BINARY EDWARDS CURVES IN ELLIPTIC CURVE CRYPTOGRAPHY

by

Graham Enos

A dissertation submitted to the faculty of
The University of North Carolina at Charlotte
in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in
Applied Mathematics

Charlotte

2013

Approved by:

Dr. Yuliang Zheng

Dr. Gabor Hetyei

Dr. Thomas Lucas

Dr. Evan Houston

Dr. Shannon Schlueter

ABSTRACT

GRAHAM ENOS. Binary edwards curves in elliptic curve cryptography. (Under the direction of DR. YULIANG ZHENG)

Edwards curves are a new normal form for elliptic curves that exhibit some cryptographically desirable properties and advantages over the typical Weierstrass form. Because the group law on an Edwards curve (normal, twisted, or binary) is *complete* and *unified*, implementations can be safer from side channel or exceptional procedure attacks. The different types of Edwards provide a better platform for cryptographic primitives, since they have more security built into them from the mathematic foundation up.

Of the three types of Edwards curves—original, twisted, and binary—there hasn't been as much work done on binary curves. We provide the necessary motivation and background, and then delve into the theory of binary Edwards curves. Next, we examine practical considerations that separate binary Edwards curves from other recently proposed normal forms. After that, we provide some of the theory for binary curves that has been worked on for other types already: pairing computations. We next explore some applications of elliptic curve and pairing-based cryptography wherein the added security of binary Edwards curves may come in handy. Finally, we finish with a discussion of `e2c2`, a modern C++11 library we've developed for Edwards Elliptic Curve Cryptography.

ACKNOWLEDGMENTS

I'd like to thank my advisor, Dr. Yuliang Zheng, for his guidance and support. His knowledge, insight, and willingness to work with me helped me reach the realization and completion of this immense undertaking. I am also grateful to Dr. Gabor Hetyei for serving as my mathematics advisor while I worked on this project, despite his very busy schedule. The entire Mathematics Department at UNC Charlotte was very supportive and taught me a lot, for which I am very thankful. My parents have never stopped supporting me; they fostered a curiosity and love of learning in me that got me to where I am. Last but certainly not least, my wonderful wife Jessica helped me in more ways than I can count. Without her love and support, I would not have finished this dissertation.

DEDICATION

For Jess, who is always right.

TABLE OF CONTENTS

CHAPTER 1: INTRODUCTION

The group of rational points on an elliptic curve over a finite field has proven very useful in cryptography since Miller and Koblitz first suggested its use independently in the 1980s ([59] and [48]). Due to the lack of subexponential algorithms to solve the Discrete Logarithm Problem in this group, elliptic curve cryptography cryptosystems tend have to have a level of security comparable to other ElGamal-type systems (e.g. ones that use the DLP, or more precisely the Diffie Hellman problem, over the multiplicative group $\mathbb{F}_p^*$; see [75]) while using much smaller key sizes. Moreover, the group of points on an elliptic curve E over a finite field K is rather nice to work with; it's isomorphic to either a cyclic group or the direct product of two cyclic groups and (at least in the typical Weierstrass coordinates) has a simple geometric interpretation.

There is some room for improvement, however. Typically the group operation on E involves a number of special cases, all of which must be checked for at every turn:

- What if one point is the neutral element, the so-called "point at infinity?"
- What if the two points are the same? What if their x-coordinate is zero?
- What if the two points are inverses of each other?

In each of these cases, the exception to the usual formula can cause implementations to giving up more information to outside observers than intended—leaking "side channel information." That is, though the theory of elliptic curve cryptography is perfectly sound from a mathematical standpoint, in practice it is either less secure (or at least more complicated) than originally thought due to shortcomings in the theory's applicability. Such types of attacks have been outlined in papers like [12] and [45], and considerable effort has been spent trying to make Weierstrass curve implementations secure "after the fact," as it were; see e.g. [61].

In 2007, Dr. Harold Edwards discussed a new normal form for elliptic curves in [25]. Despite his paper not focusing on cryptography, the normal form put forth by Edwards has very desirable cryptographic properties that help combat the leakage of side-channel information from the very start; as noted by Bernstein and Lange in [7], the group law is *complete* and *unified*, two terms we will discuss later. Moreover, in many cases the group law involves less operations, meaning that the more secure computations involved can also be faster. While this is not the case over binary fields, the benefits of the law's completeness make the loss of speed seem negligible; in fact, [62]'s authors argue that with specialized hardware the speed difference can be greatly reduced, while the completeness of the binary Edwards curve group law actually makes it faster than Weierstrass implementations that must constantly check for special cases. Add to this the reduced complexity of implementation code, and binary Edwards are just as competitive as their non-binary counterparts.

Though there has been significant work to build up the literature on Edwards curves, there is still room to explore the cryptographic and mathematical aspects of these new normal forms. In this dissertation we do just that, specifically focusing on binary Edwards curves (which have been explored less than other types). Our work is as follows: in Chapter 2, we begin with the necessary background on elliptic curves, elliptic curve cryptography, and two of the three types of Edwards curves. Then in Chapter 3, we build on the theoretical foundation of the previous Chapter to discuss binary Edwards curves. Next, in Chapter 4, we examine the practical considerations involved in applying that theory in cryptographic practice, including a discussion of the shortcomings of four recently proposed normal forms. After that, we explore pairing computations over binary Edