z}_p^2 \mid y^2 = x^3 + ax + b\} \cup \{O\}. \tag{1.3}$$

Our addition equations above are exactly the same but modulo $p$. They remain

[5]This is the 'line at infinity' which intercepts the curve at only one point [46, p. 175]. The concept is more easily understood when using projective coordinates, discussed in Section 4.3.

valid in all fields of characteristic $\neq 2$ or 3 [53]. We again can summarise $+$:

$$
\begin{aligned}
P = (x_1, y_1), Q = (x_2, y_2) :& \\
P + P &= (\lambda_d^2 - 2x_1 \mod p, \lambda_d(x_1 - x_S) - y_1 \mod p), \\
\lambda_d &= (3x_1^2 + a)(2y_1)^{-1} \mod p, \\
P + Q &= (\lambda_a^2 - x_1 - x_2 \mod p, \lambda_a(x_1 - x_S) - y_1 \mod p), \\
\lambda_a &= (y_2 - y_1)(x_2 - x_1)^{-1} \mod p, \\
P + (-P) &= O.
\end{aligned}
$$

Since we have individual group elements, plotting an elliptic curve over $\mathbb{Z}_p$ results in a collection of points (see Figure 1.4). From here on, this report uses elliptic curves solely for cryptography, so $\mod p$ is omitted from algebraic equations.

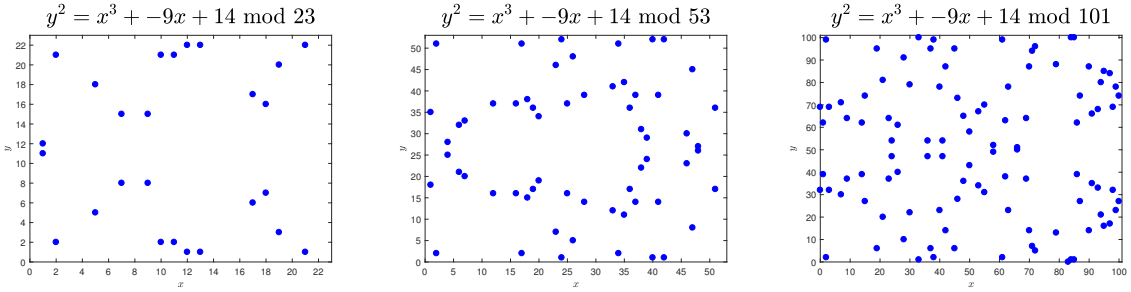



Figure 1.4: Elliptic curves over $\mathbb{Z}_{23}$, $\mathbb{Z}_{53}$, and $\mathbb{Z}_{101}$ [53].

### 1.1.2 Cryptography and the ECLP

*"From a practical point of view, [Elliptic Curve Discrete Logarithms] are the most important issue in discrete logs." – Odlyzko, [41, p. 140].*

We have introduced the concept of intractability in finite fields, specifically the near impossibility of finding $x$ knowing $a$ and $y$ where $a^x = y$, for $a, x, y \in G$. This problem, the DLP (Definition 1.8), is difficult when $G$ is a suitable multiplicative finite group [22].

So how does the group $E$ as in Equation 1.3 have an analogous property? There is no defined multiplication operation, like in $\mathbb{Z}_p$, to multiply two group elements together. Instead the group only has addition, which is repeated to perform scalar multiplication. This scalar multiplication can be carried out in the same order of time as exponentiation in a group [25].

**Definition 1.12.** For an elliptic curve $E$, a point $P \in E$ of order $n$, and a point $Q \in \langle P \rangle$ the Elliptic Curve Discrete Logarithm Problem (ECDLP) asks one to find the

unique positive integer $k < n - 1$ in:

$$kP = Q. \tag{1.4}$$

The result $Q$ is found by computing $P+P+P+..$ $k$ times. Similarly to the DLP the size of the the group generated by $P$, $n$, must be large for the problem to be 'difficult'. However, the ECDLP is thought to be even harder than the DLP, so much smaller group sizes can be used than, say, if one was using a finite field [27].

By Lagrange's Theorem, $n$ must divide the order of the group, $N$ [10, p. 44]. Here, $N$ is the number of points on the curve $E$ . We factorise $N$ and choose a suitably large $n$ first, rather than choose a point $P$. If $n$ is the order of the subgroup, $h = N/n$ is the *cofactor* of the subgroup.

As described in more detail in *Mathematics of Bitcoin*, the ECC key exchange system uses the following parameters:

- **p**, the prime size of the finite field $\mathbb{Z}_p$,
- **a** and **b**, constants defined by the Weierstrass form of E over $\mathbb{Z}_p$,
- **G**, the generator of $\langle G \rangle$,
- **n**, the order of the subgroup $\langle G \rangle$,
- **h**, the cofactor of the subgroup $\langle G \rangle$.

The Diffie-Hellman method is used in terms of the above parameters to produce cryptographic keys. We again meet Alice and Bob, who choose their respective private keys, $d_A$ and $d_B$, as a random integer between 1 and $n - 1$. Their public keys are calculated using repeated point addition as $P_A = d_A G$ and $P_B = d_B G$. The public keys and generator point $G$ are both known to everyone, but due to the ECDLP (and therefore the DHA) the private keys are nearly impossible to find. Alice and Bob can compute a shared secret point, $S$. Alice finds $d_A P_B = d_A d_B G = S$ and Bob finds $d_B P_A = d_B d_A G = S$. The point is the same for both since the group is commutative.

Note that this algorithm makes no mention of Weierstrass form curves specifically. In fact, many other forms of elliptic curve exist which are suitable for cryptographic use and follow the repeated addition key exchange method above. As long as such forms produce a non singular curve and are a suitable abelian group when defined over a finite field, they can be used in the Diffie-Hellman method. Recall the example of $\mathbb{Z}^+_{p-1}$ and $\mathbb{Z}_p$; the groups are isomorphic but their presentation dictates how easy the DLP is to solve in each. In this project we consider equivalent, sometimes isomorphic, forms of elliptic curve which also differ only in their presentation to discover their cryptographic properties.

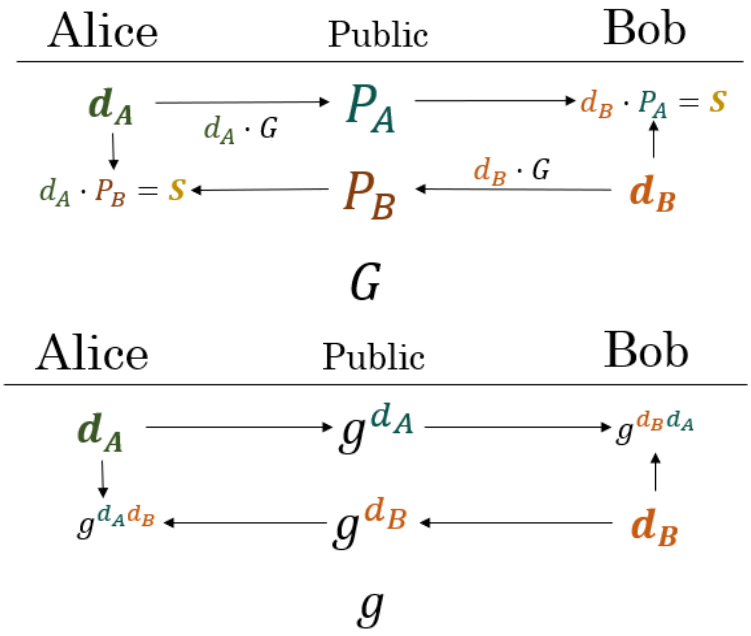



Figure 1.5: Elliptic curve DH and finite group exponentiation DH.

## 1.2 Project Outline

This report runs alongside a Python implementation of key generation using three forms of elliptic curve: Weierstrass, Edwards, and Montgomery. We first introduce those forms and the mappings between them, demonstrating how each can be used cryptographically. This includes the arithmetic on Edwards and Montgomery curves, and a proof of analogousness between Weierstrass and Montgomery arithmetic.

Therefore, we can generate keys to encrypt data using each form of curve. This project asks: which is best?

The aim of the Python code is to simulate key generation using parameters suitable for use in real world cryptography. We test the security and speed of each form of curve to discover which is the most suitable for encryption. In Chapter 3 the code is explained in relation to the algorithms of previous sections, then we outline assumptions based on the literature. The results of the testing are stated and, in Chapter 4, analysed. The Python functions are shown in full in Appendix A.2 rather than in the main text for ease of reading.

Finally, we summarise the findings of the project and what they mean for ECC. As the world of cryptography is ever expanding and relevant, the chapter also includes recommendations for future research and how this project could be extended.

# Chapter 2

# Forms of Elliptic Curves

The Weierstrass Normal Form has been used from as early as 1894 as an elegant representation of elliptic curves [11]. Karl Weierstrass himself lectured on how to obtain his form[1] far before the basis of modern cryptography.

Since Koblitz and Miller's introduction of ECC in 1985, other forms of elliptic curve have emerged. Perhaps the most widely used and known alternatives to the WNF are Montgomery and Edwards curves.

The Montgomery form was introduced in 1987 by Peter Montgomery to speed up factorisation on elliptic curves [38]. Though not originally intended for ECC, it has since "become central to elliptic curve cryptography"[13]. Research has discovered that the form is resistant to some forms of attack when used in cryptography [33]. Work continues relatively recently in assessing its suitability for encryption and building new and better algorithms [13, 37, 42]. Of particular note is that all Montgomery curves are transformable to Weierstrass curves, though the reverse is not true, as a WNF curve must satisfy certain conditions [33, 37]. We discuss this further in Section 2.2.1.

**Definition 2.1.** A Montgomery Curve $M$ where $b(a^2-4) \neq 0$ is the set of points such that [38]:

$$M = \left\{(x,y) \in \mathbb{R}^2 \,|\, by^2 = x^3 + ax^2 + x\right\}. \tag{2.1}$$

We will be using $M$ over the finite field $\mathbb{Z}_p$ with prime $p$:

$$M = \{(x,y) \in \mathbb{Z}_p^2 \mid by^2 = x^3 + ax^2 + x\}. \tag{2.2}$$

Much more recently, in 2007, the Edwards form for elliptic curves was introduced [16]. Harold Edwards took previous notions by Euler and Gauss to find a form which simplifies the addition law greatly. Edwards curves are used widely in cryptography due to this [4, 5, 13]. However, unlike Montgomery curves, they are not so easily

[1] *Volume 5: Lectures on the theory of elliptic functions, edited by J. Knoblauch*, 1915, as cited in [11].

transformed to and from Weierstrass form; "over a finite field, only a small fraction of elliptic curves can be expressed in this form" [4]. This becomes less of a problem when we use the modified twisted Edwards form, discussed below.

**Definition 2.2.** An Edwards Curve $E$ where $cd(1 - dc^4) \neq 0$ is the set of points such that [5]:

$$E = \{(x, y) \in \mathbb{R}^2, d \in \mathbb{R} \setminus \{0, 1\} \mid x^2 + y^2 = c^2(1 + dx^2y^2)\}. \tag{2.3}$$

As usual in cryptography we define our curve over a finite field:

$$E = \{(x, y) \in \mathbb{Z}_p^2 \mid x^2 + y^2 = c^2(1 + dx^2y^2)\}. \tag{2.4}$$

Edwards curves were originally introduced in the form [16]:

$$x^2 + y^2 = a^2 + a^2x^2y^2.$$

Bernstein and Lange used the above Definition 2.2. They also found a form where $c = 1$, which is valid when the curve has a point of order four over a finite field with characteristic $\neq 2$. If such a point does not exist, we can extend the field [5]. This simplifies point addition as described in Section 2.1.2.

Also by Bernstein and Lange (along with Birkner, Joye, and Peters), the twisted Edwards form encompasses far more Weierstrass curves through isomorphisms [4]. It was introduced just a year after the above Edwards form publications.

**Definition 2.3.** A twisted Edwards Curve $T$ where $cd(1-dc^4) \neq 0$ and $a, d$ are distinct non-zero constants is the set of points such that [4]:

$$T = \{(x, y) \in \mathbb{R}^2, d \in \mathbb{R} \setminus \{0, 1\} \mid ax^2 + y^2 = c^2(1 + dx^2y^2)\}. \tag{2.5}$$

Over a finite field the set equation becomes:

$$T = \{(x, y) \in \mathbb{Z}_p^2 \mid ax^2 + y^2 = 1 + dx^2y^2\}. \tag{2.6}$$

We set $c = 1$ as, in fact, all twisted Edwards curves meet the requirements set by Bernstein and Lange. This is because the set of twisted Edwards curves is equivalent to the set of Montgomery curves. More formally [39]:

**Definition 2.4.** Two curves $A$ and $B$ over a finite field $F$ are birationally equivalent if:

- $\exists$ a map of polynomial functions $\phi : A \mapsto B$ such that for every point $P \in A$, $\phi(P) \in B$, everywhere the mapping is defined (*rational*),

- $\exists$ a map $\sigma : B \mapsto A$ such that:

$$\sigma \circ \phi = 1_A, \quad \phi \circ \sigma = 1_B, \tag{2.7}$$

where this equality does not hold for a finite number of points (*birational*).

Essentially, birational equivalence means that the curves can be mapped to one another point by point using quotients of polynomials, apart from a few exceptional elements.

**Theorem 2.5.** *Every twisted Edwards curve over a field $F$ with characteristic $\neq 2$ is birationally equivalent to a Montgomery curve over $F$. Conversely, every Montgomery curve over $F$ is birationally equivalent to a twisted Edwards curve over $F$ [4].*

We prove this theorem with the rational maps given in Section 2.2.2.

**Definition 2.6.** Two curves $A$ and $B$ over a finite field $F$ are isomorphic if they are birationally equivalent over the mappings $\phi$ and $\sigma$ as in Definition 2.4 which are defined everywhere, and:

$$\sigma \circ \phi = 1_A, \quad \phi \circ \sigma = 1_B, \tag{2.8}$$

holds for all points.

This is a somewhat indirect definition of isomorphism, but it serves here to directly compare the types of equivalences we see between forms of curve. In particular [42]:

**Theorem 2.7.** *Every Montgomery curve over a finite field $F$ with characteristic $\neq 2$ is isomorphic to a Weierstrass curve over $F$.*

The converse here is not true, with restrictions and proof given in Section 2.2.1.

## 2.1 Arithmetic on Elliptic Curves

For each form it has been proven that the point operations below satisfy Definition 1.1, and so are abelian groups [4, 16, 38].

### 2.1.1 Montgomery

Like Weierstrass curves, we use the point at infinity $O$ as the identity element, and if $P = (x, y)$ then $-P = (x, -y)$ [51]. Point operations are also similar [38, 51]:

$$
\begin{aligned}
P = (x_1, y_1), Q = (x_2, y_2),\ P, Q \in M :& \\
P + P &= (b\lambda_d^2 - 2x_1 - a, \quad \lambda_d(x_1 - x_S) - y_1), \\
\lambda_d &= (3x_1^2 + 2ax_1 + 1)(2by_1)^{-1}, \\
P + Q &= (b\lambda_a^2 - x_1 - x_2 - a, \quad \lambda_a(x_1 - x_S) - y_1), \\
\lambda_a &= (y_2 - y_1)(x_2 - x_1)^{-1}, \\
P + (-P) &= O.
\end{aligned}
$$

### 2.1.2 Edwards

An advantage of the Edwards form is that the identity element can be written algebraically; it is the point $(0, c)$. If $P = (x, y)$ then $-P = (-x, y)$. The operation $+$ is performed on an Edwards curve thus [5, 16]:

$$
\begin{aligned}
P = (x_1, y_1), Q = (x_2, y_2),\ P, Q \in E :& \\
P + Q &= \left( \frac{x_1 y_2 + y_1 x_2}{c(1 + dx_1 x_2 y_1 y_2)}, \frac{y_1 y_2 - x_1 x_2}{c(1 - dx_1 x_2 y_1 y_2)} \right), \\
P + (-P) &= (0, c).
\end{aligned}
$$

No $\lambda$ required. Note that $d$ must not be square in the field for the addition to work on every input [4].

### 2.1.3 Twisted Edwards

The arithmetic on twisted Edwards curves is unsurprisingly very similar to an Edwards curve and adheres to the group definitions above [4]:

$$
\begin{aligned}
P = (x_1, y_1), Q = (x_2, y_2),\ P, Q \in T :& \\
P + Q &= \left( \frac{x_1 y_2 + y_1 x_2}{c(1 + dx_1 x_2 y_1 y_2)}, \frac{y_1 y_2 - a x_1 x_2}{c(1 - dx_1 x_2 y_1 y_2)} \right), \\
P + (-P) &= (0, c).
\end{aligned}
$$

## 2.2 Mappings of Elliptic Curves

This section aims to describe the transformations between Weierstrass, Edwards, and Montgomery forms of elliptic curve. Most can be completed with a change of variables (in the cases where birational equivalence occurs) which is implemented in Python. We

use these mappings to find equivalent curves in each form to test them as fairly as possible.

### 2.2.1 Montgomery and Weierstrass

Consider a Montgomery curve:

$$M = \big\{(u, v) \in \mathbb{Z}_p^2 \mid Bv^2 = u^3 + Au^2 + u\big\} \cup \big\{O\big\}, \tag{2.9}$$

and a Weierstrass curve:

$$W = \big\{(x, y) \in \mathbb{Z}_p^2 \mid y^2 = x^3 + ax + b\big\} \cup \big\{O\big\}, \tag{2.10}$$

subject to the conditions in Definition 2.1 and Definition 1.2 respectively. We may map the parameters of $M$ to the parameters of $W$ by defining [51]:

$$a = \frac{3 - A^2}{3B^2}, \quad b = \frac{2A^3 - 9A}{27B^3}. \tag{2.11}$$

From $M$ to $W$, we map $O_M$ to $O_W$. All other points are mapped as follows:

$$(u, v) \mapsto \left(\frac{u}{B} + \frac{A}{3B}, \frac{v}{B}\right).$$

Not all Weierstrass curves $W$ may be mapped injectively to a Montgomery curve $M$ [33, 51]. A curve of the form $M$ will have a point of order two, $(0, 0)$, but $W$ may not, particularly if the group has prime order, which can occur with Weierstrass curves [20].

A point $(x, y)$ of order two on $W$ must satisfy $P + P = O$ by definition, which means that $P = -P$ and so $y = -y$. This can only occur when $y = 0 = y^2$. So to find this crucial point we must find a root of the equation $x^3 + ax + b = 0$. Over the reals, this is a relatively simple task involving well known cubic formulae. However, we are working in the finite field $Z_p$ where square and cube roots do not always exist and are much more complex to find. Section 2.3 outlines methods to find them in more detail. For now, we assume $W$ does have a point of order two, and define [42]:

$$\begin{aligned} &\alpha \in \mathbb{Z}_p, \text{ a root of } x^3 + ax + b = 0, \\ &s = (\sqrt{3\alpha^2 + a})^{-1}. \end{aligned}$$

The point mapping from $W$ to $M$ is given by:

$$(\alpha, 0) \mapsto (0, 0), \quad (x, y) \mapsto (s(x - \alpha), sy).$$

The curve is mapped by:

$$A = 3\alpha s, \quad B = s.$$

So $\alpha$ is the $x$ coordinate of a point at which $y = 0$, and $f'(\alpha) := 3\alpha^2 + a$ is the derivative of the curve equation when evaluated at the root $\alpha$. Clearly $f'(\alpha)$ must be a square in the field.

We may now prove Theorem 2.7:

*Proof.* From the condition in Definition 2.1, $B \neq 0$, so the constants $a$ and $b$ are valid. The mapping from $M$ to $W$ takes the single point at infinity of one form to the other and the $x$-coordinate to $x/B + A/3B$ and the $y$-coordinate to $y/B$. By substitution into WNF we can demonstrate that:

$$\begin{aligned}
\frac{y^2}{B^2} &= \frac{(A+3x)^3}{27B^3} + a\frac{A+3x}{3B} + b \\
&= \frac{(A+3x)^3}{27B^3} + \frac{3-A^2}{3B^2} \cdot \frac{A+3x}{3B} + \frac{2A^2-9A}{27B}, \\
y^2 &= \frac{(A+3x)^3 + 2A^3 - 9A}{27B} + \frac{(3-A^2)(A+3x)}{9B} \\
&= \frac{(A+3x)^3 + 2A^3 - 9A - 3A^3 - 9A^2x + 9A + 27u}{27B} \\
&= \frac{(A+3x)^3 - A^3 - 9A^2x + 27u}{27B}, \\
(A+3x)^3 &= A^3 + 9A^2x + 27Ax^2 + 27x^3, \\
\implies By^2 &= x^3 + Ax^2 + x.
\end{aligned}$$

So all points in $M$ are mapped rationally to points in $W$, as required. □

Therefore Weierstrass curves are isomorphic to Montgomery curves if and only if there exists $\alpha$, a root of $x^3 + ax + b = 0$, such that $3\alpha^2 + a$ is a square.

### 2.2.2 Montgomery and Twisted Edwards

Consider a Montgomery curve:

$$M = \{(u, v) \in \mathbb{Z}_p^2 \mid Bv^2 = u^3 + Au^2 + u\} \cup \{O\}, \tag{2.12}$$

and a twisted Edwards curve:

$$T = \{(x, y) \in \mathbb{Z}_p^2 \mid ax^2 + y^2 = 1 + dx^2y^2\}. \tag{2.13}$$

A Montgomery curve always has a point of order four (described below), so we may set $c = 1$. We map the curve parameters of $M$ to those of $T$ and $T$ to $M$ by defining [51]:

$$a = \frac{A+2}{B}, \quad d = \frac{A-2}{B}, \tag{2.14}$$

$$A = \frac{2(a+d)}{a-d}, \quad B = \frac{4}{a-d}. \tag{2.15}$$

This is an isomorphism when $a$ is a square in the finite field and $d$ is not [4]. If there exists a solution when $v = 0$, i.e. to the quadratic $u^3 + Au^2 + u = 0$ where $u \neq 0$, in $M$, then $M$ has points [4]:

$$\left(\frac{-A \pm \sqrt{(A+2)(A-2)}}{2}, 0\right).$$

These points are undefined in terms of the mapping from $M$ to $T$ and can only exist when $ad$ is square. So in that case they are mapped to two points at infinity of order two on $T$. If there is a solution in $M$ when $u = -1$, then it has points:

$$\left(-1, \pm\sqrt{(A-2)/B}\right),$$

which are mapped to two points at infinity of order four on $T$. They will only exist if $d$ is square. When $u = 1$ has a solution in $M$ the points:

$$\left(1, \pm\sqrt{(A+2)/B}\right),$$

are of order four. However, these *are* defined in the mapping of $M$ to $T$ below. In any finite field at least one of $B(A+2) = aB$, $B(A-2) = dB$, and $(A+2)(A-2) = adB^2$ must be a square [13]. This means a point of order four must exist on any Montgomery curve.

The rest of the points are mapped from $M$ to $T$ as follows:

$$O \mapsto (0,1), \quad (0,0) \mapsto (0,-1), \quad (u,v) \mapsto \left(\frac{u}{v}, \frac{(u-1)}{(u+1)}\right).$$

From $T$ to $M$ we have:

$$(0,1) \mapsto O, \quad (0,-1) \mapsto (0,0), \quad (x,y) \mapsto \left(\frac{1+y}{1-y}, \frac{1+y}{(1-y)x}\right).$$

The isomorphism relies on the fact that $(0,0)_M$ and $(0,-1)_T$ are both points of order two, so the mapping can occur in both directions with no exceptional points if $a$ is square and $d$ is not. In other cases, we have a birational equivalence. We have covered

the Montgomery points at which the $M$ to $T$ mapping is undefined ($v = 0$, $u = -1$). In the other direction, the only undefined points are where $x = 0$ and $y = 1$ on $T$. In the first case, $y$ can only be -1 or 1 and the second leads to $(a - d)x^2 = 0$. We know that $a$ and $d$ are not equal, therefore $x$ must be 0. Therefore the only undefined points are $(0, 1)$ and $(0, -1)$ which are taken care of above. We may now prove Theorem 2.5 [4]:

*Proof.* We have set in Definition 2.3 that $a \neq d$, so $A$ and $B$ are valid constants. Since $ad = B(A^2 - 4) \neq 0$, $B \neq 0$ and $A \neq \pm 2$, both $a$ and $d$ are non-zero as required. Therefore $M$ with parameters $A$ and $B$ calculated above is a valid Montgomery curve by Definition 2.1.

Conversely $B \neq 0$ and $A \neq \pm 2$ as in Definition 2.1, so the constants $a$ and $d$ are defined and non-zero. They also cannot be equal due to the conditions on $A$. Therefore $T$ calculated above is a valid twisted Edwards curve as in Definition 2.3.

The quantities $u = \frac{1+y}{1-y}$ and $v = \frac{1+y}{(1-y)x}$ indeed satisfy the Montgomery equation $Bv^2 = u^3 + Au^2 + u$ for all possible elements of $T$ above apart from $y = 1$ and $x = 0$ due to [4]. These exceptions make up finitely many points, and so are permitted in the birational equivalence Definition 2.4.

In the same way, the quantities $x = \frac{u}{v}$ and $y = \frac{(u-1)}{(u+1)}$ satisfy $T$ for all possible elements of $M$ apart from $v = 0$ and $u = -1$, which account for finitely many points.

Finally we may demonstrate:

$$A = \frac{2(a+d)}{a-d} = 2\frac{(A+2)/B + (A-2)/B}{(A+2)/B - (A-2)/B} = 2\frac{2A/B}{4/B} = A,$$
$$B = \frac{4}{a-d} = \frac{4}{(A+2)/B - (A-2)/B} = \frac{4}{4/B} = B.$$

□

### 2.2.3 Twisted Edwards and Weierstrass

First we consider Edwards curves, which are twisted Edwards curves with $a = 1$.

As mentioned above, we can extend the field over which the elliptic curve is defined when it does not have a point of order four[2]. In fact [5]:

**Proposition 2.8.** *An elliptic curve which:*

- *is an abelian group when defined over a finite field $F$ with characteristic $\neq 2$,*
- *has an element of order four,*

[2]If the Weierstrass curve has a point of order two with $x$-coordinate $\alpha$, we can use it to find a point of order four. From [37], division polynomials of order two and four give that $\alpha \pm 1$ are points of order four when $3\alpha \pm 2$ are squares in the field respectively.

*has a* quadratic twist *which is birationally equivalent to an Edwards curve.*

However, transforming Weierstrass curves to Edwards curves can require extending the field $F$, an operation "the cost of which can outweigh any advantages the special forms might otherwise afford"[37]. Note also the caveat of the quadratic twist; we can remove this requirement by adding such a twist to the Edwards curve definition rather than the general elliptic curve.

This is the idea behind twisted Edwards curves, and why conventionally mapping between Weierstrass and Edwards curves is done by composing the maps of Sections 2.2.1 (Weierstrass and Montgomery) and 2.2.2 (Montgomery and Edwards) [4, 51]. For completeness, we describe the direct mapping from an elliptic curve below. We consider an Edwards curve, which satisfies the requirements for Bernstein and Lange's form [5]. It is therefore restricted by $d(1-d) \neq 0$:

$$E = \left\{(X, Y) \in \mathbb{Z}_p^2 \mid X^2 + Y^2 = 1 + dX^2Y^2\right\}. \tag{2.16}$$

Here we define a generic form of elliptic curve:

$$C = \left\{(x, y) \in \mathbb{Z}_p^2 \mid, y^2 = x^3 + a'x^2 + b'x\right\}. \tag{2.17}$$

This form is derived from the *long* Weierstrass form, which is generally not used in cryptography but is transformable through linear changes of variables to the familiar short form [39]. The curve $C$ is defined to apply Proposition 2.8 without too much complexity. Let the point of order four on $C$ be $P = (x_1, y_1)$. From $C$ to $E$ we set:

$$d = 1 - \frac{4x_1^3}{y_1^2}. \tag{2.18}$$

The $x$-coordinate of $P$ cannot be zero so $d \neq 1$ as required. Also $d \neq 0$ because $4x_1^3 = y_1^2$ implies that $y^2 = x(x + x_1)^2$, from [5], which contradicts $C$ being an elliptic curve.

To map the points we must consider quadratic twists of $C$ [5]. Let $C_\alpha$ and $C_\beta$ be defined respectively by:

$$\frac{x_1}{1-d}y^2 = x^3 + a'x^2 + b'x, \tag{2.19}$$

$$\frac{dx_1}{1-d}y^2 = x^3 + a'x^2 + b'x. \tag{2.20}$$

The choice of twists ensures that the $y^2$ coefficient is a square in one of the cases. This is because $d$ must be non-square for the addition to work everywhere on an Edwards curve, so either $\frac{x_1}{1-d}$ is a square and $C$ is isomorphic to $C_\alpha$ or $\frac{dx_1}{1-d}$ is a square and $C$ is isomorphic to $C_\beta$.

By a change of variables, $x \mapsto u = x/x_1$ and $y \mapsto v = y/x_1$, $C_\alpha$ and $C_\beta$ are respectively isomorphic to the curves [5]:

$$\frac{1}{1-d}v^2 = u^3 + \frac{2(1+d)}{1-d}u^2 + u, \tag{2.21}$$

$$\frac{d}{1-d}v^2 = u^3 + \frac{2(1+d)}{1-d}u^2 + u. \tag{2.22}$$

These curves may be mapped to and from the Edwards curve $E$. In the case where $C$ is isomorphic to $C_\alpha$ we map the points on the curve defined by (2.21) to $E$:

$$(0,0) \mapsto (0,-1), \quad (u,v) \mapsto \left(\frac{2u}{v}, \frac{u-1}{u+1}\right). \tag{2.23}$$

From $E$ we map:

$$(0,-1) \mapsto (0,0), \quad (X,Y) \mapsto \left(\frac{1+Y}{1-Y}, \frac{2(1+Y)}{X(1-Y)}\right). \tag{2.24}$$

The first mapping is undefined for points where $v(u+1) = 0$ and the second where $X(1-Y) = 0$, which occurs only for a finite number of points [5]. If the elliptic curve $C$ has a point at infinity it maps to and from $(0,1)$ on $E$. Since both mappings are rational, we have a birational equivalence between $E$ and $C_\alpha$ and therefore $C$. In the other case, $C$ is isomorphic to $C_\beta$ and therefore the curve defined by (2.22). We can

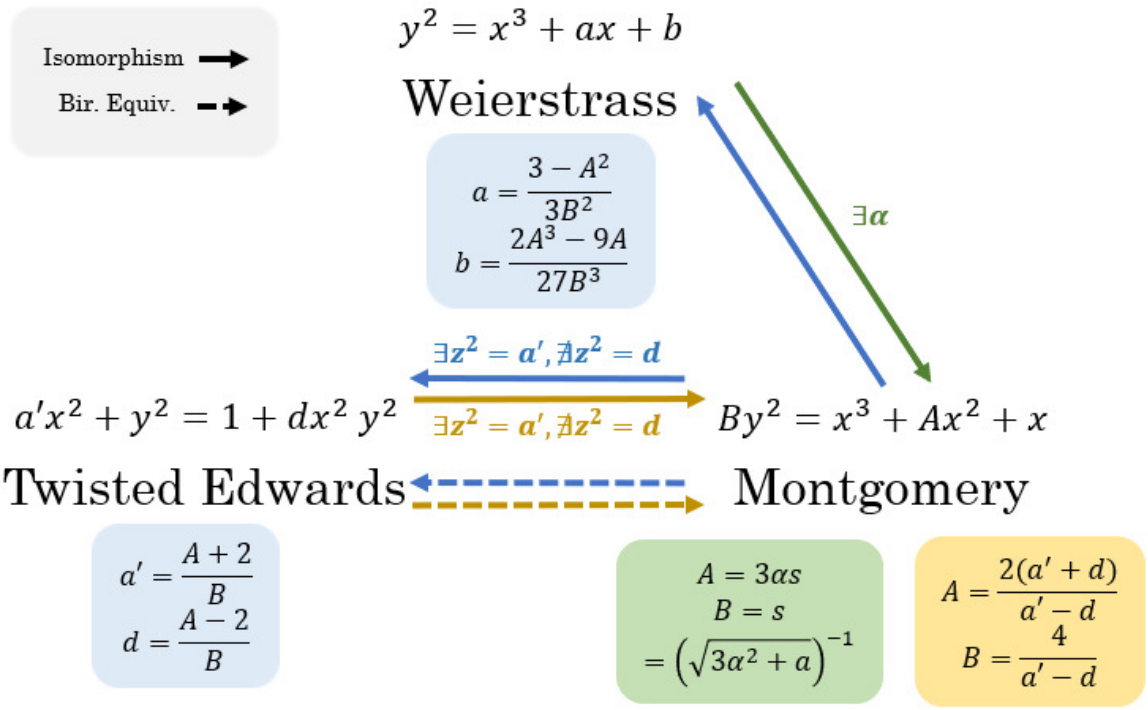



Figure 2.1: Elliptic Curve Transformations.

perform a further change of variables: $u \mapsto -u$ and $d \mapsto 1/d$ in $E$ [5]. These curves are then isomorphic to the original $C$ and $E$ respectively. The same mapping above then applies so that $E$ is birationally equivalent to $C_\beta$ and therefore $C$.

It is hopefully clear why it is conventional to use twisted Edwards curves, composing the mappings of Sections 2.2.1 and 2.2.2 rather than perform the above. Transforming a WNF curve to a Montgomery curve then a twisted Edwards curve saves on algorithmic complexity and is much easier to set up, not to mention that it increases the number of Weierstrass curves which can be tested. This report and the accompanying code therefore uses map composition, which is presented in Figure 2.1.

### 2.2.4 Equivalence of Point Addition

For each of three curve forms, Weierstrass, Montgomery, and twisted Edwards, we now have at least birational equivalences between them. We also have a unique point addition operation per form. To accurately test the speed of operations we must ensure we are computing analogous results. This section demonstrates that, if two points are added on one form of curve their equivalent points on the other forms are added to find an equivalent result. We find algebraically that all point operations on a Montgomery curve and doubling on a twisted Edwards curve are analogous to the same operations on a Weierstrass curve.

In other words, if $P$ and $Q$ are added on a WNF curve to the result $R$, then the equivalent points $P_M$ and $Q_M$ on a Montgomery curve will add to $R_M$, equivalent to $R$. Similarly for a twisted Edwards curve, if $P_T$ is equivalent to $P_M$, then $2P_M$ will be equivalent to $2P_T$.

We assume that the curves satisfy the requirements stated in their Definitions 1.2, 2.3, and 2.1, and that $d$ of a twisted Edwards curve is non square.

#### Montgomery Arithmetic

Recall that for a Montgomery curve:

$$M = \big\{(u, v) \in \mathbb{Z}_p^2 \mid Bv^2 = u^3 + Au^2 + u\big\} \cup \big\{O\big\},$$

we have the binary operation defined by:

$$
\begin{aligned}
P = (u_1, v_1), Q = (u_2, v_2), P, Q \in M :& \\
P + P = (u_S, v_R) = (B\lambda_{dM}^2 - 2u_1 - A, \lambda_{dM}(u_1 - u_S) - v_1), \quad \lambda_{dM} &= \frac{3u_1^2 + 2Au_1 + 1}{2Bv_1}, \\
P + Q = (u_S, v_R) = (B\lambda_{aM}^2 - u_1 - u_2 - A, \lambda_{aM}(u_1 - u_S) - v_1), \quad \lambda_{aM} &= \frac{v_2 - v_1}{u_2 - u_1}, \\
P + (-P) = O.&
\end{aligned}
$$

Similarly for a Weierstrass curve:

$$W = \big\{(x, y) \in \mathbb{Z}_p^2 \mid y^2 = x^3 + ax + b\big\} \cup \big\{O\big\},$$

we have:

$$
\begin{aligned}
P = (x_1, y_1), Q = (x_2, y_2) :& \\
P + P = (x_S, y_R) = (\lambda_{dW}^2 - 2x_1, \lambda_{dW}(x_1 - x_S) - y_1), \quad \lambda_{dW} &= \frac{3x_1^2 + a}{2y_1}, \\
P + Q = (x_S, y_R) = (\lambda_{aW}^2 - x_1 - x_2, \lambda_{aW}(x_1 - x_S) - y_1), \quad \lambda_{aW} &= \frac{y_2 - y_1}{x_2 - x_1}, \\
P + (-P) = O.&
\end{aligned}
$$

Since for both forms the inverse is found by multiplying the $y$-coordinate by -1 and the identity is a single point $O$, it is clear $P + (-P) = O$ is equivalent for any $P$. As to point doubling, we use the mappings in Section 2.2.1 to substitute thus:

$$
\begin{aligned}
\lambda_{dW} = \frac{3x_1^2 + a}{2y_1} &= \frac{3x_1^2 + \frac{3-A^2}{3B^2}}{2y_1} = \frac{3(\frac{u_1}{B} + \frac{A}{3B})^2 + \frac{3-A^2}{3B^2}}{2v_1/B} \\
&= \frac{\frac{3}{B^2}(u_1 + \frac{A}{3})^2 + \frac{3-A^2}{3B^2}}{2v_1/B} = \frac{3(u_1 + \frac{A}{3})^2 + \frac{3-A^2}{3}}{2Bv_1} \\
&= \frac{3u_1^2 + 2Au_1 + \frac{A^2}{3} + \frac{3-A^2}{3}}{2Bv_1} = \frac{3u_1^2 + 2Au_1 + 1}{2Bv_1} = \lambda_{dM}. \qquad (2.25)
\end{aligned}
$$

We can now find that:

$$
\begin{aligned}
P + P = (x_S, y_R) &= (\lambda_{dW}^2 - 2x_1, \lambda_{dW}(x_1 - x_S) - y_1), \\
x_S = \frac{u_S}{B} + \frac{A}{3B} &= \lambda_{dM}^2 - 2\left(\frac{u_1}{B} + \frac{A}{3B}\right) = \frac{3B\lambda_{dM}^2 - 6u_1 - 2A}{3B}, \\
\implies 3u_S &= 3B\lambda_{dM}^2 - 6u_1 - 2A - A, \\
u_S &= B\lambda_{dM}^2 - 2u_1 - A,
\end{aligned}
$$

as required. Similarly for addition:

$$\begin{aligned}\lambda_{aW} &= \frac{y_2 - y_1}{x_2 - x_1} = \frac{\frac{v_2}{B} - \frac{v_1}{B}}{(\frac{u_2}{B} + \frac{A}{3B}) - (\frac{u_1}{B} + \frac{A}{3B})} \\ &= \frac{v_2 - v_1}{u_2 - u_1} = \lambda_{aM}.\end{aligned}$$

Now we show that $x_S$ is equivalent to $u_S$:

$$\begin{aligned}P + Q = (x_S, y_R) &= (\lambda_{aW}^2 - x_1 - x_2, \lambda_{aW}(x_1 - x_S) - y_1), \\ x_S = \frac{u_S}{B} + \frac{A}{3B} &= \lambda_{aM}^2 - \left(\frac{u_1}{B} + \frac{A}{3B}\right) - \left(\frac{u_2}{B} + \frac{A}{3B}\right) \\ &= \frac{B\lambda_{aM}^2 - u_1 - u_2}{B} - \frac{2A}{3B}, \\ \implies u_S &= B\lambda_{dM}^2 - u_1 - u_2 - A. \end{aligned} \tag{2.26}$$

For the $y$-coordinate result of both doubling and general addition:

$$\begin{aligned}y_R = \frac{v_R}{B} &= \lambda_W(x_1 - x_S) - y_1 = \lambda_M\left(\frac{u_1}{B} + \frac{A}{3B} - \frac{u_S}{B} - \frac{A}{3B}\right) - \frac{v_1}{B} \\ &= \frac{\lambda_M}{B}(u_1 - u_S) - \frac{v_1}{B}, \\ \implies u_R &= \lambda_M(u_1 - u_S) - v_1. \end{aligned} \tag{2.27}$$

Note that we may substitute $\lambda_{dW}$ or $\lambda_{aW}$ for $\lambda_W$, similarly for $\lambda_M$, since the formula for the $y$-coordinate is the same in each form and we have shown that the respective $\lambda$s are equivalent.

Demonstrating the operations from $M$ to $W$ gives the same results as above, since we have simply performed a linear change of variables where the mappings in Section 2.2.1 are inverses.

### Twisted Edwards Arithmetic

Recall that on a twisted Edwards curve $T$:

$$T = \left\{(x, y) \in \mathbb{Z}_p^2 \mid ax^2 + y^2 = 1 + dx^2y^2\right\},$$

the inverse of the point $(x, y)$ is $(-x, y)$. We can set $c = 1$ since we are considering *twisted* Edwards curves and their correspondence with Montgomery curves, which always have a point of order four (see Section 2.2.2). We assume $a$ is a square in the field and $d$ is not. The exceptional points we encounter otherwise are mapped to points of

infinity. Our point operation on $T$ is described by:

$$\begin{aligned} P = (x_1, y_1), Q = (x_2, y_2) : \\ P + Q = (x_R, y_R) = \left( \frac{x_1 y_2 + y_1 x_2}{1 + d x_1 x_2 y_1 y_2} = (-x_S, y_S), \frac{y_1 y_2 - a x_1 x_2}{1 - d x_1 x_2 y_1 y_2} \right), \\ P + (-P) = (0, 1). \end{aligned}$$

We can see algebraically that since $-P = (-x_1, y_1)$, $P + (-P)$ yields the $y$-coordinate of the solution as $1 = (y_1^2 - a x_1^2)/(1 - d x^2 y^2)$. From the definition of $T$, these quantities are equal, so we have $P + (-P) = (0, 1)$. This point is mapped to and from the point at infinity $O$ on $M$. For the points on $M$ which satisfy $y = 0$ or $x = -1$, we have defined the mapping in Section 2.2.2 either directly or as points of infinity on $T$, so we do not consider them here.

Twisted Edwards curves need no slope in the operation formulae, so we must first convert the $\lambda$ values for $M$. We have:

$$\begin{aligned} \lambda_{dM} &= \frac{3u_1^2 + 2u_1 + 1}{2Bv_1} = \frac{3u_1^2 + 2\frac{2(a+d)}{a-d}u_1 + 1}{2\frac{4}{a-d}v_1} \\ &= \frac{3(\frac{1+y_1}{1-y_1})^2 + 2\frac{2(a+d)}{a-d}\frac{1+y_1}{1-y_1} + 1}{2\frac{4}{a-d}\frac{1+y_1}{x_1(1-y_1)}} \\ &= \frac{-2dxy^2 + axy - dxy + 2ax}{2(1 - y^2)}. \end{aligned} \tag{2.28}$$

Note that from Section 2.2.2, $v_1 = \frac{u_1}{x_1}$. Point doubling on $T$ is calculated:

$$P + P = (x_R, y_R) = (-x_S, y_S) = \left( \frac{2x_1 y_1}{1 + d x_1^2 y_1^2}, \frac{y_1^2 - a x_1^2}{1 - d x_1^2 y_1^2} \right). \tag{2.29}$$

It is more difficult here simply to replace variables with their corresponding transformations [39]. Instead we take an indirect but faster route. In the case of doubling, $P + P = (u_S, v_R) = (u_S, -v_S)$ on $M$, so from the operations:

$$\begin{aligned} v_R &= \lambda_{dM}(u_1 - u_S) - v_1, \\ v_R = -v_S = \frac{-u_S}{-x_R} &= \lambda_{dM}(u_1 - u_S) - \frac{u_1}{x_1}, \\ \frac{u_1}{x_1} + \frac{u_S}{x_R} &= \lambda_{dM}(u_1 - u_S). \end{aligned} \tag{2.30}$$

Where the coordinates are given by:

$$u_1 = \frac{1+y_1}{1-y_1}, \tag{2.31}$$

$$u_S = \frac{1+y_S}{1-y_S} = \frac{1-dx_1^2y_1^2-ax_1^2+y_1^2}{1-dx_1^2y_1^2+ax_1^2-y_1^2} = \frac{1-ax_1^2}{1-y_1^2}, \tag{2.32}$$

$$\implies \frac{u_S}{x_R} = \frac{1-ax_1^2}{1-y_1^2} \cdot \frac{1+dx_1^2y_1^2}{2x_1y_1}. \tag{2.33}$$

With everything in terms of the parameters of $T$ and the coordinates of $P$, we can check if point doubling on the twisted Edwards curve satisfies (2.30).

$$\begin{aligned}
\frac{u_1}{x_1} + \frac{u_S}{x_R} &= \frac{1+y_1}{x_1(1-y_1)} + \frac{1-ax_1^2}{1-y_1^2} \cdot \frac{1+dx_1^2y_1^2}{2x_1y_1} \\
&= \frac{2y_1(1+y_1)^2 + (1-ax_1^2)(1+dx_1^2y_1^2)}{2x_1y_1(1-y_1^2)} \\
&= \frac{2y_1(1+y_1)^2 - adx_1^4y_1^2 + \mathbf{dx_1^2y_1^2 - ax_1^2 + 1}}{2x_1y_1(1-y_1^2)} \\
&= \frac{2y_1(1+y_1)^2 - adx_1^4y_1^2 + \mathbf{y_1^2}}{2x_1y_1(1-y_1^2)} \\
&= \frac{2(1+y_1)^2 - adx_1^4y_1 + y_1}{2x_1(1-y_1^2)}.
\end{aligned} \tag{2.34}$$

We have the left hand side of (2.30). What remains is to show the right hand side is equal:

$$\begin{aligned}
\lambda_{dM}(u_1 - u_S) &= \frac{-2dx_1y_1^2 + ax_1y_1 - dx_1y_1 + 2ax_1}{2(1-y_1^2)} \left( \frac{1+y_1}{1-y_1} - \frac{1-ax_1^2}{1-y_1^2} \right) \\
&= \frac{-2dx_1y_1^2 + ax_1y_1 - dx_1y_1 + 2ax_1}{2(1-y_1^2)} \left( \frac{(1+y_1)^2 - 1 + ax_1^2}{1-y_1^2} \right) \\
&= \frac{(-2dx_1y_1^2 + ax_1y_1 - dx_1y_1 + 2ax_1) \cdot (y_1^2 + 2y_1 + ax_1^2)}{2(1-y_1^2)^2}.
\end{aligned}$$

The numerator of this expression is expanded to:

$$a^2x_1^3y_1 + 2a^2x_1^3 - 2adx_1^3y_1^2 - adx_1^3y_1 + ax_1y_1^3 + 4ax_1y_1^2 + 4ax_1y_1 - 2dx_1y_1^4 - 5dx_1y_1^3 - 2dx_1y_1^2.$$

We multiply by $x_1$ and collect terms:

$$\begin{aligned}\lambda_{dM}(u_1 - u_S) &= \frac{5y_1(ax_1^2 - dx_1^2y_1^2) - adx_1^4y_1(1-y_1^2) + 2(ax_1^2 - dx_1^2y_1^2)(1+y_1^2)}{2x_1(1-y_1^2)^2} \\ &= \frac{5y_1(1-y_1^2) - adx_1^4y_1(1-y_1^2) + 2(1-y_1^2)(1+y_1^2)}{2x_1(1-y_1^2)^2} \\ &= \frac{5y_1 - adx_1^4y_1 + 2 + 2y_1^2}{2x_1(1-y_1^2)} \\ &= \frac{2(1+y_1)^2 - adx_1^4y_1 + y_1}{2x_1(1-y_1^2)} = \frac{u_1}{x_1} + \frac{u_S}{x_R}, \end{aligned} \tag{2.35}$$

as required. Both coordinates satisfy (2.30) if and only if $(u_S, v_R) = (u_S, -v_S)$ is a solution to $P + P$, so we are done [39].

As to general point addition, the derivation is much more complicated (as the author has discovered). Similarly to above we must show that:

$$\frac{u_1}{x_1} + \frac{u_S}{x_R} = \lambda_{aM}(u_1 - u_S). \tag{2.36}$$

After some algebra, this leads to showing that:

$$-x_1^2y_1y_2 + ax_1x_2y_1y_2 + dx_1^2y_1^3y_2^3 - x_1x_2y_1^2 + ax_2^2y_1^2 + dx_1x_2y_1^4y_2^2 \tag{2.37}$$

is equal to:

$$\begin{aligned}&(-y_1^2 + y_1y_2 - y_1^3y_2 + y_1^2y_2^2 + ax_1x_2y_1^2 - ax_1x_2y_1y_2 + dx_1x_2y_1^3y_2 - dx_1x_2y_1^2y_2^2) \\ &\cdot (1 + dx_1x_2y_1y_2). \end{aligned} \tag{2.38}$$

Using the definition of $T$, we may multiply by $ax^2 + y^2 - dx^2y^2 = 1$ anywhere, as it does not change the value of any terms. Therefore a script in an algebra program (Sage or Mathematica, for instance) may be preferable to continue it.

The algebra up to this point is shown in Appendix B.

## 2.3 More Arithmetic and Finite Fields

All the curve and point mappings are calculated modulo a large prime $p$. This adds considerable complexity in implementation as, for instance, when we write $\frac{1+y}{1-y}$ we mean the modular inverse of $1 - y$ multiplied by $1 + y$.

A particularly hard problem is finding roots of integers. We require finding square roots modulo $p$, for Montgomery mappings (Sections 2.2.2 and 2.2.1), and cube roots modulo $p$, for finding solutions to the Weierstrass cubic when $y^2 = 0$ (Section 2.2.1).

As $p$ is prime and very large, manually checking for solutions is not a viable option.

While the DLP (Definition 1.8) asks to find $x$ in $a^x = y$ (for a finite group), the above problems ask to find $a$ when $x = 2$ or 3 and $y$ is known in $\mathbb{Z}_p$. There is no known efficient algorithm for finding such an $a$ when $p$ is not prime. However, when $p$ *is* prime there are polynomial time approaches, including a deterministic algorithm for constructing points on a WNF curve (which requires finding the square root of $y^2$) [45]. However, it is somewhat out of scope of this project. We outline more comprehensive methods below.

### 2.3.1 Square Roots

The *Legendre Symbol* equals $-1 \mod p$ when the element does not have a square root and 1 if it does [23, p. 43]. It is calculated:

$$x^{\frac{p-1}{2}} \mod p. \tag{2.39}$$

This allows us to simplify finding square roots further. When $p = -1 \mod 4$, we can find the square root of $x$ with a single line [23, p. 49]:

$$a = x^{\frac{p+1}{4}} \mod p. \tag{2.40}$$

Since $p + 1 = 0 \mod 4$, we know $\frac{p+1}{4}$ and $\frac{p+1}{2}$ are integers in $\mathbb{Z}_p$. We can then show (omitting $\mod p$):

$$\begin{aligned} a^2 &= x^{2\frac{p+1}{4}} = x^{\frac{p+1}{2}}, \\ &= x^{1+\frac{p-1}{2}} = x \cdot x^{\frac{p-1}{2}} = x, \end{aligned} \tag{2.41}$$

since $x^{\frac{p-1}{2}}$ is the Legendre symbol, which must equal 1 for $a$ to exist.

When $p = 1 \mod 4$, we can use the *Tonelli-Shanks algorithm*, a probabilistic polynomial time method. The below explanation is due to Koblitz with the same inputs as above [24, p. 128].

Since $p$ is odd, $p-1$ is even and therefore in the form $2^s t$ where $t$ is odd. We suppose that $s$ and $t$ are greater than 1, which encompasses most cases, especially with large $p$.

We choose a random integer $u \in \mathbb{Z}_p$ which is a non-square. We check this with the Legendre Symbol. We then let $v = u^t$, so that $v^{2^s} = u^{2^s t} = 1$. Our first guess for $a = \sqrt{x}$ is given by:

$$a_1 = x^{\frac{t+1}{2}}, \implies a_1^2 = x \cdot x^t, \tag{2.42}$$

where $x^{2^{s-1}t} = 1$. Therefore there must exist some $m$ such that:

$$\begin{aligned} a &= a_1 v^m = \sqrt{x}, \\ &= x^{\frac{t+1}{2}} v^m \implies v^{2m} = x^t = a_1 x^{-1}. \end{aligned} \tag{2.43}$$

We find $m$ in terms of successive powers of two i.e. binary digits. Let $m = m_0 2^0 + m_1 2^1 + m_2 2^2 + ... + m_{s-2} 2^{s-2}$. Note that $m_0 = 0$ (i.e. $m$ is even) if and only if [24, p. 128]:

$$\frac{a_1^{2^{s-2}}}{x^{2^{s-2}}} = v^{2 \cdot 2^{s-2} m} = v^{2^{s-1} m} = x^{2^{s-2} t} = 1. \tag{2.44}$$

Similarly, $m_1 = 0$ if and only if we raise $x^t$ to the power of $2^{s-3}$ and find 1. We continue this way until we find all $m_i$ which satisfy (2.43). Our solution is

$$a = \pm a_1 v^m \mod p.$$

It is not immediately clear what this algorithm does, so we visit a small example. Say we want to find $\sqrt{10} \mod 13$.

We have $p - 1 = 12$, so $p - 1 = 2^2 3$, giving $s = 2$ and $t = 3$, both greater than 1 as required. Now we must choose a random non-square $u$. Here let us choose 5, so $v = 5^3 \mod 13 = 8$.

The 'first guess' $a_1$ is given by $x^{\frac{t+1}{2}} = 10^2 \mod 13 = 9$. We now find successive digits of $m$ in binary. Since $s = 2$, $m$ has only digit: $m_0 2^0$. We calculate:

$$x^{2^{s-2}t} = x^{2^0 3} = 10^3 \mod 13 = 12.$$

The result $\neq 1$, so $m_0 = m = 1$. We can finally work out $\sqrt{10} = \pm a_1 v^m = \pm 9 \cdot 8^1 = \pm 7 \mod 13$.

### 2.3.2 Cube Roots

As to cube roots, $a^3 = x$, a short solution can be found when $p = -1 \mod 3$:

$$a = x^{\frac{2p-1}{3}} \mod p. \tag{2.45}$$

This is because 3 divides $2p - 1$, so:

$$a^3 = x^{2p-1} = x^{-1} x^2 = x. \tag{2.46}$$

Otherwise $p = 1 \mod 3$ and a third of the non zero elements in $\mathbb{Z}_p$ have three distinct cube roots and the remaining two thirds have none. Peter Cameron has devised the following probabilistic algorithm to find cube roots when $p = 1 \mod 3$ and $p \neq 1 \mod 9$.

We know that $p - 1$ is divisible by 3, so the elements with a cube root form a subgroup of order $\frac{p-1}{3}$ (we minus one from $p$ for the 0 element). Therefore,

$$x^{\frac{p-1}{3}} = 1 \mod p$$

if and only if $x$ has a unique cube root. Consider this the 'Legendre Symbol for cube roots' subject to the above conditions on $p$. That 9 does not divide $p - 1$ eliminates multiple roots.

First, we choose an integer $k$ such that $3k \mod \frac{p-1}{3} = 1$. If $p \mod 9 = 4$, we can set $k = (2p + 1)/9$. If $p \mod 9 = 7$, then $k = (p + 2)/9$ also works. Otherwise, we try random values of $k$.

Then $a = x^k$ since:

$$a^3 = x^{3k} = x^{m\frac{p-1}{3}} x^1 = x, \tag{2.47}$$

for some integer $m$, as we are repeatedly multiplying $x$ where it is in a subgroup of order $\frac{p-1}{3}$.

To summarise, cube roots always exist for $p = -1 \mod 3$ and exist for a third of the elements of $\mathbb{Z}_p$ for $p = 1 \mod 3$. However, For $p = -1 \mod 4$, square roots exist for approximately half of the elements of $\mathbb{Z}_p$. The elements with a Legendre Symbol equalling 1 are that 'half' which have a square root.

These algorithms and 'shortcuts' are implemented in the following code.

# Chapter 3

# Implementation

In this chapter we describe and present a working implementation of the curve mappings from Section 2.2 and the curve addition operations for Montgomery, twisted Edwards, and Weierstrass curves from Section 2.1. As an extension of the simulation given in *Mathematics of Bitcoin* ([53]), initial functions for curve mappings were created in Matlab. These functions are given in Appendix A.1, where all the written code is new unless otherwise stated.

Further functions were either converted to or built with Python so large integers can be used. This allows us to simulate ECC with parameters used in real world encryption.

## 3.1 Previous Matlab Code

From *Mathematics of Bitcoin*, we have point addition and curve generation on Weierstrass form elliptic curves. Due to Matlab's limitations, we cannot use a realistically sized curve without extra packages[1]. However, Matlab can be particularly useful to generate and transform small curves, in which we can output each point for testing the functions.

To this end the functions in Appendix A.1 perform the following transformations:

- `WtoM` converts a Weierstrass form curve with parameters $a$ and $b$ of prime base $p$ to a Montgomery curve, outputting all points and the new parameters $A$ and $B$.
- `MtoW` performs the inverse operation, taking a Montgomery curve with $A$ and $B$ and outputting a Weierstrass curve, with all points parameters $a$ and $b$.
- `EtoM` converts a twisted Edwards form curve with parameters $a$ and $d$ to a Montgomery curve with $A$ and $B$ and all its points.

[1]The author previously installed the VPI package to allow for the use of real Bitcoin parameters [53]. In this project we can work with large integers in the more widely used Python.

- `MtoE` transforms a Montgomery curve to a twisted Edwards curve, taking $A$ and $B$ to output $a$ and $d$ and the mapped points.
- `WtoE` transforms a Weierstrass curve to a twisted Edwards curve by composing the code from `WtoM` and `MtoE`.
- `EtoW` similarly composes `EtoM` and `MtoW` to transform a twisted Edwards curve to a Weierstrass curve.

These core functions make use of `EllCW`, `EllCM`, and `EllCE` which output all points on a given Weierstrass, Montgomery, or twisted Edwards curve respectively with a prime base $p$. They simply check for solutions for all values possible in the finite field the curve is being defined over. The point at infinity, $O$, is represented by `inf`.

In some cases the functions call `subgsize`, which is taken from *Mathematics of Bitcoin* to output each point on a given Weierstrass curve and its order. This is useful particularly for finding points of order two. Clearly a modular inverse function is also required, here it is called `mminv` and uses Matlab's greatest common divisor function.

There is also an option written into these functions such that the user may choose to plot the curve of a graph or check if the completed transformation is an isomorphism, using the requirements in Section 2.2.

### 3.1.1 Example

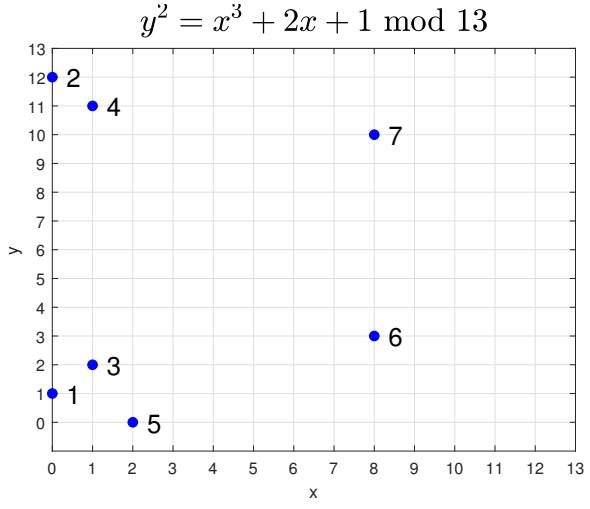
These Matlab functions are used in the following small example. We demonstrate the mappings between Weierstrass, Montgomery, and twisted Edwards curves with a nicely behaved curve. In this example, take all $(x, y) \in \mathbb{Z}_p^2$ where $p = 13$. We label each point to show the mappings between them, i.e. point 3 on $W$ is mapped to point 3 on $T$.

Starting with the Weierstrass form, we have:

$$W = \{y^2 = x^3 + 2x + 1\} \cup \{O_W\}. \tag{3.1}$$

Using `EllCW`, $W$ has points:

| label | x | y |
|---|---|---|
| 1 | 0 | 1 |
| 2 | 0 | 12 |
| 3 | 1 | 2 |
| 4 | 1 | 11 |
| 5 | 2 | 0 |
| 6 | 8 | 3 |
| 7 | 8 | 10 |
| 8 | O | |

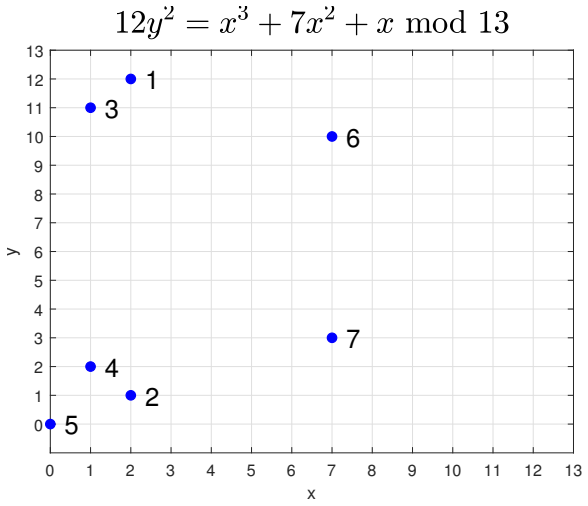


Since this curve has a point of order two, $(2, 0)$, it can be mapped to a Montgomery curve. Following the process in Section 2.2.1, we use `WtoM`:

$$M = \{12y^2 = x^3 + 7x^2 + x\} \cup \{O_M\}, \tag{3.2}$$

The points of $W$ are mapped to the points of $M$:

| label | x | y |
|---|---|---|
| 1 | 2 | 12 |
| 2 | 2 | 1 |
| 3 | 1 | 11 |
| 4 | 1 | 2 |
| 5 | 0 | 0 |
| 6 | 7 | 10 |
| 7 | 7 | 3 |
| 8 | O | |

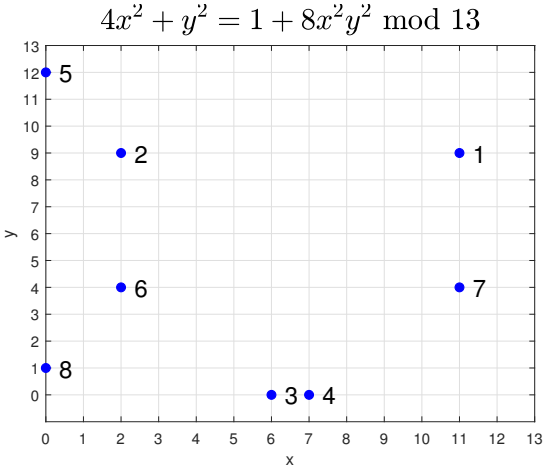



We then map to a Twisted Edwards curve as in Section 2.2.2 using `MtoE`.

$$T = \{4x^2 + y^2 = 1 + 8x^2y^2\}. \tag{3.3}$$

In $\mathbb{Z}_{13}$, 4 is a square and 8 is not, so the mapping is isomorphic. Since all mappings between Weierstrass and Montgomery curves are isomorphic, all three of our curves are isomorphic. The curve $T$ has points:

| label | x | y |
|---|---|---|
| 1 | 11 | 9 |
| 2 | 2 | 9 |
| 3 | 6 | 0 |
| 4 | 7 | 0 |
| 5 | 0 | 12 |
| 6 | 2 | 4 |
| 7 | 11 | 4 |
| 8 | 0 | 1 |




### 3.1.2 Conversion to Python

Both Python and Matlab are functional languages, so creating Python code which does the same thing as some Matlab code is not too difficult. However, if the reader is not completely familiar with Python there are a few caveats to note.

In Matlab it is very clear when a number is calculated modulo $p$. In Python, the

notation is much more succinct; `a % p` means $a \mod p$. To divide two integers, we use two slashes, `//`, and for multiplication we only use one star, `*`. Two stars, `**`, represents raising to a power. If we wish to raise a number to a certain power and compute it modulo $p$, we can instead use the `pow` function, where $a^{b/c} \mod p$ is performed with `pow(a, b//c, p)`. In Python, we need not define a modular inverse function, since there is one built in, which is imported at the top of each file.

As we can work with large integers in core Python, the functions are defined for use for any sized curve; from small testing curves to actual cryptographic size. This project uses values of $p$ up to nearly $2^{256}$. So, we no longer produce a list of points. Instead, there are separate functions for mapping individual points and mapping curve parameters. We also define point operations for all curves to complete the cryptographic algorithms. All the functions described below can be dound in Appendix A.2.

## 3.2 Python Outline

As before, we have the mapping functions between curve types. However here they are much shorter, as they just convert the parameters:

- `wtom` converts Weierstrass to Montgomery.
- `mtow` converts Montgomery to Weierstrass.
- `etom` converts twisted Edwards to Montgomery.
- `mtoe` converts Montgomery to twisted Edwards.
- `wtoe` converts Weierstrass to twisted Edwards curve by composing the code from `wtom` and `mtoe`.
- `etow` converts twisted Edwards to Weierstrass by composing `etom` and `mtoe`.

We now do not perform brute force calculations, like checking all possible values for a square root, so we require some extra functions:

- `legendres` finds the Legendre Symbol of an integer modulo prime $p$.
- `modroot` calculates the modular square root following the methods in Section 2.3.1.
- `tonellishanks` performs the Tonelli Shanks algorithm and is called only when `modroot` computes a square where the prime $p = 1 \mod 4$.
- `modcube` calculates the modular cube root using the methods in Section 2.3.2.

- `w2check` finds a point of order two on a Weierstrass curve, if it exists.

All algorithms except `w2check` have been described already in this paper. To find a point of order two on Weierstrass curve, we must find a solution to $x^3+ax+b=0$. The Python function uses a variant of Tartaglia's formula. As described in [34] we have:

$$\alpha = \sqrt[3]{\frac{-b}{2}+\sqrt{\frac{b^2}{4}+\frac{a^3}{27}}}+\sqrt[3]{\frac{-b}{2}-\sqrt{\frac{b^2}{4}+\frac{a^3}{27}}}. \tag{3.4}$$

The function `w2check` implements this by first calculating:

$$r = \frac{b^2}{4}+\frac{a^3}{27},$$

then finding its square root, $rt$, if it exists. It then finds $bf=-b/2$ and calculates:

$$\alpha = \sqrt[3]{bf+rt}+\sqrt[3]{bf-rt}.$$

This, of course, only finds one root of the equation, but that is all we need. The point of order two is given by $(\alpha, 0)$.

We may now introduce the point operation and mapping functions. Again, the point at infinity is represented by `inf`. Note that, for instance, a point at infinity of order two is represented by $(\texttt{inf}, 2)$. If we are mapping a Montgomery curve to a twisted Edwards curve and require two points at infinity of order two (recall from Section 2.2.2) they are represented by $(\texttt{inf}, 2)$ and $(\texttt{inf}, -2)$ to differentiate them.

The point mapping operations below take the coordinates of a point, curve parameters, and the prime base $p$ as input following the mappings in Section 2.2:

- `pwtom` maps a point on a Weierstrass curve to the equivalent point on a Montgomery curve. The user can choose whether to input Montgomery parameters if they are known, or leave them out and the function will calculate them.

- `pmtow` maps a point on a Montgomery curve to the equivalent point on a Weierstrass curve.

- `petom` maps a point on a twisted Edwards curve to the equivalent point on a Montgomery curve. It takes into account all the possible points at infinity the twisted Edwards curve may have.

- `pmtoe` maps a point on a Montgomery curve to the equivalent point on a twisted Edwards curve. It, too, returns any of the possible points at infinity the Montgomery point may be mapped to.

- `pwtoe` maps a point on a Weierstrass curve to the equivalent point on a twisted Edwards curve by composing `pwtom` and `pmtoe`.
- `petow` maps a point on a twisted Edwards curve to the equivalent point on a Weierstrass curve, also composing `petom` and `pmtow`.

All of the functions above verify that the input point exists on the given curve, and tests requirements for curve transformation. Finally, from Section 2.1, we have functions performing point addition for each curve:

- `wadd` performs point addition on a Weierstrass curve, converted from the Matlab addition function used in [53].
- `madd` performs point addition on a Montgomery curve.
- `eadd` performs point addition on a twisted Edwards curve.

Recall that point multiplication, the operation needed for key generation, is simply repeated addition $k$ times, for a scalar $k$. However, it is not particularly efficient to repeat the addition $k$ times. Instead we use the *Double and Add Algorithm*, inputting the point $P$ and multiplier $k$ in binary with $l$ denoting the number of digits. As described in [53]:

**Algorithm 1:** Double and Add

$Q \leftarrow O$
**for** $i \leftarrow l$ **to** $0$ **do**
  double $Q$
  **if** $k_i = 1$ **then**
    $Q \leftarrow Q + P$
  **end**
**end**
return $Q$

For instance, $12 = 2^2 3$, so instead of adding a point $P$ to itself 12 times we double it twice, finding $4P$, then add to itself three times, finding $12P$ in 5 steps. This algorithm is implemented for the point multiplication functions:

- `wmul` performs point multiplication on a Weierstrass curve.
- `mmul` performs point multiplication on a Montgomery curve.
- `emul` performs point multiplication on a twisted Edwards curve.

These functions do not contain plotting options, since generating and storing each point on a realistically sized curve takes far too much memory. However, all the curve mapping functions automatically determine whether the transformation is an isomorphism.

### 3.2.1 Example

We illustrate the Python functions with an example, beginning with curve and point mappings. The Internet Research Task Force has published two secure elliptic curves for real cryptographic use [29]. The curves are accompanied with some transformations which allow us to verify the functions work. Again, here take $(x, y) \in \mathbb{Z}_p^2$.

Consider `Curve25519`, originally put forward for ECC by Bernstein in 2006 [3]. Its Montgomery curve parameters are:

$$\begin{aligned} M &= \{y^2 = x^3 + 486662x^2 + x\} \cup \{O_M\}, \\ p &= 2^{255} - 19. \end{aligned}$$

We use the Python function `mtoe` to transform this curve to twisted Edwards form. We find $M$ is isomorphic to:

$$T = \{486664x^2 + y^2 = 1 + 486660x^2y^2\}.$$

The mapping is an isomorphism since $a = 486664$ is a square and $d = 486660$ is not.

Note that the publication ([29]) actually states that `Curve25519` is *birationally equivalent* to another twisted Edwards curve, `ed25519`, given by:

$$\begin{aligned} E &= \{-x^2 + y^2 = 1 + dx^2y^2\}, \\ d =& 37095705934669439343138083508754565189542113798432190163887855330859402835 55. \end{aligned}$$

The calculations are not incorrect, as in fact Langley et al. have transformed the output $T$ to a curve with $a = -1$. This transformation is a birational equivalence and commonly performed in cryptography to simplify parameters [51]. Though it is not stated, we can deduce that this is the following mapping on $T$:

$$x \mapsto \frac{x}{\sqrt{-486664}}, \qquad \implies 486660x^2y^2 \mapsto -\frac{486660}{486664}x^2y^2 = -\frac{121665}{121666}x^2y^2.$$

We compute -121665 multiplied by the modular inverse of 121666 to find the same $d$ used in $E$ [29]:

$$\begin{aligned} d =& 37095705934669439343138083508754565189542113798432190163887855330859402835 55. \end{aligned}$$

Recall that key exchange using elliptic curves requires some generator point, $G$, as described following Definition 1.12. The given generator must produce a subgroup,

$\langle G \rangle$, with a large order $n$, so it is often published with other ECC parameters. The point used with `Curve25519` is:

$$\begin{aligned} G_M &= (9, y_{GM}), \\ y_{GM} &= 14781619447589544791020593568409986887264606134616475 \\ &\quad 28896488183775558623740 1. \end{aligned}$$

The function `pmtoe` used on $G_M$ gives the equivalent point on the twisted Edwards curve $T$:

$$\begin{aligned} G_T &= (x_{GT}, y_{GT}), \\ x_{GT} &= 38213832894368730265794714087330135568483813637251082 4 \\ &\quad 00757400312561599933396, \\ y_{GT} &= 46316835694926478169428394003475163141307993866256225 6 \\ &\quad 15783033603165251855960. \end{aligned}$$

The equivalent generator $G_E$ on `ed25519` ($E$) given by Langley et al. is [29]:

$$\begin{aligned} G_E &= (x_{GE}, y_{GE}), \\ x_{GE} &= 15112221349535400772501151409588531511454012693041857206046113283949847762202, \\ y_{GE} &= 46316835694926478169428394003475163141307993866256225 6 \\ &\quad 15783033603165251855960. \end{aligned}$$

Following the mapping above, we must multiply $x_{GT}$ by $\sqrt{-486664} \mod p$. Using `modroot` we find that:

$$\begin{aligned} \sqrt{-486664} \mod p &= \pm 5104256939916053613020613523314632928415220225303463182268183378866687721520 7, \\ \implies x_{GE} &= \sqrt{-486664} \cdot x_{GT} \mod p \\ &= 15112221349535400772501151409588531511454012693041857206046113283949847762202, \end{aligned}$$

as required. Since we do not transform the $y$-coordinate, $y_{GT} = y_{GE}$.

Recall the Diffie-Hellman method for key generation from the introduction. Say Alice wants to establish keys to communicate private messages with Bob over an insecure channel. The Internet Research Task Force gives a test set of possible keys [29]. How-

ever, their function codes the output slightly differently, so for this part of the example just the private key values are used. We start with Alice's randomly chosen private key:

$$d_A = 538383494829344682498082851209911065601763461286791508587295017959562187503517 8.$$

Her public key is calculated with $d_A G_M$ on the Montgomery curve `Curve25519`. Using `mmul` we discover it is given by:

$$P_{AM} = (461516306948643042639601548469017143598281787390621171590252776472615584412, 13070594169239478115446582128550067480452326318343516314474879361527799519490).$$

We replicate this multiplication on the twisted Edwards curve $T$ using `emul`, finding:

$$P_{AT} = (163844540449376546400780968842350101318716342152828591859326731767089675565005710204791229032130762366049463729017453772598980045724866897062908603925513 9).$$

So we have two points, one on $M$ which satisfies $P_{AM} = d_A G_M$ and one on $T$ which satisfies $P_{AT} = d_A G_T$. We know that $G_M$ and $G_T$ are equivalent points in terms of the isomorphic mapping between the curves and $d_A$ is just an integer. Therefore, $P_{AM}$ and $P_{AT}$ should also be equivalent. We check this by inputting $P_{AM}$ in the point mapping function `mtoe`:

$$\begin{aligned} &\texttt{mtoe} : P_M \in M \mapsto P_T \in T, \\ \texttt{mtoe}(P_{AM}) &= (163844540449376546400780968842350101318716342152828591859326731767089675565005710204791229032130762366049463729017453772598980045724866897062908603925513 9), \\ &= P_{AT}. \end{aligned}$$

For completeness, `Curve25519` is transformed into a Weierstrass curve $W$, where we

compute Alice's public and private keys:

$$\begin{aligned}
W =& \{y^2 = x^3 + ax + b\},\\
a =& 19298681539552699237261830834781317975544997444273427339909579334573241639236,\\
b =& 55751746669818908907645289078257140818241103727901012315294400837956729358436,\\
G_W =& (19298681539552699237261830834781317975544997444273427339909597334652188435546,\\
& 14781619447589544791020593568409986887264606134616475288964881837755586237401)\\
d_A G_W = P_{AW} =& (75542676157589057894364931773390784087381838505152624792060829784282392000000,\\
& 13070594169239478115446582128550067480452326318343516314474879361527799519490).
\end{aligned}$$

As a check, we find that `petow` used with $P_{AT}$ and `pmtow` used with $P_{AM}$ both result in $P_{AW}$. The user may follow the method above with a randomly chosen integer private key to find public keys on different forms of curve. Finally, Bob chooses his private key, also from [29]:

$$\begin{aligned}
d_B =& 42367284387596200873619098685140639460546026971275625760\\
& 324530697030477603051.
\end{aligned}$$

Using the original Montgomery form of `Curve25519`, $M$, Bob's public key is:

$$\begin{aligned}
d_B G_M = P_{BM} =& (51534159768607372314891409816707645824172465383971870511120431129112472931762548351020155575593031973450374357363288544924357265966716119297912815393754 4).
\end{aligned}$$

To establish a shared secret, Alice and Bob exchange their public keys, $P_{AM}$ and $P_{BM}$ respectively. The secret point $S$ satisfies: $d_A P_{BM} = d_A d_B G_M = d_B d_A P_{AM} = d_B P_{AM}$. Since solving the ECDLP (Definition 1.12) is extremely difficult, the secret point is safe between Alice and Bob. In fact, when `Curve25519` was released by Bernstein, every known attack on it was slower than attempting a brute force search of the shared secret

[3]. With the function `mmul` $S$ is calculated:

$$d_A P_{BM} = d_B P_{AM} = (795619927674238113347381155679510017105877537333510$$
$$635569308399507069532874 5,$$
$$136069249970463861743314550562765181899958184296343$$
$$431588446325471758725304 44).$$

### 3.2.2 Assumptions

The aspects of each form of curve considered here have been written about since Koblitz's introductory paper [13, 19, 20, 39, 46]. Most consider Montgomery and twisted Edwards curves to have certain advantages over Weierstrass curves, whether speed or security [16, 38, 42]. We begin with the speed of point multiplication. Below summarises the number and type of operations needed for point addition and doubling.

Table 3.1: Point Addition:

| Form | Multiplications | Additions | Subtractions | Inversions |
|---|---|---|---|---|
| Weierstrass | 3 | 0 | 6 | 1 |
| Montgomery | 4 | 0 | 7 | 1 |
| Twisted Edwards | 14 | 2 | 2 | 2 |

Table 3.2: Point Doubling:

| Form | Multiplications | Additions | Subtractions | Inversions |
|---|---|---|---|---|
| Weierstrass | 7 | 1 | 3 | 1 |
| Montgomery | 11 | 2 | 4 | 1 |
| Twisted Edwards | 14 | 2 | 2 | 2 |

It looks like twisted Edwards curves would be the slowest to perform point operations, followed by Montgomery, then Weierstrass. In the example above the multiplication to find Alice's public key took:

| Form | Calculation | Time(s) |
|---|---|---|
| Weierstrass | $d_A G_W$ | 0.0625 |
| Montgomery | $d_A G_M$ | 0.046875 |
| Twisted Edwards | $d_A G_T$ | 0.09375 |

Obviously this is a small and isolated example, which may depend on the points involved, their orders, and the curve parameters. For instance, the twisted Edwards

and Montgomery curve parameters have up to six base-10 digits, but the Weierstrass parameters have 77. This may explain why the Montgomery curve was unexpectedly faster than Weierstrass. However, the scalar multiple $d_A$ and the prime base $p$ remain constant[2], so the *number* of doublings and additions is the same for each curve. These kind of comparisons will be put to use more extensively in the following section.

When we say one form of curve is 'more secure' than other, what do we really mean? Blockchain, for instance, is a technology which relies on ECC for its use as an immutable ledger of information [53]. We know that the information is authentic because of *hash functions* which act as a digital fingerprint, making it very clear if data has been tampered with [44]. However, hashes do not stop data being tampered with or a user being able to impersonate someone else. It is the ECDLP (Definition 1.12) and the particular curve's resistance to attacks which make the technology secure.

The security of a cryptosystem is traditionally measured by its resistance. Perhaps the simplest type of attack to resist is a *brute force* attack. Elliptic curve cryptography is easily made resistant to a brute force attack. Repeatedly computing $kG$ for many values of $k$ is considered infeasible as long as the subgroup order $n$ is greater than $2^{80}$, in other words larger than 80 bits, which we can ensure with a large prime base $p$ and large factors[3] of the order of the curve $N$.

To illustrate this, see how easy it is to break the cryptosystem based on the small example given in Section 3.1.1. We have the curve $W$ given by $O_W$ and solutions $(x, y) \in \mathbb{Z}_{13}^2$ to:

$$y^2 = x^3 + 2x + 1.$$

Say Bob randomly chooses his private key $d_B = 5$. The generator of the subgroup is publicly known as $G = (0, 1)$, which has an order of $n = N = 8$. So Bob calculates his public key $P_B = d_B G = (8, 3)$. An attacker would know the value of $G$ and $P_B$, so they could try multiplying integers by $G$:

$$\begin{aligned} 1G &= (0, 1), \\ 2G &= (1, 11), \\ 3G &= (8, 10), \\ 4G &= (2, 0), \\ 5G &= (\mathbf{8}, \mathbf{3}) = P_B. \end{aligned}$$

[2]However, $p \mod 4 = 1$, which means to find square roots we use the Tonelli Shanks algorithm, which is much slower and more variable than if $p \mod 4 = 3$.

[3]Recall that due to Lagrange's Theorem the order of a subgroup of $G$ must divide the order of $G$ [53].

The order $N$ of the elliptic curve is often published or can be found in polynomial time[4]. Therefore the attacker could verify the order of $G$, which divides $N$, and so that any integer $k = 5 \mod n$ results in Bob's public key. If $n > 2^{80}$, it would take an average of $n/2 > 2^{79}$ calculations to find the private key through brute force.

As to more specific attacks, Koblitz, Menezes, and Vanstone outlined a process for selecting resistant curves in 2000 [27, p. 186]:

- To safeguard against the *Pollard $\rho$-attack* the number of points in the curve must be divisible by a sufficiently large prime $n > 2^{160}$.
- To safeguard against the *Semaev-Smart-Satoh-Araki attack*[5] the number of points in the curve should not equal the prime base $p$.
- To safeguard against the *MOV reduction attack*[6], the subgroup size $n$ as in Section 1.1.2 should not divide $p^k - 1$ for all positive $k \leq C$. So discrete logarithms cannot be computed, $C$ should be sufficiently large ($C = 20$ works in practice).

The first point also protects against an attack based on the 1978 *Pohlig-Hellman* algorithm, which reduces the ECDLP to smaller problems by factoring $n$ [27, 49]. These smaller problems can then be attacked with any of the three above. Clearly, if $n$ is prime it cannot be factored.

The last two points are still considered some of the most important attacks to resist. For instance, Ivey-Law and Rolland generated strong curves specifically resistant to MOV and Semaev-Smart-Satoh-Araki in 2010 (by which time choosing $n$ prime was common practice) [20].

In the last 19 years elliptic curves have been increasingly used in smartphones and other mobile devices [22]. This leads to some "careless" implementations being susceptible to *side channel attacks* in which the power consumption or computing time is monitored [21]. Like *frequency analysis* attacks on conventional ciphers, which analyse the number of times letters or symbols appear compared to the plaintext language, side channel attacks can potentially discover private keys from the number of operations used. These attacks are considered rather serious because they need no ciphertext or plaintext. As we can see in Tables 3.1 and 3.2, adding and doubling usually require a different number of operations, so an adversary could use the double and add algorithm to work backwards and discover the private key used as a scalar multiple. Note that a key advantage of twisted Edwards curves is that the operations require the same amount of time, therefore protecting against side channel attacks.

[4]The most commonly used algorithm is due to Schoof, which is out of scope of this project [20, 25].

[5]Introduced separately by Smart, Semaev, and Satoh-Araki in the late 1990s, also known as the Smart-ASS attack [6].

[6]Introduced by Menezes, Okamoto, and Vanstone in 1993 [36].

This project is focused on finding fast and resistant curves, so for our purposes we only need the methods to protect against such attacks. Below is just a brief outline of how these attacks work.

### Pollard $\rho$

Pollard's 1975 algorithm is based on the *birthday paradox* [28]. This well known paradox shows that to be almost sure that two people in a group have the same birthday we only need 70 people [47, p. 404]. To be 50% sure we only need 23 people - a surprising result [30]. Informally, the paradox exists because finding two elements in a set which are both in a subset, known as *collisions*, is far easier than finding two specific elements in a set [47, p. 389].

Say we have the ECDLP as in Definition 1.12, so we are looking for $k$ in $kP = Q$ where $P, Q$ are given and in the group $E$ of order $n$ defined by an elliptic curve.

The Pollard $\rho$ algorithm first finds random integers $a, b \in [0, n-1]$ [28]. Then the point $aP+bQ$ is calculated and the user attempts to find an identical point by selecting random integers and repeating the multiplication. Thanks to the property of collisions, this should only take around $\sqrt{n}$ choices. Once the user finds the identical point given by $cP + dQ$, it satisfies:

$$\begin{aligned} aP + bQ &= cP + dQ \\ (a-c)P &= (d-b)Q \\ kP = \frac{a-c}{d-b}P &= Q. \end{aligned}$$

As long as the modular inverse of $d - b$ exists we can find $k$ easily. In reality we must define subsets of an elliptic curve, which is not always that simple, but for our purposes the above outline suffices [47]. Since the algorithm takes around $\sqrt{n}$ steps, a subgroup order of 160 or more bits will take at least $\sqrt{2^{160}} = 2^{80}$ steps [27].

A more general method, known as *baby-step giant-step*, is also applicable to the ECDLP. A particular algorithm is due to Shanks, but takes the same amount of time as Pollard $\rho$ (hence needs the same protection, $n \geq 2^{160}$) and requires much more storage so we do not consider it here [47, p. 382].

### Pohlig-Hellman

Pohlig and Hellman's algorithm aims to reduce the ECDLP by recursively finding smaller prime subgroups and therefore smaller problems to solve [49]. Let the ECDLP be described as above, but where $n$ has the prime factors:

$$n = q_1^{e_1} q_2^{e_2} ... q_r^{e_r}.$$

Then the target integer $k$ can be expressed by:

$$\begin{aligned} k &= k_1 \mod q_1^{e_1}, \\ &= k_2 \mod q_2^{e_2}, \\ &... \\ &= k_r \mod q_r^{e_r} \end{aligned}$$

where each $q$ is clearly co-prime. This system can be solved uniquely using the *Chinese Remainder Theorem*, a quick method for finding such linear congruences [47, p. 378]. How do we find these congruences? Each $k_i$ can be broken down thus [40]:

$$k_i = z_0 + z_1 q_i + z_2 q_i^2 + ... + z_{e-1} q_i^{e-1},$$

where $e = e_i$, the power of $q_i$ in $k = k_i \mod q_i^{e_i}$. Now we can apply this to our points on an elliptic curve. First we define:

$$\begin{aligned} P_0 &= \frac{n}{q_i} P, \\ Q_0 &= \frac{n}{q_i} Q, \\ \implies Q_0 &= \frac{n}{q_i} kP = kP_0 \\ \text{where} \quad q_i P_0 &= q_i \frac{n}{q_i} P = nP = O. \end{aligned}$$

This shows that our defined $P_0$ has order $q_i$, which is prime so the order cannot be any smaller multiple. We have a much smaller ECDLP to solve in finding $kP_0 = z_0 P_0 = Q_0$ [40]. Say we can apply some method (like MOV or Pollard $\rho$) to solve for $z_0$. Then we do the same for $z_1$ by solving:

$$\begin{aligned} Q_1 &= z_1 P_0, \\ \text{where} \quad Q_1 &= \frac{n}{q_i^2}(Q - z_0 P). \end{aligned}$$

This method will eventually find all $k_i$ and therefore $k$ in a number of steps which rely on the prime factors of $n$, not $n$ itself. Pohlig-Hellman takes around $\sqrt{q_j}$ steps, where $q_j$ is the largest prime factor of $n$ [49]. Combining this algorithm with Pollard $\rho$ is the fastest known way of solving the ECDLP [17, 40].

### MOV

The MOV attack is a type of *multiplicative transfer* in which the ECDLP is 'transferred' to an easier to solve DLP in a linear algebraic group [6]. A similar algorithm of the

same type is the *Frey-Rück*[7] attack which can be resisted against in the same way as resisting MOV [49].

Say that an elliptic curve has a subgroup of order $n$ such that $n|p^d - 1$ for some $d$. This $d$ is the *embedding degree*, which means that the MOV algorithm can be used to reduce the ECDLP to the DLP in the multiplicative group $\mathbb{Z}^*_{p^d}$, which is considered easier for small $d$ [47, p. 387].

Informally we find some point $T$ which (also) generates $\langle P \rangle$. So, we have three known points, $P$, $Q$, and $T$, which are all in the subgroup of order $n$. It can be shown that this subgroup is strictly inside the group formed by the same elliptic curve over $\mathbb{Z}^*_{p^d}$, so all calculations can be done in $\mathbb{Z}^*_{p^d}$. The point $T$ is used like an anchor, in that its relationship to $Q$ is compared to its relationship to $P$ so that $k$ may be extracted [47, p. 387].

The class of curves susceptible to the MOV attack are also known as *super-singular* [27, 36]. Rather confusingly, they are never singular. They have an embedding degree of $d \leq 6$ and so the MOV attack reduces the problem significantly, whereas an embedding degree of $d \geq 20$ actually transfers the ECDLP to an even harder DLP, rendering the attack useless [27]. Before the attack was discovered, super-singular curves were thought to be "very promising" in ECC [41].

### Semaev-Smart-Satoh-Araki

Recall from earlier in this project the comparison between $\mathbb{Z}^*_p$ and $\mathbb{Z}^+_{p-1}$. These two groups are isomorphic, but the DLP is difficult to solve in the former and easy in the latter. The Semaev-Smart-Satoh-Araki attack is an *additive transfer* which 'transfers' the ECDLP to a DLP in an isomorphic additive group [27]. The DLP in an additive group is, as stated above, essentially trivial and solvable in polynomial time.

If the number of points in an elliptic curve, $N$, equals the prime base $p$ the curve is said to be *anomalous* [27]. Therefore the elliptic curve is isomorphic to the additive group $\mathbb{Z}^+_p$. The attack, when published in the late 1990s, showed that this isomorphism can be computed in linear time [40]. The mathematics involved is out of scope of this topic, but briefly; we have a polynomial $f(x) = 0 \mod p$ and 'lift' some element $x$ to $x'$ such that $x' = x \mod p$ and $f(x') = 0 \mod p^2$. This method, *Hensel's lemma*, lifts points in the curve over $\mathbb{Z}_p$ to the curve over $\mathbb{Q}_p$, the field of *p-adic numbers* [47, p. 389]. An isomorphism from $\mathbb{Q}_p$ to the additive group $\mathbb{Z}^+_p$ can then be efficiently calculated [27].

[7] The MOV attack uses the *Weil pairing* whereas the Frey-Rück attacks uses the *Tate pairing* to 'transfer' the problem [27].

### Mappings

What if we discover some fantastically secure twisted Edwards curve and want to reduce the running time of encryption based on it? Would converting that curve to another form with faster operations, a), maintain the good security properties, and, b), be worth it with the cost of transformation?

We aim to discover this for at least a few real world examples. The 'cost' of transforming our curves in terms of parameters is given below.

Table 3.3: Parameter transformation:

| From | To | Mult. | Add. | Subt. | Inver. | Sqrt. | Cbrt. |
|---|---|---|---|---|---|---|---|
| Weierstrass | Montgomery | 7 | 4 | 2 | 4 | 2 | 2 |
| Montgomery | Weierstrass | 10 | 0 | 2 | 2 | 0 | 0 |
| Twisted Edwards | Montgomery | 1 | 1 | 2 | 2 | 0 | 0 |
| Montgomery | Twisted Edwards | 0 | 1 | 1 | 2 | 0 | 0 |
| Weierstrass | Twisted Edwards | 7 | 5 | 3 | 6 | 2 | 2 |
| Twisted Edwards | Weierstrass | 11 | 1 | 4 | 4 | 0 | 0 |

Below are the operations required for point mappings, assuming the parameters have already been mapped and the point is not an exceptional one.

Table 3.4: Point transformation:

| From | To | Mult. | Add. | Subt. | Inver. | Sqrt. | Cbrt. |
|---|---|---|---|---|---|---|---|
| Weierstrass | Montgomery | 2 | 0 | 1 | 0 | 0 | 0 |
| Montgomery | Weierstrass | 1 | 1 | 0 | 3 | 0 | 0 |
| Twisted Edwards | Montgomery | 1 | 2 | 2 | 2 | 0 | 0 |
| Montgomery | Twisted Edwards | 0 | 1 | 1 | 2 | 0 | 0 |
| Weierstrass | Twisted Edwards | 2 | 1 | 2 | 2 | 0 | 0 |
| Twisted Edwards | Weierstrass | 2 | 3 | 2 | 5 | 0 | 0 |

It appears that, unsurprisingly, the requirement of a point of order two on a Weierstrass curve means that transformations from it are the most expensive. They are the only ones to need square and cube roots. Otherwise, twisted Edwards and Montgomery have a relatively fast set of mappings between them.

Below we choose three curves to test. Each of the curves are converted to the other two forms, giving nine separate groups which are tested for speed and security. Security is tested using Koblitz et. al's recommended three tests, since they encompass all the attacks apart from side-channel, which will be discussed separately.

## 3.2.3 Results

The transformations and multiplications in this section have been completed with the Python functions described previously. The reader may duplicate them using the code

in Appendix A.2. The resistance to the MOV attack has been checked with the following function:

```
def MOVcheck(n,p):
    for k in range(1,20):
        q = (p**k - 1) % n
        if q == 0:
            print(k)
            return "oh no!"
    return "Safe for all k < 20"
```

The functions are timed individually by adding the line:

```
start = time.process_time()*1000
```

at the beginning of each and the line:

```
print(time.process_time()*1000 - start)
```

at the end. We multiply by 1000 to have outputs in milliseconds.

`Curve25519`

Bernstein's curve, used in the examples above, has Montgomery curve parameters:

$$M = \{y^2 = x^3 + 486662x^2 + x\} \cup \{O_M\},$$

twisted Edwards curve parameters:

$$T = \{486664x^2 + y^2 = 1 + 486660x^2y^2\},$$

and Weierstrass parameters:

$$\begin{aligned} W &= \{y^2 = x^3 + ax + b\} \cup \{O_W\}, \\ a &= 19298681539552699237261830834781317975544997444273427339909597334573241639236, \\ b &= 55751746669818908907645289078257140818241103727901012315294400837956729358436. \end{aligned}$$

All curves are defined with the prime base $p = 2^{255} - 19$ where $(x, y) \in \mathbb{Z}_p$. Recall that all the forms are isomorphic to each other. We have calculated the given base point,

$G_M$ in the other forms:

$$
\begin{aligned}
G_M &= (9, y_{GM}),\\
y_{GM} &= 14781619447589544791020593568409986887264606134616475288964881837755586237401.\\
G_T &= (x_{GT}, y_{GT}),\\
x_{GT} &= 38213832894368730265794714087330135568483813637251082400757400312561599933396,\\
y_{GT} &= 46316835694926478169428394003475163141307993866256225615783033616525185596003165251855960.\\
G_W &= (x_{GW}, y_{GW}),\\
x_{GW} &= 19298681539552699237261830834781317975544997444273427339909597334652188435546,\\
y_{GW} &= 14781619447589544791020593568409986887264606134616475288964881837755586237401.
\end{aligned}
$$

We first test how long the transformations took. The values are taken as an average of 10 trials and can be found in Table 3.5. Note that the mappings were so fast they often read as taking 0.0 milliseconds (less than one ten-thousandth of a second). However, when the time read as above 0.0 it was either 15.625ms or more rarely 31.25ms. The point mappings took a negligible amount of time, all reading 0.0ms.

Table 3.5: Parameter transformation:

| From | To | Time (ms) |
|---|---|---|
| $M$ | $T$ | 15.625 |
| $M$ | $W$ | 15.625 |

We randomly choose two new private keys for Alice for Bob. Bernstein gave the order of the subgroup generated by $G_M$ as

$$n = 2^{252} + 27742317777372353535851937790883648493,$$

with cofactor 8. Using the `random.randint` Python function to choose the keys between 1 and $n - 1$, they are:

$$
\begin{aligned}
d_A =& 70346741210494194156093322210816861606269611125252819066253088690699863499972,\\
d_B =& 45148821345967468885410277587750960607366613335787211925834391701215684448817.
\end{aligned}
$$

We time the calculation of each person's public key:

| Form | Calculation | Time (ms) |
|---|---|---|
| Weierstrass | $d_A G_W$ | 48.4375 |
| Weierstrass | $d_B G_W$ | 51.5625 |
| Montgomery | $d_A G_M$ | 50 |
| Montgomery | $d_B G_M$ | 48.4375 |
| Twisted Edwards | $d_A G_T$ | 96.875 |
| Twisted Edwards | $d_B G_T$ | 95.3125 |

Table 3.6: `Curve25519` Multiplication

The forms have the following security properties:

| Form | $n > 2^{160}$ | $N \neq p$ | $n \nmid p^k - 1$ |
|---|---|---|---|
| Weierstrass | Y | Y | Y |
| Montgomery | Y | Y | Y |
| Twisted Edwards | Y | Y | Y |

Table 3.7: `Curve25519` Properties

Since the curves are isomorphisms, all the generators have the same order, $n$.

`GSS192`

Consider a super-singular curve, `GSS192`. This curve was found by the author using a definition by Galbraith and some online tools[8] [18]. Galbraith states that a Weierstrass curve with $a = 0$ and $p$ a prime equivalent to $-1 \mod 3$ will be super-singular with $N = p + 1$.

So, a 192 bit prime base was chosen first with $b = 7$. After a few candidates, a curve was found that is transformable. `GSS192` is defined by $(x, y) \in \mathbb{Z}_p^2$ satisfying:

$$W = \left\{y^2 = x^3 + 7\right\} \cup \left\{O_W\right\},$$
$$p = 6277101735386680763835789423207666416102355444464034513407.$$

In Montgomery form it has parameters:

$$M = \left\{By^2 = x^3 + Ax^2 + x\right\} \cup \left\{O_M\right\},$$
$$A = 3146409713659991312944025474448559867907002794198816287850,$$
$$B = 1461027439901713700097924012583120382231530248201854856464.$$

[8] A prime $p$ and the factors of $N$ were found using the tools at `https://www.numberempire.com/`.

In twisted Edwards form:

$$
\begin{aligned}
T &= \left\{ax^2 + y^2 = 1 + dx^2y^2\right\},\\
a &= 611240939468179922428865613171498571789037864826345267070 0,\\
d &= 142359854623578508706840955228795322562712638744724199501.
\end{aligned}
$$

The parameter $a$ is square and $d$ is non-square, so the above `GSS192` mappings are isomorphisms. We know, from Galbraith, that the order of the curve is $N = p + 1$. We factor this using online tools to find a large divisor of $N$ is:

Table 3.8: Parameter transformation:

| From | To | Time (ms) |
|---|---|---|
| $W$ | $M$ | 15.625 |
| $W$ | $T$ | 31.25 |

$$n = 6974557483762978626484210470230740462335950493848927237 12$$

with cofactor 9. Clearly $n$ is not prime, so we will have to check that it is indeed the order of the points we look at. Otherwise, a factor of $n$ may be the true order.

We then randomly choose a point such that its order is $n$ to find a suitable $G$. One method of doing is this choosing a random $x$ candidate between 1 and $n$, finding $x^3 + ax + b \mod p$ then testing if the result is a square. If it is, then take the square root as $y$. We then multiply the point $(x, y)$ by $n$, if the result is $O$ and none of the factors of $n$ multiplied by the point make $O$, we have found a suitable base point, $G$. If not, try a new $x$ and repeat. Using this method we find:

$$
\begin{aligned}
G_W &= (x_{GW}, y_{GW})\\
x_{GW} &= 274575876994753672208097084134289515595252772046688238 04,\\
y_{GW} &= 761649546075906055737049241521228226407507704329895350097.
\end{aligned}
$$

The reader may check that $nG_W$ is indeed $O$ using `wmul`. Now we map this point to the other curve forms. On $M$ and $T$ we have:

$$
\begin{aligned}
G_M &= (x_{GM}, y_{GM})\\
x_{GM} &= 133263502367044626580588557726923109520601548851992249038 0,\\
y_{GM} &= 273568486213561484055863446163567824728151195420574135188 1,\\
G_T &= (x_{GT}, y_{GT})\\
x_{GT} &= 303177552885097364558367603421682169587414667606695534060 4,\\
y_{GT} &= 162048051762533000082257148974515455918780600842641348697 8.
\end{aligned}
$$

Again the function `random.randint` is used to choose the keys:

$$d_A = 655068376910452461247813616685105994105137083400609410 61,$$
$$d_B = 272612786536472890421513157732016979212152322677716400469.$$

The timings for `GSS192` are given by:

| Form | Calculation | Time (ms) |
|---|---|---|
| Weierstrass | $d_A G_W$ | 28.125 |
| Weierstrass | $d_B G_W$ | 25 |
| Montgomery | $d_A G_M$ | 35.9375 |
| Montgomery | $d_B G_M$ | 31.25 |
| Twisted Edwards | $d_A G_T$ | 56.25 |
| Twisted Edwards | $d_B G_T$ | 51.5625 |

Table 3.9: `GSS192` Multiplication

The curve has the following security properties:

| Form | $n > 2^{160}$ | $N \neq p$ | $n \nmid p^k - 1$ |
|---|---|---|---|
| Weierstrass | Y | Y | N |
| Montgomery | Y | Y | N |
| Twisted Edwards | Y | Y | N |

Table 3.10: `GSS192` Properties

### The Million Dollar Curve

The *Million Dollar Curve*[9] was devised in 2016 with as random as possible means, including combinations of many national lottery draws [2]. The motivation was to provide a secure curve which has no secret vulnerabilities built in by the author[10]. The prime base $p$ is 256 bits where $(x, y) \in \mathbb{Z}_p^2$. It was given in Edwards form:

$$E = \left\{x^2 + y^2 = 1 + dx^2y^2\right\},$$
$$d = 393848177413506285731611843012259158003587705893375607194826462 5$$
$$804612259721,$$

[9] All its parameters can be found at `https://cryptoexperts.github.io/million-dollar-curve/`.

[10] These are known as 'backdoors', previously implemented by the NSA, information that was famously leaked by Edward Snowden [7, 53].

where the prime base is:

$$p = 109112363276961190442711090369149551676330307646118204517771511330536253156371.$$

We transform it to Montgomery form:

$$M = \{(C+1)y^2 = x^3 + (C-1)x^2 + x\} \cup \{O_M\},$$
$$C = 319023361857462807194596673782970286074262228290438742479902237839 42726201143.$$

Again, $a$ is a square and $d$ is not in the original Million Dollar Curve, so the mapping between $E$ and $M$ is isomorphic. Note that because $a = 1$ on $E$:

$$B - A = \frac{4 - 2(d+1)}{1-d} = \frac{2-2d}{1-d} = 2.$$

In Weierstrass form the curve is:

$$W = \{y^2 = x^3 + ax + b\} \cup \{O_W\},$$
$$a = 806185785313885370115864908287245283293178973382306829344692560879005534266 1$$
$$b = 123291964297699991703503861278071525457552764575891989426434919675992316989 50.$$

The mappings above again took a very short amount of time. In fact, in the composition of `etom` and `mtow` recorded the same time as `etom`. The time taken to compute the `mtow` part was negligible. Like Bernstein's curve, a base point is given. In Edwards form $G_E$ has the coordinates:

Table 3.11: Parameter transformation:

| From | To | Time (ms) |
|---|---|---|
| $E$ | $M$ | 15.625 |
| $E$ | $W$ | 15.625 |

$$x_{GE} = 825498032222023993400244620329649425120258568187004142547263642050967314243 15,$$
$$y_{GE} = 915495456374157344226582887991190417563782595230971478078133969151259328114 45.$$

We transform it to Montgomery and Weierstrass forms:

$$
\begin{aligned}
G_M &= (x_{GM}, y_{GM}),\\
x_{GM} &= 642066528749174165996339877006026309253537185608244629804157144061099266229 42,\\
y_{GM} &= 302670997117810160139393197188978501643605521874361266409778442441018705203844.\\
G_W &= (x_{GW}, y_{GW}),\\
x_{GW} &= 418954541530157840805661121419701478703925588602749523657271972523200363510 76,\\
y_{GW} &= 439765906067181269911181877490801233217972819852379212315719261870442203263 90.
\end{aligned}
$$

The order of $G$ on each curve is:

$$
\begin{aligned}
n =& 272780908192402976106777725922873879189305095740480688876309782931855219732 43,
\end{aligned}
$$

where $n$ is prime with cofactor 4. We randomly choose the final private keys:

$$
\begin{aligned}
d_A =& 376300209676405022659952259674368969122057775997649562340461280563317787160 7,\\
d_B =& 180016037860310645929131612756262412460675560212021467450584313238795480357 41.
\end{aligned}
$$

The timings and security properties of the curve are:

| Form | Calculation | Time (ms) |
|---|---|---|
| Weierstrass | $d_A G_W$ | 45.3125 |
| Weierstrass | $d_B G_W$ | 51.5625 |
| Montgomery | $d_A G_M$ | 53.125 |
| Montgomery | $d_B G_M$ | 56.25 |
| Edwards | $d_A G_E$ | 95.3125 |
| Edwards | $d_B G_E$ | 101.5625 |

Table 3.12: MDC Multiplication

| Form | $n > 2^{160}$ | $N \neq p$ | $n \nmid p^k - 1$ |
|---|---|---|---|
| Weierstrass | Y | Y | Y |
| Montgomery | Y | Y | Y |
| Edwards | Y | Y | Y |

Table 3.13: MDC Properties

# Chapter 4

# Analysis and Conclusions

## 4.1 Explanation of Results

This section brings together the results above for analysis. To summarise, the results are:

| Curve | | Million Dollar Curve | `GSS192` | `Curve25519` |
|---|---|---|---|---|
| Bit size of $n$ | | 252 | 188 | 252 |
| Form | Calculation | Time (ms) | Time (ms) | Time (ms) |
| Weierstrass | $d_A G_W$ | 45.3125 | 28.125 | 48.4375 |
| Weierstrass | $d_B G_W$ | 51.5625 | 25 | 51.5625 |
| Montgomery | $d_A G_M$ | 53.125 | 35.9375 | 50 |
| Montgomery | $d_B G_M$ | 56.25 | 31.25 | 48.4375 |
| Twisted Edwards | $d_A G_T$ | 95.3125 | 56.25 | 96.875 |
| Twisted Edwards | $d_B G_T$ | 101.5625 | 51.5625 | 95.3125 |

Table 4.1: Multiplication times

| Curve | Million Dollar Curve | | | `GSS192` | | | `Curve25519` | | |
|---|---|---|---|---|---|---|---|---|---|
| Form | $n > 2^{160}$ | $N \neq p$ | $n \nmid m$ | $n > 2^{160}$ | $N \neq p$ | $n \nmid m$ | $n > 2^{160}$ | $N \neq p$ | $n \nmid m$ |
| W | Y | Y | Y | Y | Y | N | Y | Y | Y |
| M | Y | Y | Y | Y | Y | N | Y | Y | Y |
| T | Y | Y | Y | Y | Y | N | Y | Y | Y |

Table 4.2: Security Properties, $m = p^k - 1$

As to the time taken for curve mappings, it appears that the smallest amount of time Python can measure is 15.625ms. Anything faster than that, for instance in the case of `Curve25519`, is recorded as 0.0ms. We also have `GSS192` taking double that increment, 31.25ms, to perform the Weierstrass to twisted Edwards mapping. Though not entirely accurate, the fact that it was the only mapping to take two minimum increments while having the smallest bit size reflects the high number of operations required. In all, the curve mappings are rational and so take an almost negligible amount of time to

compute even considering the 256 bit parameters of the Million Dollar Curve (MDC). Similarly, for all the point mappings of $G$ the timings read as 0.0ms.

Point multiplication speeds generally follow the bit size and form of curve. For `Curve25519`, $n$ has 252 bits and the Weierstrass multiplication was the slowest of the three overall, though there was not much difference between it and the Montgomery form. As observed before, this is probably because the parameters of the Weierstrass form are so much larger than the other forms. However, both Montgomery and twisted Edwards multiplication were noticeably faster than the MDC on average. Unlike all other cases, the Montgomery form had the fastest point multiplication of all forms. This is noteworthy and in line with the assumptions because `Curve25519` was originally given in those coordinates, which only had 6 base-10 digits. The twisted Edwards form, too, had small parameters compared to the curve's bit size.

In the case of `GSS192`, its very small Weierstrass parameters ($a = 0$, $b = 7$) likely explain why it has the largest gap between multiplication on the forms. Indeed, any multiplication involving $a$ is eliminated. The curve provides adequate security in terms of the *size* of $n$ but only takes around half the time of `Curve25519` to perform twisted Edwards multiplication. Additionally, the proportional difference between multiplication on the Weierstrass form and the twisted Edwards form is smaller than on the other curves.

As expected, the twisted Edwards form of each curve has considerably slower point multiplication. For `Curve25519` and the MDC it takes nearly double the amount of time. From the assumptions above this is not all that surprising. Both point addition and doubling require 14 multiplications whereas Montgomery only needs 4 and Weierstrass 3. It may also be significant that the twisted Edwards form uses two modular inversions rather than one. Inverting an element in a finite field is a costly operation which cryptographers generally try to avoid[1] [3].

Despite the long algorithms for modular cube and square roots, the curve mapping calculations on `GSS192` involving them were quick. We can see that this is because the prime base $p$ is equivalent to $-1 \mod 3$ and $-1 \mod 4$, so roots can be found in just one line. Since $p = -1 \mod 3$ was a requirement of Galbraith's to produce a super-singular curve, it likely contributes to the reputation of the class being efficient in cryptography.

For completeness, let's look at the other forms' performance on roots. The MDC also has a base $p$ allowing for one line calculations of cube and square roots. However, `Curve25519` has $p$ congruent to $1 \mod 4$ and $1 \mod 3$. Any roots computed in the field underlying the curve would be completed with the longer probabilistic algorithms.

[1] This is partly the motivation behind using *projective coordinates*, which are discussed in Section 4.3.

However, they are rarely needed in cryptography. As an illustration of this property the below table shows the average time taken (of 10 trials) to compute 100 roots[2] in each curve's underlying field:

| Curve | Sqrt (ms) | Cbrt (ms) |
|---|---|---|
| `Curve25519` | 260.9375 | 48.4375 |
| MDC | 173.4375 | 23.4375 |

Though it is important to point out that Bernstein chose the value $p = 2^255 - 19$ because, being so close to a power of two, modular calculations are particularly fast[3] [3].

We can conclude that, in these three examples, a transformation to another form of curve never outweighs any benefit the new form may bring. This may not be true for curves which are not transformable in their original finite field. For instance, the curve `secp256k1`, used by Bitcoin, has many useful real world properties but has no point of order two [7]. Therefore, it is not transformable to another form of curve without extending the underlying field, which we did not consider in this paper. Doing so could change the number of elements in the group represented by the curve and hence its security.

If we 'ignore' side-channel attacks for the moment, it appears that transforming the MDC from Edwards to another form would gain the user much faster point arithmetic without compromising on security. Though if we do consider side-channel attacks, there is a small difference in time taken between multiplying by $d_A$ and $d_B$. This is not to say such attacks are not a problem; more sophisticated timing software could likely discover how many doublings and additions have occurred [21]. The private key $d_A$ used for the MDC has 117 ones in binary, while $d_B$ has 135. This translates to 117 vs 135 doublings in point multiplication and over 6ms of time difference. It is unlikely that Python is able to pick up more exact difference between them. In the case that an adversary is able to detect an accurate difference in computational power used between doubling and addition, one might think that 'masking' which operation is occurring would slow down the process. However, this is not necessarily true, as is discussed in Section 4.3.

Since the order of $G$ `GSS192` is not prime, the curve is susceptible to the Pollig-Hellman attack, reducing the ECDLP to that of its prime factors. They are $2^9$, 768614336404564651, and 177230399437988782976979507730256145 1. The largest of these factors, and therefore of $N$, is only 120 bits. Whatever choice of $n$ the user

[2]Note that the cube roots are considerably faster. Square roots always need the Legendre Symbol computed and the Tonelli-Shanks algorithm needs many more steps in finding $m$ than the cube root algorithm needs altogether.

[3]This is more to do with how CPUs perform multiplication using binary than mathematical complexity.

makes, it is either vulnerable to the Pollig-Hellman attack or the Pollard $\rho$ attack.

In all cases we have isomorphisms, therefore the same number of points and size of subgroups. It is then straightforward to see why all the forms considered share the same security properties. Is this true for birational equivalences?

If there exists a mapping between a Weierstrass and Montgomery curve, we know it is isomorphic, so we need not consider those. Between a twisted Edwards and Montgomery form curve there is an isomorphism if $a$ is a square and $d$ is not. Then the addition operation is complete, in other words it works everywhere. Otherwise, there are exceptional points as defined in Section 2.2.2. The first occurs when $ad$ is a square and the second if $d$ is a square. These points have order two and four respectively, so if $n$ is chosen to be a large prime they should not lie in the subgroup with order $n$. The number of points, $N$, remains the same since the exceptional points derive from rational points on the Montgomery curve which are easily 'counted'[4]. Therefore, if the crucial subgroup is chosen correctly the security properties will remain the same.

For example, take a 162 bit Mongtomery curve defined by Okeya et. al in [42]. It is defined in Montgomery form as:

$$\begin{aligned} M &= \left\{By^2 = x^3 + Ax^2 + x\right\} \cup \left\{O_M\right\}, \\ A &= 3335899736583916783320294232912375101009135441327, \\ B &= 8083900039894232551375264866333856154441580703 72, \end{aligned}$$

with the prime base:

$$p = 576689958026100003984456641521269520129468316011 9.$$

In twisted Edwards form it is:

$$\begin{aligned} T &= \left\{ax^2 + y^2 = 1 + dx^2y^2\right\}, \\ a &= 220926061725198727346623460465880813644918700121, \\ d &= 111884986507327681808298095429593310718244249086 5, \end{aligned}$$

where $a$ is non-square and $d$ is a square. Therefore the mapping is a birational equivalence. Using the same method as for `GSS192` we choose a subgroup order and generator $G$. Okeya et. al give the order of the curve as:

$$N = 576689958026100003984456829877482140832400462177 2,$$

[4]However, if we are mapping from a twisted Edwards to a Montgomery curve where $d$ is a square, the points at infinity which map to rational points cannot be 'seen'. Therefore, if the transformation is point by point, finding the size of the elliptic curve is prone to errors.

which is divisible by 4. The result, $n = N/4$, is prime, so we have a group order. The generator is found randomly:

$$\begin{aligned} G_M &= (x_{GM}, y_{GM}) \\ x_{GM} &= 563318564087466329990658253543778055385440511754 2, \\ y_{GM} &= 561280963998290175737791077350613468410470059352 9. \end{aligned}$$

The generator is transformed to:

$$\begin{aligned} G_T &= (x_{GT}, y_{GT}) \\ x_{GT} &= 345197194703165058389021256509393763935047978510 3, \\ y_{GT} &= 209363881479416205210513501842540058100767261843 9, \end{aligned}$$

where the order of $G_T$ is also $n$. We can check this with the function `emul`. Since $n$ is prime and $G_T$ is not the identity, we know that its order must be $n$. This case demonstrates that security properties[5] reliant on $n$ are unchanged through the birational equivalences in Section 2.2.2.

As curves used in real world cryptography, we expected `Curve25519` and the MDC to pass all the security tests. Additionally as a super-singular curve, `GSS192` predictably failed the MOV attack check. With the function `MOVcheck` it is shown that $n|p^2 - 1$, so the curve has a small embedding degree of 2. The ECDLP in the curve can hence be reduced to the DLP in $\mathbb{Z}^*_{p^2}$. Though it was not predicated that an arbitrary choice of $p$ would lead to poor values of $n$. Otherwise `GSS192` had impressive speed performance and it can be seen why super-singular curves were so promising [41].

Resistance against attacks can be achieved relatively easily with tools such as Schoof's algorithm for counting points [20]. The size of an elliptic curve can be checked in polynomial time and hence whether it has a suitably large subgroup. This doesn't require prime factorisation since the cofactor $h$ should be small, so small values of $h$ are tested to see if $N/h = n$ is a large prime. A large $n$ protects against brute force, baby-step giant-step, and Pollard $\rho$ attacks while a prime $n$ protects against the Pohlig-Hellman attack. Simple verification checks can protect against the MOV attack, by verifying that $n \nmid p^k - 1$, and the Semaev-Smart-Satoh-Araki attack, by verifying $N \neq p$.

[5] Though in this case the curve is 162 bits, $n$ is 159 bits, just failing the size requirement.

## 4.2 Conclusions

The still widely used Weierstrass form demonstrated fast multiplication and easily checked security properties. However, the form is the only one considered here which could have a prime number of points equalling $p$. If this occurs, it leaves the curve susceptible to the Semaev-Smart-Satoh-Araki attack, which is perhaps the most serious attack because of the reduction to an additive group. The ECDLP can then be calculated in seconds [31]. Though it should be noted that randomly generated curves rarely have this property and it is easily checked.

Meanwhile, Montgomery form curves showed very close timings to Weierstrass while having the useful property of a curve order divisible by four. Schemes have been devised to produce secure Montgomery curves which have order $4n$ or $8n$ where $n$ is a large prime [33]. Indeed, Bernstein specifically chose to construct `Curve25519` in Montgomery form to have a small cofactor and parameter $A$ without affecting its security. This was before the discovery of the Edwards form. Since then, `Curve25519` has been used in both Montgomery and Edwards form [29].

Though many claim that the Edwards form permits fast point arithmetic, with the algorithms used in this project we have found the opposite [20]. Variations on those algorithms may reduce the number of steps, but could hinder the ease of implementation the form provides. Its key advantage is the complete point addition system encompassing all cases point doubling and addition, partly why considered the most promising form today [48]. The operations additionally do not require a point at infinity, making the job of the developer far easier. It is probably for this reason that work continues mostly on Edwards curves. For instance the MDC, introduced in 2015, is in Edwards form [2].

Another contrast between each form of curve is the method used to choose parameters. The MDC was chosen as randomly as possible, whereas `Curve25519` was built specifically for speed and security. The curve `GSS192` was built arbitrarily by the author with some constraints to ensure a super-singular curve. This explains the faster performance of `Curve25519`. It is even evident in its twisted Edwards form, since the parameters did not increase in bit size when transformed. However Edwards curves were not even discovered when `Curve25519` was built. From this project's results and analysis, it is unlikely that the use of randomness in the choice of parameters affected efficiency.

The arbitrariness of `GSS192` left it vulnerable to many security issues. Ignoring for the moment that it was chosen to be super-singular, the fact is that predicting whether certain curve parameters will lead to vulnerabilities is difficult. After finding a transformable group, we discovered that any choice of $n$ leads to a weak curve. Though the probability of choosing an insecure curve with large $p$ is low (as mentioned above),

finding, checking, and re-finding is time consuming.

This was a problem during the time NSA curves were discovered to be built with a 'backdoor', allowing the government agency to break the encryption [7]. Trust was failing in these 'official' curves but many did not have the confidence or knowledge to build one themselves. This lack of trust led to mathematicians working on randomly generated curves with proof that no backdoors could have been installed. The MDC is one of these.

Both twisted Edwards and Montgomery forms are naturally resistant to the Semaev-Smart-Satoh-Araki attack. This leads to a useful observation between transformability and security. The term 'transformable' means that the curve can be mapped as described in Section 2.2.

**Theorem 4.1.** *An elliptic curve defined over a finite field $F$, with characteristic greater than 3, which is transformable in $F$ is resistant to the Semaev-Smart-Satoh-Araki attack.*

*Proof.* All elliptic curves over $F$ with characteristic $\neq 2, 3$ can be represented in Weierstrass normal form [22, 39]. A curve in this form is transformable to a Montgomery form curve if and only if it has a point of order two. In turn, a Montgomery form curve is always transformable to a twisted Edwards curve. Both forms always a point of order four.

If an elliptic curve has a point of order two or four, there exists a subgroup of that order and hence the order of the curve is divisible by two. Therefore the order of the curve is not a prime and so cannot be equal to the prime base $p$. □

This is by no means a new result, but brings together much of the work in this project.

An interesting and unexpected conclusion from this project is the difficulty of calculating roots modulo $p$. Though many papers explain the mapping between Weierstrass and Montgomery curves, which need the solution to a cubic in a finite field, none state how to compute it [13, 33, 42]. Further research into solving quadratics and cubics finds little to no work in the field of cryptography, despite that, for these mappings at least, they are required. For instance, Liu et. al state that their method of choosing a secure Montgomery form first then transforming after if required is preferable since we avoid solving quadratics in $\mathbb{Z}_p$ [33]. Their conclusion fits with the findings in this paper, but there is no mention of how or why solving quadratics should be avoided at all. This is surprising, because it is an interesting problem closely related to the DLP but known to be solvable when $p$ is prime. The Tonelli-Shanks algorithm is by no means new but is often the only process mentioned in finding square roots [24, p. 128]. This project's implementation of the algorithm is original. In Rene Struik's recent online resource,

there is an explanation of some smaller cases of square roots (such as when $p = -1 \mod 4$) but nothing for the general case [51].

Cube roots modulo $p$ are even scarcer in terms of material. There appears to be no general algorithm within the scope of this paper for calculating them. The author's supervisor developed an algorithm (Section 2.3.2) specifically for this project. Only in a very niche corner of the cryptographic world[6] could the author find mention of solving cubics in a finite field.

## 4.3 Recommendations

This final section outlines further aspects which follow from this report. Cryptography is a unique study; the concept has been used for centuries, but it was only in the 20th that asymmetric systems arose. An elite few developed secure cryptosystems from areas of mathematics thought to have no real world applications. Suddenly, encryption, and number and group theory with it, is necessary in the modern connected world. Everyone uses it, many understand it, but only a few have the patience and skill to grow and improve elliptic curve cryptography. Some we have encountered already; Edwards, Montgomery, Koblitz, Miller, Bernstein, Lange, Smart, and Merkle. Below are recommendations, many from these authors, for extension and clarification of ECC covered here.

Until now, we have been representing elliptic curves in *affine coordinates* with points $(x, y)$. An alternative representation is in *projective coordinates*, where we replace $x$ with $X/Z$ and $y$ with $Y/Z$ [9]. Coordinates are presented in a triple, $(X, Y, Z)$. This substitution modifies the Weierstrass equation to:

$$Y^2Z = X^3 + aX^2Z + bZ^3.$$

If the point $(x_1, y_1)$ satisfies the affine equation, then $(x_1, y_1, 1)$ satisfies the projective equation above. Any multiple of this point, $(\lambda x_1, \lambda y_1, \lambda)$, will also satisfy it since:

$$\begin{aligned}(\lambda y_1)^2\lambda &= (\lambda x_1)^3 + a(\lambda x_1)^2\lambda + b\lambda^3,\\ \lambda^3(y_1^2) &= \lambda^3(x_1^3 + ax_1^2 + b).\end{aligned}$$

We collect all these multiples into the equivalence class $(x_1 : y_1 : 1)$. In general, all points $(\lambda X_i, \lambda Y_i, \lambda Z_i)$ are in the equivalence class $(X_i : Y_i : Z_i)$. Projective coordinates are considered a more natural way to present elliptic curves [52, p. 42]. Indeed, projective space was originally came from adding 'points at infinity' to affine space [47, p. 6]. Let's

[6] In an archived email review of [51], it is mentioned that "the root of $x^3 + ax + b$ in $F_p$ could be provided explicitly" [15].

formalise this concept to ascertain why.

Two-dimensional *projective space* over a field $F$ is the collection of equivalence classes $(X : Y : Z)$, known as points, where $X, Y, Z \in F$ and at least one of them are non-zero [52, p. 18]. It is usually denoted by $\mathbb{P}^2(F)$.

The *affine plane* is the collection of $(X : Y : 1)$ equivalence classes, so it is clear to see that the affine plane is included in projective space. Therefore, all points in an elliptic curve over $F$ are in the projective space over $F$. This includes the point at infinity, which is easiest explained with the projective addition formulae on a Weierstrass form curve [52, p. 42]:

- $P + P$: We convert the affine point doubling formula where $P = (x_1, y_1) = (X/Z, Y/Z)$ and $2P = (x_S, y_R)$. First, $\lambda_d$ and the $x$-coordinate are found:

$$\begin{aligned}\lambda_d &= \frac{3X^2 + aZ^2}{2YZ} = \frac{T}{2YZ},\\ x_S = \lambda_d^2 - 2x_1 &= \frac{T^2}{(2YZ)^2} - \frac{2X}{Z} = \frac{T^2}{(2YZ)^2} - \frac{2X(4Y^2Z)}{Z(4Y^2Z)}\\ &= \frac{T^2 - (2YZ)4XY}{(2YZ)^2} = \frac{W}{(2YZ)^2}.\end{aligned}$$

  Now we may find the $y$-coordinate:

$$\begin{aligned}y_R = \lambda_d(x_1 - x_S) - y_1 &= \frac{T}{2YZ}\left(\frac{X}{Z} - \frac{W}{(2YZ)^2}\right) - \frac{Y}{Z}\\ &= \frac{T}{2YZ}\left(\frac{X(4Y^2Z)}{Z(4Y^2Z)} - \frac{W}{(2YZ)^2}\right) - \frac{Y(8Y^3Z^2)}{Z(8Y^3Z^2)},\\ &= \frac{T}{2YZ}\left(\frac{2XY(2YZ) - W}{(2YZ)^2}\right) - \frac{2Y^2(2YZ)^2}{(2YZ)^3}\\ &= \frac{T(2XY(2YZ) - W) - 2Y^2(2YZ)^2}{(2YZ)^3}.\end{aligned}$$

  To avoid having to compute any inverses we make $x_S$ and $y_R$ have the same denominator. Since $x$ is represented by $X/Z$ and $y$ by $Y/Z$, this common denominator is $Z$:

$$x_S = \frac{T(2YZ)}{(2YZ)^3}, \quad y_R = \frac{T(2XY(2YZ) - W) - 2Y^2(2YZ)^2}{(2YZ)^3},$$

$$(x_S, y_R) \mapsto (x_S : y_R : 1) \sim (T(2YZ) : T(2XY(2YZ) - W) - 2Y^2(2YZ)^2 : (2YZ)^3)$$

  Where $\sim$ denotes equivalence.

- $P + Q$, $P \neq \pm Q$: We have $P = (x_1, y_1) = (X_1/Z_1, Y_1/Z_1)$ and $Q = (x_2, y_2) = (X_2/Z_2, Y_2/Z_2)$. Derivation is expectedly more complex than for doubling, so we

use the below due to Washington [52, p. 42]:

$$x_S = \frac{UW}{U^3 Z_1 Z_2}, \quad y_R = \frac{T(U^2 X_1 Z_2 - W) - U^3 Y_1 Z_2}{U^3 Z_1 Z_2},$$
$$\text{where:} \quad T = Y_1 Z_2 - Y_2 Z_1, \quad U = X_1 Z_2 - X_2 Z_1,$$
$$W = T^2(Z_1 Z_2) - U^2(X_1 Z_2 + X_2 Z_1).$$
$$(x_S, y_R) \mapsto (UW : T(U^2 X_1 Z_2 - W) - U^3 Y_1 Z_2 : U^3 Z_1 Z_2).$$

- $P + (-P)$: Here $-P = (x_1, -y_1) = (X_1/Z_1, -Y_1/Z_1)$. Using the notation above:

$$T = -2Y_1 Z_1,\ U = 0,\ W = T^2 Z_1^2,$$
$$\implies (x_S, y_R) \mapsto (0 : -TW : 0) \sim (0 : 1 : 0).$$

Hence the point at infinity is represented by $(0 : 1 : 0)$, the only point in projective coordinates on an elliptic curve to have $Z = 0$ [9]. We divide the coordinates by $Z$ to convert them back into affine coordinates, so having $Z = 0$ for the point at infinity is an intuative result. Additionally, the lack of inversions in the above formulae save a significant amount of time computing point operations on all three forms [4, 21, 42].

We know that three points on line add to $O$. This makes more sense in projective coordinates since *every* line will intersect an elliptic curve in three points, counting multiplicity [12]. In an affine curve we may have horizontal lines which intersect it in one place, and we only excluded those because we do not consider lines with less than two points. However, in projective terms we do not need to do that.

Again: what about the point at infinity? We consider the line at $Z = 0$ to be the *line at infinity*. As we know, not every line has three distinct points; in the case of doubling we count the multiplicity of the point twice. The same is true for the point at infinity. It intersects the elliptic curve exactly once with multiplicity three, and is therefore a triple [12]. Vertical lines, when we add $P$ to $-P$, intersect the point at infinity 'off' the visible curve. Now, the notion of $P + (-P) + O = O$ is easier to understand.

Even better performance on all forms can be found using *Jacobian Coordinates* where the triple $(X : Y : Z)$ represents $(X/Z^2, Y/Z^3)$ [21]. The point at infinity is represented by $(1 : 1 : 0)$ and, again, there are no inversions in the point addition formulae [52, p. 43]. Choosing $a = -3$ on Weierstrass curves permits particularly fast addition. Jacobian coordinates are generally recommended for use, and are the reason that the U.S. National Institute of Standards and Technology (NIST) put forward Weierstrass curves with $a = -3$ [51].

Furthermore, using projective coordinates for calculations on a Montgomery curve make it resistant to side channel attacks with little cost [42]. Due to Okeya et. al, point

operations only rely on the number of bits in the private key, not the number of zeros and ones. In affine coordinates, masking the number of operations with added noise is often used, which clearly slows down the process [21]. With projective coordinates, operations are faster *and* naturally resistant [42].

For instance, Coron's dummy addition method essentially extends the Double and Add Algorithm by adding at each iteration, not just when the current binary digit is one [21]. It works by only storing the points added when the digit is one, which is a nearly undetectable operation. However, it is rather slow.

In fact, for Weierstrass curves (therefore, all curves up to birational equivalence) Liardet and Smart have discovered that Jacobian coordinates allow for protection against side channel attacks [32]. Though twisted Edwards curves are resistant in their affine form, projective coordinates also provide more efficient calculation [4]. Bernstein and Lange further improved this with their *inverted projective coordinates*. Here, the point $(X : Y : Z)$ represents $(Z/X, Z/Y)$ in the affine plane. The authors also wrote a detailed comparison of the speeds of operations for many alternative coordinates, revealing that (in 2007) projective Edwards curves were the fastest of all known forms [5]. This was one year after Bernstein released `Curve25519` which held the record for the fastest Diffie-Hellman computations, even including built in attack protection [3].

Implementing such improvements and necessary safeguards is, in reality, hard work for the developer. This is probably the main reason elliptic curve cryptography has not strayed far from the Weierstrass norm. Though papers like this one demonstrate clear differences between forms of curve, it is easier said than done to build a functioning cryptosystem. The short list of names working on the mathematics behind the system produce complex and often intimidating ideas. It is worth investigating how mathematical reports in cryptography trickle down to real world use, since a 12 year old discovery (recent in the mathematical world, ancient in the technological one) has yet to break into mainstream online security.

Indeed, human nature is a factor in the security of systems. Papers on resisting against side-channel attacks largely cite careless implementations as the key issue [9, 21]. For example, Sony's Playstation 3 was infamously hacked when a digital signature parameter was kept static when it should have been randomised on each use[7]. Also, random number generators may not be random at all, leading to predictable outputs of integers such as private keys [46, p. 238]. In 2003, a group including Menezes and Vanstone of the MOV attack discovered new vulnerabilities on systems which do not check whether public keys are actually on the elliptic curve. A single line check could prevent "drastic consequences" [1].

[7]The method and parameter is discussed in [53]. The attack itself was reported widely and is mentioned in [43].

Another attack not considered here is Lenstra's *elliptic curve method* (ECM) for factorisation [46, p. 190]. It is not an attack used on elliptic curves, rather a consequence of elliptic curve arithmetic which is applied to finding prime factors of large numbers [14, p. 335]. It is similar to the Pollard's $p-1$ approach (not to be confused with the above Pollard $\rho$ attack) which can efficiently factorise integers divisible by $p$ where $p-1$ has small factors [46, p. 190]. Such techniques are applicable when $n$ is not prime to reduce the ECDLP and, more widely, to cryptosystems based on the difficulty of factorising, such as RSA.

As Shemanske notes in 2017, ECM is a tool used in factorisation today due to its speed and elegance. It uses pseudocurves which have the same equation as an elliptic curve but over $\mathbb{Z}_n$ for $n$ not prime. Since $n$ is not prime, $\mathbb{Z}_n$ is not a field[8], and the curve itself is not a group [14, p. 336]. The group law is not valid because elements of $\mathbb{Z}_n$ will not necessarily have an inverse, which is needed for finding the slope of the line intersecting two points. Using the Euclidean algorithm for finding modular inverses on such an element will instead output a prime factor [14, p. 337]. Lenstra utilised this fact to develop the algorithm, with a running time dependent on the *least* prime factor of $n$ [46, p. 190].

[8]It is a ring, which has two operations, '+' and '$\cdot$'. It is an abelian group under + while $\cdot$ is associative, distributive, and has an identity.

# Appendix A

# Code

## A.1 Matlab Code

A simple modular inverse function, from [53]:

```
function m=mminv(x,p)
%modular inverse - prime p
x=mod(x,p);
[~,U,~]=gcd(x,p); %second term of the gcd function outputs the
    modular inverse
m=mod(U,p);
```

A function outputting the size of each subgroup generated by each point of an elliptic curve, originally created for [53] and modified for this project:

```
function ES=subgsize(a,b,p)
%outputs all points in E and the size of the subgroup generated
    by each point
E=EllCW(a,b,p); %creates table of points
N=length(E)+1 %number of rational points plus 1 for point at
    infinity, O
ES=[E(:,1),E(:,2),zeros(N-1,1)]; %intialises output table
for ii=1:N-1
    for jj=1:N
        Q=PMulD(E(ii,:),jj,a,b,p); %for each point, multiply by a
            constant jj up to N
        if isscalar(Q) %when the point times jj is O, store jj
            ES(ii,3)=jj; %size of subgroup
           break
        end
    end
end
```

The below functions generate the points on an elliptic curve of Weierstrass, Montgomery, and twisted Edwards form respectively. They check for solutions by calculating left and

right hand side expressions, then storing a point if the coordinates match. Note that they do not include the point at infinity, `inf`, so we must add one in some cases to the size of the set.

```
function W=EllCW(a,b,p)
%Creates W - set of solutions to EC y^2 = x^3 + ax + b (modp)
z=0:p-1;
W=[];
lhs = mod( z.^2, p);
rhs = mod( z.^3+a.*z+b , p);
for ii=1:length(rhs)
    A = find(lhs == rhs(ii));
    for j=1:length(A)
        W=[W;z(ii),z(A(j))];
    end
end
```

```
function M=EllCM(A,B,p)
%Creates M - set of solutions to EC By^2 = x^3 + Ax^2 + x (modp)
x=0:p-1;
y=0:p-1;
M=[];
lhs = mod( B.*(y.^2), p);
rhs = mod( x.^3+A.*(x.^2)+ x , p);
for ii=1:p
    A = find(lhs == rhs(ii));
    for j=1:length(A)
        M=[M;x(ii),y(A(j))];
    end
end
```

```
function E=EllCE(a,d,p)
%Creates E - set of solutions to EC ax^2 + y^2 = 1 + dx^2y^2 (
   modp)
%a=1 for non twisted
x=0:p-1;
y=0:p-1;
E=[];
for ii=1:p
    for jj=1:p
        if mod(a*x(ii)^2 + y(jj)^2 - d*(y(jj)^2)*(x(ii)^2), p) ==
            1
            E=[E;x(ii),y(jj)];
        end
    end
end
```

All the mappings which follow are manual algorithms for small ($p < 10^3$) curves. Weierstrass to Montgomery function:

```
function M=WtoM(a,b,p) %Weierstrass to Montgomery
%Mapping exists when W has a point of order 2
%Equivalently when y=0, cubic has a root, and 3(root)^2 + a is a
    square
K=(0:p-1); %all integers in Z_p
W=subgsize(a,b,p); %orders of each point in W
N=length(W); %stores number of points in W
M=zeros(N,2); %initialises table
F=find(W(:,3) == 2); %identifies point(s) of order 2

if isempty(F)
    error('No point of order 2, cannot be mapped')
end

r=W(F,1); %possible roots of cubic
rc=mod(r.^3 + a.*(r) + b, p); %sub. into cubic equation
G=find(rc==0,1); %identify roots
alpha=W(W(:,1)==r(G) & W(:,2)==0,1); %chooses single root
s1=mod(3*alpha^2 + a,p); %calculates part of s
s2=K(mod(K.^2,p)==s1); %finds sqrt of s1
s=mminv(s2(end),p); %finds mod inverse of above, uses largest
    value if many

A=mod(3*alpha*s,p); %stores A
B=s; %stores B
for ii=1:N %for each point in W, maps to a point in M
    xj=W(ii,1);
    yj=W(ii,2);
    oj=W(ii,3);
    if and(xj==alpha, yj==0)
        M(ii,:)=[0,0]; %if the point is the point of order 2, map
            to 0,0 of M
    else   %otherwise, follow general rule
    xi=mod(s*(xj-alpha),p);
    yi=mod(s*yj,p);
    M(ii,:)=[xi,yi]; %stores point
    end
end
C=['M:',num2str(B),'y^2=x^3+',num2str(A),'x^2 + x']; %curve
    equation of M
disp(C);
disp('An isomorphism!'); %all W to M mappings are isomorphic

Q= input('Create plot? Y/N: ','s');
```

```
41 if strcmp(Q,'N')
42     return
43 else
44 %--- Plotting Function ---%
45     plot(M(:,1),M(:,2),'.b', 'markersize', 20)
46     hold on
47     axis([0 p -1 p])
48     if p<40
49         set(gca, 'Xtick', 0:1:p)
50         set(gca, 'Ytick', 0:1:p)
51         grid on
52     else
53         set(gca, 'Xtick', 0:10:p)
54         set(gca, 'Ytick', 0:10:p)
55         grid on
56     end
57     xlabel('x')
58     ylabel('y')
59     title(['$$',num2str(B),'y^2=x^3+',num2str(A),'x^2 + x $$ mod
          $$', num2str(p), '$$'], ...
60         'FontSize', 20, 'FontWeight','normal', 'interpreter','
             latex')
61     for ii=1:length(M)
62         text(M(ii,1),M(ii,2), ['  ',num2str(ii),],'FontSize',16,
             'HorizontalAlignment','left')
63     end
64     hold off
65 end
```

Montgomery to Weierstrass function:

```
1  function W=MtoW(A,B,p) %Montgomery to Weierstrass
2  M=EllCM(A,B,p); %generates points of Montgomery curve
3  N=length(M); %stores number of points
4  W=zeros(N,2); %initialises output
5  for ii=1:length(M) %maps each point in M to a point in W
6      xj=M(ii,1);
7      yj=M(ii,2);
8      if isinf(xj) %maps point at infinity
9          W(ii,:)=[inf,inf]; %O -> O
10     else %otherwise follow formulae
11     xi=mod(xj*mminv(B,p) + A*mminv(3*B,p),p);
12     yi=mod(yj*mminv(B,p),p);
13     W(ii,:)=[xi,yi]; %stores each point
14     end
15 end
16 a=mod((3-A^2)*mminv(3*(B^2),p), p); %calculates a
```

```
b=mod((2*(A^3) - 9*A)*mminv(27*(B^3),p), p); %calculates b
C=['W: y^2=x^3+',num2str(a),'x+',num2str(b)]; %curve equation for
    W
disp(C);
disp('An isomorphism!'); %all M to W mappings are isomorphic

Q= input('Create plot? Y/N: ','s');
if strcmp(Q,'N')
    return
else
%Plot function here -- omitted for space
```

Twisted Edwards to Montgomery function:

```
function M=EtoM(a,d,p) %Tw. Edwards to Montgomery
%a and d are distinct, non zero:
if a==d
    error('a = d')
elseif a==0
    error('a=0')
elseif d==0
    error('d=0')
end

E=EllCE(a,d,p); %generates points of tw. Edwards curve, E
N=length(E); %stores number of points in E
M=zeros(N,2); %initialises output
A=mod(2*(a+d)*mminv(a-d,p),p); %calculates A
B=mod(4*mminv(a-d,p),p); %calculates B
for ii=1:N %maps each point in E to a point in M
    xj=E(ii,1);
    yj=E(ii,2);
    %exceptional points
    if xj==0
        if yj==mod(-1,p)
        M(ii,:)=[0,0]; % (0,-1) -> (0,0)
        elseif yj==1
        M(ii,:)=[inf,inf]; % (0,1) -> O
        end
    elseif isinf(xj) %either y=0 or x=-1
        q=mod((0:p-1).^2,p); %stores squares of Zp
        if yj == 2
            F=find(q==mod((A+2)*(A-2),p),1); %manually finds sqrt
            G1=mod(mminv(2,p)*(-A + F + 1),p); %positive x coord.
            G=find(M(:,1) == G1,1); %checks not already in M
            if isempty(G)
                M(ii,:)=[G1,0]; %if not in M, assign point
```

```
34              else
35                  G2=mod(mminv(2,p)*(-A -(F + 1),p);
36                  M(ii,:)=[G2,0];%otherwise is in M, assign
                        negative x coord.
37              end
38          elseif yj == 4
39              F=find(q==mod(mminv(B,p)*(A-2),p),1); %manually finds
                    sqrt
40              G=find(M(:,2) == F+1,1); %checks positive y coord.
                    not in M
41              if isempty(G)
42                  M(ii,:)=[-1,F+1]; %if not in M, assign point
43              else
44                  M(ii,:)=[-1,mod(-(F+1),p)]; %otherwise in M,
                        assign negative x coord.
45      end
46      %non exceptional points
47      else
48      xi=mod((1+yj)*mminv(1-yj,p),p);
49      yi=mod((1+yj)*mminv(xj*(1-yj),p),p);
50      M(ii,:)=[xi,yi]; %stores each point
51      end
52  end
53  C=['M:',num2str(B),'y^2=x^3+',num2str(A),'x^2 + x']; %curve
        equation of M
54  disp(C);
55
56  Q= input('Check if isomorphism? Y/N: ','s');
57  if strcmp(Q,'N')
58      return
59  else %Isomorphism if a is square, and d is not
60      q=mod((0:p-1).^2,p);
61      F=find(q==a, 1); %manually finds sqrt of a
62      G=find(q~=d, 1); %manually checks there are no sqrts of d
63      if and(isempty(F)==0,isempty(G)==0)
64          disp('An isomorphism!')
65      else
66          disp('Not an isomorphism :(')
67      end
68  end
69
70  Q= input('Create plot? Y/N: ','s');
71  if strcmp(Q,'N')
72      return
73  else
74  %Plot function here -- omitted for space
```

Montgomery to twisted Edwards function:

```
function E=MtoE(A,B,p) %Montgomery to Tw. Edwards, A not in
    {-2,2}, B non zero
%Isomorphism if a is square, d is nonsquare:
if (A<mod(3,p) && A>mod(-2,p))
    error('A in {-2,2}')
elseif B == mod(0,p)
        error('B = 0')
end

M=EllCM(A,B,p); %generates points of Montgomery curve M
N=length(M)+1; %includes point at infinity
E=zeros(N,2); %initialises output
for ii=1:length(M) %maps each point in E to a point in M, apart
    from inf
    xj=M(ii,1);
    yj=M(ii,2);
    %exceptional points
    if yj==0
        if xj==0
        E(ii,:)=[0,mod(-1,p)];%(0,0) -> (0,-1)
        else
        E(ii,:)=[inf,2];%y=0 -> O
        end
    elseif xj == mod(-1,p)
        E(ii,:)=[inf,4];%x=-1 -> O
    else
    %non exceptional points
    xi=mod(xj*mminv(yj,p),p);
    yi=mod((xj-1)*mminv(xj+1,p),p);
    E(ii,:)=[xi,yi]; %stores each point
    end
end
E(N,:)=[0,1]; %(0,1) -> O, as inf not included in orginal set
a=mod((A+2)*mminv(B,p), p); %calculates a
d=mod((A-2)*mminv(B,p), p); %calculates d
C=['E:',num2str(a),'x^2+y^2=1+',num2str(d),'x^2y^2']; %curve
    equation of E
disp(C);

Q= input('Check if isomorphism? Y/N: ','s');
if strcmp(Q,'N')
    return
else
%as above
    q=0:p-1;
```

```
43      q=mod(q.^2,p);
44      F=find(q==a, 1);
45      G=find(q~=d, 1);
46      if and(isempty(F)==0,isempty(G)==0)
47          disp('An isomorphism!')
48      else
49          disp('Not an isomorphism :(')
50      end
51  end
52
53  Q= input('Create plot? Y/N: ','s');
54  if strcmp(Q,'N')
55      return
56  else
57  %Plot function here -- omitted for space
```

The Weierstrass to twisted Edwards function is a composition of the above functions:

```
1  function E=WtoE(a,b,p) %Weierstrass to Tw. Edwards
2  %below taken from WtoM function but avoids lengthy output by
       calling it
3  K=(0:p-1);
4  W=subgsize(a,b,p);
5  F=find(W(:,3) == 2);
6
7  if isempty(F)
8      error('No point of order 2, cannot be mapped')
9  end
10
11 r=W(F,1);
12 rc=mod(r.^3 + a.*(r) + b, p);
13 G=find(rc==0);
14 alpha =W(W(:,1)==r(G) & W(:,2)==0);
15 s1=mod(3*alpha^2 + a,p);
16 s2=K(mod(K.^2,p)==s1);
17 s=mminv(s2(end),p);
18 A=mod(3*alpha*s,p);
19 B=s;
20
21 %calls MtoE function
22 E=MtoE(A,B,p);
```

Similarly in the inverse twisted Edwards to Weierstrass function:

```
1  function W=EtoW(a,d,p) %Tw. Edwards to Weierstrass
2  %below taken from EtoM function but avoids lengthy output by
       calling it
3  A=mod(2*(a+d)*mminv(a-d,p),p);
4  B=mod(4*mminv(a-d,p),p);
```

```
%calls MtoW function
W=MtoW(A,B,p);
```

## A.2 Python Functions

Using Python, we may simply import some built in functions, such as the modular inverse:

```
from sympy import mod_inverse as mminv
from sympy import isprime as isprime
import math
import random
```

As defined in Section 2.3, we generate the Legendre Symbol:

```
def legendres(dis,p):
    """legendre symbol"""
    ls = pow(dis, (p-1)//2, p)
    if ls == -1 % p:
        return -1
    else: return ls
```

The modular square root function is as follows:

```
def modroot(a,p):
    """Finds the square root of a modulo p"""
    a = a%p #reduces mod p
    ls=legendres(a,p) #checks Legendre Symbol
    if isprime(p) == 0:
        print("p not prime") #rejects if p not prime
        return
    if ls != 1:
        print("No real root") #rejects if Legendre Symbol is -1
        return False
    if p % 4 == 3: #if p mod 4 = 3, then sqrt of a must be +/- a^(p+1)/4
        b = pow(a, (p+1)//4, p)
        return b % p, -b % p
    else: #otherwise, go to TS algorithm
        return tonellishanks(a,p)
```

Which calls the Tonelli-Shanks algorithm:

```
def tonellishanks(a,p):
    """Tonelli-Shanks algorithm for modular square roots"""
    #function is only ever called by modroot,
    #so we need not check the Legendre Symbol
    t = (p-1)//2 #initialises t
    s = 1 #initialises s
```

```
    while t % 2 == 0:
        t = t//2 #divides t by two until the odd factor is found
        s = s+1 #stores power of two
    u = 2 #initialises u
    while legendres(u,p) == 1:
        u = random.randint(0,p) #chooses new u until a
                                #non-square is found
    v = pow(u,t,p) #computes v
    b = pow(a,(t+1)//2, p) #computes first guess
    #Now we find m - one binary digit at a time
    m = 0 #initialises m
    for ii in range(s-1): #from 0 to s-2:
        mii = pow(a,t*2**(s-2-ii),p) #computes value to decide
                                     #if m_ii is 1 or 0
        if mii == 1:     mi=0        #stores m_ii
        else:    mi=1
        m = (m + mi*(2**(ii))) % p #computes m up to
                                   #most current binary digit
    b = b*pow(v,m,p) % p #computes positive solution
    if b**2 % p != a % p: #double check
        b=tonellishanks(a,p)[0] #repeat if incorrect
    return b, -b % p
```

The modular cube root function:

```
def modcube(a,p):
    """Finds the cube root of a modulo p ---
    algorithm by Prof. Cameron"""
    if p % 3 == 1 and p % 9 != 1: #Cameron method
        c = pow(a, (p - 1)//3, p) #check if cube root exists
        if c != 1: return  False #if not, reject
        else:
            if p%9==4: #if p mod 9 = 4...
                k = (2*p + 1)//9 #k can be given as this
            elif p%9==7: #if p mod 9 = 7...
                k = (p+2)//9 #k can be given as this
            else:
                k = random.randint(0,(p+1)/2) #otherwise choose random k
                while True:
                    k = k+1 #increase k until...
                    if 3*k % ((p-1)//3) == 1: #suitable k is found
                        break
            b = pow(a,k,p) #compute cube root
        return b
    elif p % 3 == 2: #we can use short formula:
        b = pow(a, (2*p - 1)//3, p)
        return b
```

The function to identify a point of order two on a Weierstrass curve, even of very large order:

```
def w2check(a,b,p):
    """Checks for point of order two on WNF curve"""
    """Returns x coordinate alpha if exists"""
    dis = -16*(4 * a**3 + 27 * b**2) % p #calculates discriminant
    ldis = legendres(dis,p) #finds Legendre Symbol of discriminant
    if dis == 0:
        print("Discriminant = 0") #rejects if discriminant is 0
        return
    if ldis % 2 == 1:
        #1 root if -1, 3 if 1, doing both cases here
        #using cubic formulae:
        r = ((b**2)*mminv(4,p) + (a**3)*mminv(27,p)) % p
        rt = modroot(r,p) #sqrt of r
        if not rt:#if sqrt does not exist, no alpha exists
            print("No transformation")
            return
        else:
            rt = rt[0] #assign one root to use in algebra
        bf = ((-b)*mminv(2,p)) % p #cubic formulae
        alpha = modcube(bf+rt,p) + modcube(bf-rt,p) #cubic formulae
        if (alpha**3 + a*alpha + b) % p == 0:
            return alpha % p #double checking value of alpha
        else:
            print("error") #for error checking
        return alpha % p
    else:
        print("No point of order two")
        return
```

The curve mapping functions are as follows:

```
def wtom(a,b,p):
    """Converts WNF curve to Montgomery"""
    """Checks for conversion"""
    alpha = w2check(a,b,p) #finds point of order two on W
    if not alpha:
        print("No conversion") #if no such point, reject
        return
    else:
        if not modroot(3 * alpha**2 + a, p): #checks square root exists
            print("No conversion") #otherwise, reject
            return
        B = mminv(max(modroot(3 * alpha**2 + a, p)), p) #calulates B
        A = 3*alpha*B % p #calulates A
        print(B,"y**2 = x**3 +",A,"x**2 + x") #prints curve equation
```

```
        print("An Isomorphism!") #all W to M mappings are isomorphisms
        return A,B,alpha
```

---

```
def mtow(A,B,p):
    """Converts Montgomery curve to WNF"""
    #conversion always exists - no need to check
    a = ((3-A**2)*mminv(3*B**2,p)) % p #calculates a
    b = ((2*A**3 - 9*A)*mminv(27*B**3,p)) % p #calculates b
    print("y**2 = x**3 + ",a, "x +", b) #prints curve equation
    print("An Isomorphism!") #all M to W mappings are isomorphisms
    return a,b
```

---

```
def mtoe(A,B,p):
    """Converts Montgomery curve to twisted Edwards"""
    """Checks for conversion"""
    if A < 3 % p and A > -2 % p: #checks requirement of A
        print("A in {-2,2}") #if fails, reject
        return
    elif B == 0 % p: #checks B non zero
        print("B = 0") #otherwise reject
        return
    a = ((A+2)*mminv(B,p)) % p #calculates a
    d = ((A-2)*mminv(B,p)) % p # calculates d
    print(a,"x**2 + y**2 = 1 + ",d,"x**2y**2") #prints curve equation
    if legendres(a,p) == legendres(d,p):
        print("Not an Isomorphism :(") #not isomorphic if \\
                                       #a & d both square/not square
    elif legendres(a,p) == 1:
        print("An Isomorphism!") #isomorphic if a square, d not square
    elif legendres(a,p) == -1:
        print("Not an Isomorphism :(") #not isomorphic if a not square
    return a,d
```

---

```
def etom(a,d,p):
    """Converts twisted Edwards to Montgomery"""
    """Checks for conversion"""
    if a == d: #checks parameters distinct
        print("a=d") #otherwise reject
        return
    elif a == 0 or d == 0: #checks parameters non zero
        print("Must be non zero") #otherwise reject
        return
    A = (2*(a+d)*mminv(a-d,p)) % p #calculates A
    B = (4*mminv(a-d,p)) % p #calculates B
    print(B,"y**2 = x**3 +",A,"x**2 + x") #prints curve equation
    if legendres(a,p) == legendres(d,p):
```

```
        print("Not an Isomorphism :(") #not isomorphic if \\
                                       #a & d both square/not square
    elif legendres(a,p) == 1:
        print("An Isomorphism!") #isomorphic if a square, d not square
    return A,B
```

```
def etow(a,d,p):
    """Converts twisted Edwards to Weierstrass"""
    A,B = etom(a,d,p)
    a,b = mtow(A,B,p)
    return a,b
```

```
def wtoe(a,b,p):
    """Converts Weierstrass to twisted Edwards """
    A,B = wtom(a,b,p)
    a,d = mtoe(A,B,p)
    return a,d
```

The point mapping functions are as follows:

```
def pwtom(p1,a,b,p,A,B):
    """Transforms a Weierstrass point to a Montgomery point"""
    x1,y1 = p1 #assigns coordinates
    if math.isinf(p1[0]): return math.inf,1 #inf -> inf
    if y1**2 % p != (x1**3 + a*x1 + b) % p: #checks if point is on curve
        return False #otherwise reject
    if not A: #if user does not input A, calulates it
        s,alpha = wtom(a,b,p)[1],wtom(a,b,p)[2] #finds B, alpha from wtom
        if not alpha: #checks if a point of order two exists
            print("No conversion") #otherwise reject
            return
    else:
        alpha = A*mminv(3*B,p) % p #if user inputs A,B calculate alpha
        s = B #as above
    #exceptional points
    if p1 == (alpha,0): return 0,0 #if point has order two, map to 0,0
    #general points
    else:
        return s*(x1-alpha) % p, s*y1 % p
```

```
def pmtow(p1,A,B,p):
    """Transforms a Montgomery point to a Weierstrass point"""
    x1,y1 = p1 #assigns coordinates
    if B*y1**2 % p != (x1**3 + A*x1**2 + x1) % p:
    #^checks point is on curve
        return False #otherwise reject
    #exceptional points
```

```
    if math.isinf(x1): return math.inf,1 #inf -> inf
    #general points
    else:
        return (x1*mminv(B,p) + A*mminv(3*B,p)) % p, y1*mminv(B,p) % p
```

---

```
def pmtoe(p1,A,B,p):
    """Transforms a Montgomery point to a twisted Edwards point"""
    if A < 3 % p and A > -2 % p: #checks requirements on A
                                 #for curve mapping
        print("A in {-2,2}") #if fails, reject
        return False
    elif B == 0 % p: #checks requirements on B for curve mapping
        print("B = 0") #if fails, reject
        return False
    x1,y1 = p1 #assigns coordinates
    if math.isinf(x1): return 0,1 #inf -> (0,1)
    if B*y1**2 % p != (x1**3 + A*x1**2 + x1) % p:
    #^checks point is on curve
        return False #otherwise reject
    #exceptional points
    elif y1 == 0 % p:
        print(y1)
        if x1 == 0 % p: return 0, -1 % p #(0,0) -> (0,-1)
        elif 2*x1 % p == (-A + modroot((A+2)*(A-2),p)[0]) % p:
                return math.inf,2 #point of order two on M ->
                                  #inf ('positive' root)
        elif 2*x1 % p == (-A - modroot((A+2)*(A-2),p)[0]) % p:
                return math.inf,-2 #point of order two on M ->
                                   #inf ('negative' root)
        else: return
    elif x1 == -1 % p:
        if y1 == modroot((A-2)*mminv(B,p),p):
            return math.inf,4 #point of order four on M ->
                              #inf ('positive' root)
        elif y1 == -modroot((A-2)*mminv(B,p),p) % p:
            return math.inf,-4 #point of order four on M ->
                               #inf ('negative' root)
    #general points
    else:
        return x1*mminv(y1,p) % p, (x1-1)*mminv(x1+1,p) % p
```

---

```
def petom(p1,a,d,p):
    """Transforms a twisted Edwards point to a Montgomery point"""
    if a == d: #checks curve mapping requirements
        print("a=d") #if fails, reject
        return False
```

```
    elif a == 0 or d == 0:
        print("Must be non zero")
        return False
    x1,y1 = p1 #assigns coordinates
    #exceptional points
    if math.isinf(x1):
        A = (2*(a+d)*mminv(a-d,p)) % p #calculates A
        B = (4*mminv(a-d,p)) % p #calculates B
        if y1 % p == 2:
            #point of order two on M ('positive' root):
            x = (-A + modroot((A+2)*(A-2),p)[0])*mminv(2,p) % p
            return x,0
        elif y1 % p == -2 % p:
            #point of order two on M ('negative' root):
            x = (-A - modroot((A+2)*(A-2),p)[0])*mminv(2,p) % p
            return x,0
        elif y1 % p == 4:
            #point of order four on M ('positive' root)
            y = modroot((A-2)*mminv(B,p),p)
            return -1,y
        elif y1 % p == -4 % p:
            #point of order four on M ('negative' root)
            y = -modroot((A-2)*mminv(B,p),p) % p
            return -1,y
        else: return #no other points at infinity
                     #should exist on a twisted Edwards curve
    if (a*x1**2 + y1**2) % p != (1 + d*(x1**2)*(y1**2)) % p:
    #^ checks point is on the curve
        return False #otherwise reject
    elif x1 % p == 0 and y1 % p == 1: return math.inf,1 #(0,1) -> inf
    elif x1 % p == 0 and y1 % p == -1 % p: return 0,0 #(0,-1) -> (0,0)
    #general points
    else:
        return (1+y1)*mminv(1-y1,p) % p, (1+y1)*mminv(x1*(1-y1),p) % p
```

---

```
def petow(p1,a,d,p):
    """Transforms a twisted Edwards point to a Weierstrass point"""
    if a == d: #checks curve mapping requirements
        print("a=d") #if fails, reject
        return
    elif a == 0 or d == 0:
        print("Must be non zero")
        return
    x1,y1 = p1 #assigns coordinates
    #exceptional points
    if math.isinf(x1):
```

```
        #computes inf separately, since may be (inf,4) or (inf,2)
        p2 = petom(p1,a,d,p) #computes point at infinity ->
                             #point on montgomery
        A,B = (2*(a+d)*mminv(a-d,p)) % p, (4*mminv(a-d,p)) % p #A and B
        p3 = pmtow(p2,A,B,p) #computes point on montgomery -> point on W
        return p3
    if (a*x1**2 + y1**2) % p != (1 + d*(x1**2)*(y1**2)) % p:
    #^checks point is on curve
        return False #otherwise reject
    elif x1 % p == 0 and y1 % p == 1:
        return math.inf,1 #(0,1) -> inf -> inf
    else:
        p2 = petom(p1,a,d,p) #computes point on Montgomery
        A,B = (2*(a+d)*mminv(a-d,p)) % p, (4*mminv(a-d,p)) % p #A and B
        p3 = pmtow(p2,A,B,p) #computes point on W
        return p3
```

```
def pwtoe(p1,a,b,p):
    """Transforms a Weierstrass point to a Montgomery point"""
    x1,y1 = p1 #assigns coordinates
    if math.isinf(x1):
        return 0,1 #inf -> inf -> (0,1)
    if y1**2 % p != (x1**3 + a*x1 + b) % p: #checks point is on curve
        return False #otherwise reject
    else:
        A,B,a1 = wtom(a,b,p) #calculate Montgomery parameters
        p2 = pwtom(p1,a,b,p,A,B) #calculate point on Montgomery
        p3 = pmtoe(p2,A,B,p) #calculate point on twisted Edwards
        return p3
```

The point addition functions are as follows:

```
def wadd(p1,p2,a,b,p):
    """Adds p1 and p2 on a WNF curve"""
    if math.isinf(p1[0]) and math.isinf(p2[0]):
        return math.inf,1 #O + O = O
    if math.isinf(p1[0]): return p2 #O + p2 = p2
    if math.isinf(p2[0]): return p1 #p1 + O = p1
    else:
        x1,y1 = p1 #assigns coordinates
        x2,y2 = p2 #assigns coordinates
        if y1**2 % p != (x1**3 + a*x1 + b) % p:
        #^checks if point is on curve
            return False #otherwise reject
        if y2**2 % p != (x2**3 + a*x2 + b) % p:
        #^checks if point is on curve
            return False #otherwise reject
```

```
        if x1==x2 and -y2 % p == y1:
            return math.inf,1 #(x1,y1) + (x1,-y1) = O
        elif x1==x2:
            #point doubling
            sl = ((3 * x1**2 + a)*mminv(2*y1,p)) % p #slope
            x = (sl**2 - x1 - x2) % p
            return x, -(sl * (x - x1) + y1) % p
        else:
            #point addition
            sl = (y2-y1)*mminv(x2-x1,p) % p #slope
            x = (sl**2 - x1 - x2) % p
            return x, -(sl * (x - x1) + y1) % p
```

---

```
def madd(p1,p2,A,B,p):
    """Adds p1 and p2 on a Montgomery curve"""
    if B*(A**2 - 4) == 0 % p: #checks Montgomery requirements
        print("Singular Curve")
        return
    if math.isinf(p1[0]) and math.isinf(p2[0]):
        return math.inf,1 #O + O = O
    if math.isinf(p1[0]): return p2 #O + p2 = p2
    if math.isinf(p2[0]): return p1 #p1 + O = p1
    else:
        x1,y1 = p1 #assigns coordinates
        x2,y2 = p2 #assigns coordinates
        if B*y1**2 % p != (x1**3 + A*x1**2 + x1) % p:
        #^checks if point is on curve
            return False #otherwise reject
        if B*y2**2 % p != (x2**3 + A*x2**2 + x2) % p:
        #^checks if point is on curve
            return False #otherwise reject
        if x1==x2 and -y2 % p == y1:
            return math.inf,1 #(x1,y1) + (x1,-y1) = O
        elif x1==x2:
            #point doubling
            sl = ((3 * x1**2 + 2*A*x1 + 1)*mminv(2*B*y1,p)) % p #slope
            x = (B*(sl**2) - 2*x1 - A) % p
            return x, (sl*(x1 - x) - y1) % p
        else:
            #point addition
            sl = (y2-y1)*mminv(x2-x1,p) % p #slope
            x = (B*(sl**2) - x1 - x2 - A) % p
            return x, (sl*(x1 - x) - y1) % p
```

---

```
def eadd(p1,p2,a,d,p):
    """
```

```
    Adds p1 and p2 on a twisted Edwards curve
    We assume that there exists a pt of order 4, so identity is (0,1)
    """
    if d*(1 - d) == 0 % p: #checks Edwards requirements
        print("Singular Curve")
        return
    x1,y1 = p1
    x2,y2 = p2
    if (a*x1**2 + y1**2) % p != (1 + d*(x1**2)*(y1**2)) % p:
    #^checks point is on the curve
        return False #otherwise reject
    if (a*x2**2 + y2**2) % p != (1 + d*(x2**2)*(y2**2)) % p:
    #^checks point is on the curve
        return False #otherwise reject
    if p1 == (0,1) and p2 == (0,1):
        return 0,1 #O + O = O
    if p1 == (0,1): return p2 #O + p2 = p2
    if p2 == (0,1): return p1 #p1 + O = p2
    if y1==y2 and -x2 % p == x1: #(x1, y1) + (-x1, y1) = O
        return 0,1
    else:
        #point addition
        dxy = d*x1*x2*y1*y2 % p
        return (x1*y2 + y1*x2)*mminv(1+dxy,p) % p, \\
        (y1*y2 - a*x1*x2)*mminv(1-dxy,p) % p
```

The point multiplication functions (using the Double and Add algorithm) are as follows:

```
def wmul(p1,m,a,b,p):
    """Multiply p1 by m on a WNF curve"""
    """Works - prelim checks"""
    kp = math.inf,1 #starts at O
    q = p1 #starts at 1P
    k = bin(m)[2:] #converts m to binary
    for ii in reversed(k): #reverses binary digits
        if ii == '1': #if k(ii) = 1, add q to kp
            kp = wadd(kp,q,a,b,p)
        q=wadd(q,q,a,b,p) #double q each iteration
    return kp
```

```
def mmul(p1,m,A,B,p):
    """Multiply p1 by m on a Montgomery curve"""
    kp = math.inf,1 #starts at O
    q = p1 #starts at 1P
    k = bin(m)[2:] #converts m to binary
    for ii in reversed(k): #reverses binary digits
```

```
        if ii == '1': #if k(ii) = 1, add q to kp
            kp = madd(kp,q,A,B,p)
        q=madd(q,q,A,B,p) #double q each iteration
    return kp
```

```
def emul(p1,m,a,d,p):
    """Multiply p1 by m on a twisted Edwards curve"""
    kp = 0,1 #starts at O
    q = p1 #starts at 1P
    k = bin(m)[2:] #converts m to binary
    for ii in reversed(k): #reverses binary digits
        if ii == '1': #if k(ii) = 1, add q to kp
            kp = eadd(kp,q,a,d,p)
        q=eadd(q,q,a,d,p) #double q each iteration
    return kp
```

# Appendix B

# Algebra

Below is a summary of the author's attempt to prove analogousness of point addition on a twisted Edwards curve and on a Montgomery curve (hence, also a Weierstrass curve). The points are distinct non-inverses and not exceptional. On a Montgomery curve, the points have coordinates $P = (u_1, v_1)$ and $Q = (u_2, v_2)$. The equivalent points on a twisted Edwards curve have $P_T = (x_1, y_1)$, $Q_T = (x_2, y_2)$. The addition formulae compute $P + Q = (u_R, v_R) = (u_S, -v_S)$ which is equivalent to $(x_R, y_R) = (-x_S, y_S)$.

First, we find the slope on the Montgomery curve $M$ in terms of the twisted Edwards curve $T$:

$$\lambda_{aM} = \frac{v_2 - v_1}{u_2 - u_1} = \frac{\frac{1+y_2}{x_2(1-y_2)} - \frac{1+y_1}{x_1(1-y_1)}}{\frac{1+y_2}{1-y_2} - \frac{1+y_1}{1-y_1}} \tag{B.1}$$

$$= \frac{x_1(1+y_2)(1-y_1) - x_2(1+y_1)(1-y_2)}{2x_1x_2(y_2 - y_1)}. \tag{B.2}$$

Similarly to (2.30), we have the required addition relation in the same form as for doubling, apart from the above $\lambda_{aM}$:

$$\frac{u_1}{x_1} + \frac{u_S}{x_R} = \lambda_{aM}(u_1 - u_S). \tag{B.3}$$

Where the coordinates are given by:

$$u_1 = \frac{1+y_1}{1-y_1}, \tag{B.4}$$

$$u_S = \frac{1+y_S}{1-y_S} = \frac{1 + \frac{y_1y_2 - ax_1x_2}{1-dx_1x_2y_1y_2}}{1 - \frac{y_1y_2 - ax_1x_2}{1-dx_1x_2y_1y_2}} \tag{B.5}$$

$$= \frac{y_1y_2 - ax_1x_2 + 1 - dx_1x_2y_1y_2}{ax_1x_2 - y_1y_2 + 1 - dx_1x_2y_1y_2}, \tag{B.6}$$

$$\implies \frac{u_S}{x_R} = \frac{y_1y_2 - ax_1x_2 + 1 - dx_1x_2y_1y_2}{ax_1x_2 - y_1y_2 + 1 - dx_1x_2y_1y_2} \cdot \frac{1 + dx_1x_2y_1y_2}{x_1y_2 + y_1x_2}. \tag{B.7}$$

First we find the left hand side of (B.3) in terms of $T$:

$$\frac{u_1}{x_1} + \frac{u_S}{x_R} = \frac{1+y_1}{x_1(1-y_1)} + \frac{y_1y_2 - ax_1x_2 + 1 - dx_1x_2y_1y_2}{ax_1x_2 - y_1y_2 + 1 - dx_1x_2y_1y_2} \cdot \frac{1 + dx_1x_2y_1y_2}{x_1y_2 + y_1x_2} \tag{B.8}$$

On the right hand side of (B.3) we have:

$$\begin{aligned}\lambda_{aM}(u_1 - u_S) &= \lambda_{aM}\left(\frac{1+y_1}{1-y_1} - \frac{y_1y_2 - ax_1x_2 + 1 - dx_1x_2y_1y_2}{ax_1x_2 - y_1y_2 + 1 - dx_1x_2y_1y_2}\right) \\ &= \lambda_{aM}\left(\frac{-2y_1y_2 + 2ax_1x_2 + 2y_1 - 2dx_1x_2y_1^2y_2}{(1-y_1)(ax_1x_2 - y_1y_2 + 1 - dx_1x_2y_1y_2)}\right).\end{aligned} \tag{B.9}$$

Where $\lambda_{aM}$ can be written as:

$$\frac{x_1(1+y_2)(1-y_1)}{2x_1x_2(y_2-y_1)} + \frac{-x_2(1+y_1)(1-y_2)}{2x_1x_2(y_2-y_1)}. \tag{B.10}$$

If we begin by multiplying $(u_1 - u_S)$ by the right hand fraction of (B.10) above it can be rearranged to:

$$\begin{aligned}&\frac{-x_2(1+y_1)(1-y_2)}{2x_1x_2(y_2-y_1)} \cdot (u_1 - u_S) \\ &= \frac{x_2(1+y_1)(y_2-y_1)(ax_1x_2 - y_1y_2 + 1 - dx_1x_2y_1y_2) + F}{x_1x_2(1+y_1)(y_2-y_1)(ax_1x_2 - y_1y_2 + 1 - dx_1x_2y_1y_2)} \\ &= \frac{1+y_1}{x_1(1-y_1)} + \frac{F}{x_1x_2(1+y_1)(y_2-y_1)(ax_1x_2 - y_1y_2 + 1 - dx_1x_2y_1y_2)} \\ &= \frac{u_1}{x_1} + \frac{F}{x_1x_2(1+y_1)(y_2-y_1)(ax_1x_2 - y_1y_2 + 1 - dx_1x_2y_1y_2)}, \\ F &= ax_1x_2^2y_1^2 - x_2y_1^3y_2 - x_2y_2 + dx_1x_2^2y_1y_2^2 + x_2y_1^2y_2 + x_2y_1y_2 - ax_1x_2^2 - dx_1x_2^2y_1^3y_2^2.\end{aligned}$$

So it now suffices to show that:

$$\begin{aligned}&\frac{F}{x_1x_2(1+y_1)(y_2-y_1)(ax_1x_2 - y_1y_2 + 1 - dx_1x_2y_1y_2)} \\ &\qquad + \frac{x_1(1+y_2)(1-y_1)}{2x_1x_2(y_2-y_1)} \cdot (u_1 - u_S)\end{aligned} \tag{B.11}$$

is equal to $u_S/x_R$. The numerator of this expression is:

$$\begin{aligned} & F + x_1y_1^2y_2^2 - ax_1^2x_2y_1y_2 + ax_1^2x_2y_2 - x_1y_1y_2^2 + x_1y_1 - x_1y_1^2 - x_1^2x_2y_1 + ax_1^2x_2 \\ & + dx_1^2x_2y_1^3y_2^2 + dx_1^2x_2y_1^3y_2 - dx_1^2x_2y_1^2y_2^2 - dx_1^2x_2y_1^2y_2 \qquad \text{(B.12)} \\ = & \, x_1y_1^2(ax_2^2 + y_2^2 - 1) - x_2y_1y_2(ax_1^2 + y_1^2 - dx_1^2y_1^2) + x_2y_2(ax_1^2 + y_1^2 - dx_1^2y_1^2) \\ & + x_1y_1(1 + dx_2^2y_2^2 - y_2^2) - x_2y_2 + x_2y_1y_2 - x_1^2x_2y_1 + ax_1^2x_2 - ax_1x_2^2 \\ & + dx_1^2x_2y_1^3y_2^2 - dx_1^2x_2y_1^2y_2^2. \qquad \text{(B.13)} \end{aligned}$$

We then use the definition of a twisted Edwards curve to substitute the values in brackets:

$$\begin{aligned} & x_1y_1^2(\mathbf{ax_2^2} + \mathbf{y_2^2} - \mathbf{1}) - x_2y_1y_2(\mathbf{ax_1^2} + \mathbf{y_1^2} - \mathbf{dx_1^2y_1^2}) + x_2y_2(\mathbf{ax_1^2} + \mathbf{y_1^2} - \mathbf{dx_1^2y_1^2}) \\ & + x_1y_1(\mathbf{1} + \mathbf{dx_2^2y_2^2} - \mathbf{y_2^2}) - x_2y_2 + x_2y_1y_2 - x_1^2x_2y_1 + ax_1^2x_2 - ax_1x_2^2 \\ & + dx_1^2x_2y_1^3y_2^2 - dx_1^2x_2y_1^2y_2^2 \\ = & -x_1^2x_2y_1 + ax_1^2x_2 - ax_1x_2^2 + \mathbf{a}x_1\mathbf{x_2^2}y_1 + \mathbf{d}x_1\mathbf{x_2^2}y_1^2\mathbf{y_2^2} + dx_1^2x_2y_1^3y_2^2 - dx_1^2x_2y_1^2y_2^2. \\ & \qquad \text{(B.14)} \end{aligned}$$

Now we can write the whole of (B.11) as:

$$\frac{-x_1^2x_2y_1 + ax_1^2x_2 - ax_1x_2^2 + ax_1x_2^2y_1 + dx_1x_2^2y_1^2y_2^2 + dx_1^2x_2y_1^3y_2^2 - dx_1^2x_2y_1^2y_2^2}{x_1x_2(1+y_1)(y_2-y_1)(ax_1x_2 - y_1y_2 + 1 - dx_1x_2y_1y_2)} \qquad \text{(B.15)}$$

$$= \frac{-x_1y_1 + ax_1 - ax_2 + ax_2y_1 + dx_2y_1^2y_2^2 + dx_1y_1^3y_2^2 - dx_1y_1^2y_2^2}{(1+y_1)(y_2-y_1)(ax_1x_2 - y_1y_2 + 1 - dx_1x_2y_1y_2)}. \qquad \text{(B.16)}$$

Which must be equal to:

$$\frac{u_S}{x_R} = \frac{(y_1y_2 - ax_1x_2 + 1 - dx_1x_2y_1y_2)(1 + dx_1x_2y_1y_2)}{(x_1y_2 + y_1x_2)(ax_1x_2 - y_1y_2 + 1 - dx_1x_2y_1y_2)} \qquad \text{(B.17)}$$

$$= \frac{y_1y_2 - ax_1x_2 + 1 - dx_1x_2y_1y_2 + dx_1x_2y_1^2y_2^2 - adx_1^2x_2^2y_1y_2 + dx_1x_2y_1y_2 - (dx_1x_2y_1y_2)^2}{(x_1y_2 + x_2y_1)(ax_1x_2 - y_1y_2 + 1 - dx_1x_2y_1y_2)} \qquad \text{(B.18)}$$

Note that both share $ax_1x_2 - y_1y_2 + 1 - dx_1x_2y_1y_2$ in the denominator. So we multiply the numerator and denominator of (B.16) by $(x_1y_2 + x_2y_1)$ and of (B.18) by $(1+y_1)(y_2-y_1)$ to find an identical denominator: $(1+y_1)(y_2-y_1)(x_1y_2+x_2y_1)(ax_1x_2 - y_1y_2 + 1 - dx_1x_2y_1y_2)$.

The new numerator of (B.16) is given by:

$$-x_1^2y_1y_2 + ax_1^2y_2 - ax_1x_2y_2 + ax_1x_2y_1y_2 + dx_1x_2y_1^2y_2^3 + dx_1^2y_1^3y_2^3 - dx_1^2y_1^2y_2^3$$
$$-x_1x_2y_1^2 + ax_1x_2y_1 - ax_2^2y_1 + ax_2^2y_1^2 + dx_2^2y_1^3y_2^2 + dx_1x_2y_1^4y_2^2 - dx_1x_2y_1^3y_2^2. \quad \text{(B.19)}$$

The new numerator of (B.18) is given by:

$$(1 + dx_1x_2y_1y_2)\cdot$$
$$(-y_1^2y_2 - y_1^3y_2 + y_1y_2^2 + y_1^2y_2^2 + ax_1x_2y_1 + ax_1x_2y_1^2 - ax_1x_2y_2 - ax_1x_2y_1y_2$$
$$-y_1 - y_1^2 + y_2 + y_1y_2 + dx_1x_2y_1^2y_2 + dx_1x_2y_1^3y_2 - dx_1x_2y_1y_2^2 - dx_1x_2y_1^2y_2^2). \quad \text{(B.20)}$$

We can expand some of the terms in (B.19) with the twisted Edwards curve equation:

$$\begin{aligned} -dx_1^2y_1^2y_2(y_2^2) &= (dx_2^2y_2^2 + 1 - ax_2^2)(-dx_1^2y_1^2y_2) \\ &= -d^2x_1^2x_2^2y_1^2y_2^3 - dx_1^2y_1^2y_2 + adx_1^2x_2^2y_1^2y_2, && \text{(B.21)} \\ -dx_1^2y_1^2y_2 &= (ax_1^2 + y_1^2 - 1)(-y_2) = -ax_1^2y_2 - y_1^2y_2 + y_2. && \text{(B.22)} \\ dx_2^2y_1y_2^2(y_1^2) &= (dx_1^2y_1^2 + 1 - ax_1^2)(dx_2^2y_1y_2^2) && \text{(B.23)} \\ &= d^2x_1^2x_2^2y_1^3y_2^2 + dx_2^2y_1y_2^2 - adx_1^2x_2^2y_1y_2^2, \\ dx_2^2y_1y_2^2 &= (ax_2^2 + y_2^2 - 1)(y_1) = ax_2^2y_1 + y_1y_2^2 - y_1. && \text{(B.24)} \end{aligned}$$

We also separate some terms into factors of $(1 + dx_1x_2y_1y_2)$:

$$\begin{aligned} dx_1x_2y_1^2y_2^3 &= y_1y_2^2(1 + dx_1x_2y_1y_2) - y_1y_2^2, && \text{(B.25)} \\ -dx_1x_2y_1^3y_2^2 &= -y_1^2y_2(1 + dx_1x_2y_1y_2) + y_1^2y_2, && \text{(B.26)} \\ -ax_1x_2y_2 &= -ax_1x_2y_2(1 + dx_1x_2y_1y_2) + adx_1^2x_2^2y_1y_2^2, && \text{(B.27)} \\ ax_1x_2y_1 &= ax_1x_2y_1(1 + dx_1x_2y_1y_2) - adx_1^2x_2^2y_1^2y_2. && \text{(B.28)} \end{aligned}$$

Summing these terms back together gives:

$$
\begin{aligned}
& -dx_1^2y_1^2y_2^3 + dx_2^2y_1^3y_2^2 + dx_1x_2y_1^2y_2^3 - dx_1x_2y_1^3y_2^2 - ax_1x_2y_2 + ax_1x_2y_1 \qquad \text{(B.29)} \\
= & -d^2x_1^2x_2^2y_1^2y_2^3 - ax_1^2y_2 - y_1^2y_2 + y_2 + adx_1^2x_2^2y_1^2y_2 \\
& +d^2x_1^2x_2^2y_1^3y_2^2 + ax_2^2y_1 + y_1y_2^2 - y_1 - adx_1^2x_2^2y_1y_2^2 \\
& +(y_1y_2^2 - y_1^2y_2 - ax_1x_2y_2 + ax_1x_2y_1)(1 + dx_1x_2y_1y_2) \\
& -y_1y_2^2 + y_1^2y_2 + adx_1^2x_2^2y_1y_2^2 - adx_1^2x_2^2y_1^2y_2 \\
= & -d^2x_1^2x_2^2y_1^2y_2^3 - ax_1^2y_2 + y_2 + d^2x_1^2x_2^2y_1^3y_2^2 + ax_2^2y_1 - y_1 \\
& +(y_1y_2^2 - y_1^2y_2 - ax_1x_2y_2 + ax_1x_2y_1)(1 + dx_1x_2y_1y_2) \\
= & -ax_1^2y_2 + y_2 + ax_2^2y_1 - y_1 \\
& +(y_1y_2^2 - y_1^2y_2 - ax_1x_2y_2 + ax_1x_2y_1 - dx_1x_2y_1y_2^2 + dx_1x_2y_1^2y_2)(1 + dx_1x_2y_1y_2) \\
& +dx_1x_2y_1y_2^2 - dx_1x_2y_1^2y_2 \\
= & -ax_1^2y_2 + ax_2^2y_1 \\
& +(-y_1 + y_2 + y_1y_2^2 - y_1^2y_2 \\
& -ax_1x_2y_2 + ax_1x_2y_1 - dx_1x_2y_1y_2^2 + dx_1x_2y_1^2y_2)(1 + dx_1x_2y_1y_2). \qquad \text{(B.30)}
\end{aligned}
$$

So, our original expression (B.19) can be written as:

$$
\begin{aligned}
& -x_1^2y_1y_2 + ax_1x_2y_1y_2 + dx_1^2y_1^3y_2^3 - x_1x_2y_1^2 + ax_2^2y_1^2 + dx_1x_2y_1^4y_2^2 \\
& +(-y_1 + y_2 + y_1y_2^2 - y_1^2y_2 \\
& -ax_1x_2y_2 + ax_1x_2y_1 - dx_1x_2y_1y_2^2 + dx_1x_2y_1^2y_2)(1 + dx_1x_2y_1y_2). \qquad \text{(B.31)}
\end{aligned}
$$

Comparing the above with (B.20), we see many of the terms are taken care of. We must now show that:

$$
-x_1^2y_1y_2 + ax_1x_2y_1y_2 + dx_1^2y_1^3y_2^3 - x_1x_2y_1^2 + ax_2^2y_1^2 + dx_1x_2y_1^4y_2^2 \qquad \text{(B.32)}
$$

is equal to:

$$
\begin{aligned}
& (-y_1^2 + y_1y_2 - y_1^3y_2 + y_1^2y_2^2 + ax_1x_2y_1^2 - ax_1x_2y_1y_2 + dx_1x_2y_1^3y_2 - dx_1x_2y_1^2y_2^2) \\
& \cdot (1 + dx_1x_2y_1y_2). \qquad \text{(B.33)}
\end{aligned}
$$

# Bibliography


[1] A. Antipa, D. Brown, A. Menezes, et al. "Validation of elliptic curve public keys". In: *International Workshop on Public Key Cryptography*. Springer. 2003, pp. 211–223.

[2] T. Baigneres, C. Delerablée, M. Finiasz, et al. "Trap Me If You Can-Million Dollar Curve." In: *IACR Cryptology ePrint Archive* 2015 (2015), p. 1249.

[3] D. J. Bernstein. "Curve25519: new Diffie-Hellman speed records". In: *International Workshop on Public Key Cryptography*. Springer. 2006, pp. 207–228.

[4] D. J. Bernstein, P. Birkner, M. Joye, et al. "Twisted Edwards curves". In: *International Conference on Cryptology in Africa*. Springer. 2008, pp. 389–405.

[5] D. J. Bernstein and T. Lange. "Faster Addition and Doubling on Elliptic Curves". In: *Advances in Cryptology – ASIACRYPT 2007*. Ed. by K. Kurosawa. Berlin, Heidelberg: Springer, 2007.

[6] D. J. Bernstein and T. Lange. *SafeCurves: choosing safe curves for elliptic-curve cryptography*. URL: https://safecurves.cr.yp.to (visited on Aug. 6, 2019).

[7] K. Bjoernsen. "Koblitz Curves and its practical uses in Bitcoin security". In: *order (ε (GF (2k)* 2.1 (2009), p. 7.

[8] S. Blake-Wilson. "Information security, mathematics, and public-key cryptography". In: *Designs, Codes and Cryptography* 19 (2000), pp. 77–99.

[9] E. Brier and M. Joye. "Weierstrass elliptic curves and side-channel attacks". In: *International Workshop on Public Key Cryptography*. Springer. 2002, pp. 335–345.

[10] J. B. Buchmann. *Introduction to Cryptography*. New York: Springer-Verlag, 2004.

[11] F. Catanese. "Monodromy and normal forms". In: *Karl Weierstraß (1815–1897)*. Springer, 2016, pp. 195–218.

[12] P. Corn and J. Khim. *Elliptic Curves*. URL: https://brilliant.org/wiki/elliptic-curves/ (visited on Aug. 6, 2019).

[13] C. Costello and B. Smith. "Montgomery curves and their arithmetic". In: *Journal of Cryptographic Engineering* 8.3 (2018), pp. 227–240.

[14] R. Crandall and C. Pomerance. “Elliptic Curve Arithmetic”. In: *Prime Numbers: A Computational Perspective*. Springer, 2005.

[15] *Crypto Review Panel*. URL: https://mailarchive.ietf.org/arch/msg/lwip/2ZYsS9u8tOytMsgMXF_xjdJTt7U (visited on May 4, 2019).

[16] H. Edwards. “A normal form for elliptic curves”. In: *Bulletin of the American Mathematical Society* 44.3 (2007).

[17] J. Falk. *On Pollard's rho method for solving the elliptic curve discrete logarithm problem*. 2019. URL: http://www.diva-portal.org/smash/record.jsf?pid=diva2:1326270 (visited on Aug. 7, 2019).

[18] S. D. Galbraith. “Supersingular curves in cryptography”. In: *International Conference on the Theory and Application of Cryptology and Information Security*. Springer. 2001, pp. 495–513.

[19] V Gayoso Martnez, L Hernndez Encinas, A Martn Muoz, et al. “Secure elliptic curves and their performance”. In: *Logic Journal of the IGPL* 27.2 (Apr. 2019), pp. 277–238. eprint: http://oup.prod.sis.lan/jigpal/article-pdf/27/2/277/28246801/jzy035.pdf. URL: https://doi.org/10.1093/jigpal/jzy035.

[20] H. Ivey-law and R. Rolland. *Finding Cryptographically Strong Elliptic Curves: A Technical Report*. 2010. URL: http://galg.acrypta.com/telechargements/ell_tech_report_public.pdf (visited on Aug. 1, 2019).

[21] T. Izu, B. Möller, and T. Takagi. “Improved elliptic curve multiplication methods resistant against side channel attacks”. In: *International Conference on Cryptology in India*. Springer. 2002, pp. 296–313.

[22] H. Kamarulhaili and L. K. Jie. “Cryptography and Security in Computing”. In: InTech, 2012. Chap. Elliptic Curve Cryptography and Point Counting Algorithms.

[23] N. Koblitz. *A Course in Number Theory and Cryptography, 2nd Ed*. New York: Springer-Verlag, 1994.

[24] N. Koblitz. *Algebraic aspects of cryptography*. Springer, 1998.

[25] N. Koblitz. “Elliptic Curve Cryptosystems”. In: *Mathematics of Computation* 48.177 (1987), p. 203.

[26] N. Koblitz. *Introduction to elliptic curves and modular forms*. Vol. 97. Springer, 1984.

[27] N. Koblitz, A. Menezes, and S. Vanstone. “The state of elliptic curve cryptography”. In: *Designs, codes and cryptography* 19 (2000), pp. 173–193.

[28] F. Kuhn and R. Struik. “Random walks revisited: Extensions of Pollards rho algorithm for computing multiple discrete logarithms”. In: *International Workshop on Selected Areas in Cryptography*. Springer. 2001, pp. 212–229.

[29] A. Langley, M. Hamburg, and S. Turner. *Elliptic Curves for Security*. Internet Research Task Force. 2016. URL: https://tools.ietf.org/html/rfc7748 (visited on Aug. 3, 2019).

[30] A. K. Lenstra. “Integer factoring”. In: *Towards a quarter-century of public key cryptography*. Springer, 2000, pp. 31–58.

[31] F. Leprévost, J. Monnerat, S. Varrette, et al. “Generating anomalous elliptic curves”. In: *Information Processing Letters* 93.5 (2005), pp. 225–230.

[32] P.-Y. Liardet and N. P. Smart. “Preventing SPA/DPA in ECC systems using the Jacobi form”. In: *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer. 2001, pp. 391–401.

[33] D. Liu, T. Song, and Y. Dai. “Isomorphism and generation of Montgomery-form elliptic curves suitable for cryptosystems”. In: *Tsinghua Science & Technology* 10.2 (2005), pp. 145–151.

[34] J. H. Mathews. *Solving Cubic Equations*. California State University Fullerton. 2014. URL: http://mathfaculty.fullerton.edu/mathews/cubics/CubicTutorial.html (visited on June 29, 2019).

[35] U. M. Maurer and S. Wolf. “The Diffie–Hellman protocol”. In: *Designs, Codes and Cryptography* 19 (2000), pp. 147–171.

[36] A. J. Menezes, T. Okamoto, and S. A. Vanstone. “Reducing elliptic curve logarithms to logarithms in a finite field”. In: *iEEE Transactions on information Theory* 39.5 (1993), pp. 1639–1646.

[37] R. Moloney, G. McGuire, and M. Markowitz. “Elliptic Curves in Montgomery Form with B= 1 and Their Low Order Torsion.” In: *IACR Cryptology ePrint Archive* 2009 (2009), p. 213.

[38] P. L. Montgomery. “Speeding the Pollard and elliptic curve methods of factorization”. In: *Mathematics of computation* 48.177 (1987), pp. 243–264.

[39] D. Nguyen. *Correspondence Between Elliptic Curves In Edwards-Bernstein And Weierstrass Forms*. 2017. URL: https://mysite.science.uottawa.ca/mnevins/papers/NguyenMScProj2017.pdf (visited on Mar. 16, 2019).

[40] P. Novotney. *Weak Curves In Elliptic Curve Cryptography*. 2010. URL: modular.math.washington.edu/edu/2010/414/projects/novotney.pdf (visited on Aug. 6, 2019).

[41] A. Odlyzko. “Discrete logarithms: The past and the future”. In: *Designs, Codes and Cryptography* 19 (2000), pp. 129–145.

[42] K. Okeya, H. Kurumatani, and K. Sakurai. “Elliptic curves with the Montgomery-form and their cryptographic applications”. In: *International Workshop on Public Key Cryptography*. Springer. 2000, pp. 238–257.

[43] D. Poddebniak, J. Somorovsky, S. Schinzel, et al. “Attacking deterministic signature schemes using fault attacks”. In: *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE. 2018, pp. 338–352.

[44] P. Rogaway and T. Shrimpton. “Cryptographic Hash-Function Basics: Definitions, Implications, and Separations for Preimage Resistance, Second-Preimage Resistance, and Collision Resistance”. In: *Fast Software Encryption*. Vol. 3017. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2004.

[45] A. Shallue and C. E. van de Woestijne. “Construction of rational points on elliptic curves over finite fields”. In: *International Algorithmic Number Theory Symposium*. Springer. 2006, pp. 510–524.

[46] T. R. Shemanske. *Modern Cryptography and Elliptic Curves*. Vol. 83. American Mathematical Soc., 2017.

[47] J. H. Silverman. *The Arithmetic of Elliptic Curves*. 2nd ed. Springer, 2009.

[48] R. Skuratovskii. “Edwards curve counting method and supersingular Edwards curves”. In: *arXiv preprint arXiv:1811.12544* (2018).

[49] M. L. Sommerseth and H. Hoeiland. *Pohlig-Hellman Applied in Elliptic Curve Cryptography*. 2015. URL: http://koclab.cs.ucsb.edu/teaching/ecc/project/2015Projects/Sommerseth+Hoeiland.pdf (visited on Aug. 1, 2019).

[50] D. R. Stinson. *Cryptography: Theory and Practice, 2nd Ed*. Chapman and Hall/CRC, 2002.

[51] R. Struik. *Alternative Elliptic Curve Representations*. 2019. URL: https://tools.ietf.org/html/draft-ietf-lwig-curve-representations-06 (visited on July 17, 2019).

[52] L. Washington. *Elliptic Curves: Number Theory and Cryptography, Second Edition*. Discrete Mathematics and Its Applications. CRC Press, 2008.

[53] M. Wood. *Mathematics of Bitcoin*. 2018. URL: https://github.com/mirandavcw/Mathematics-of-Bitcoin (visited on Feb. 15, 2019).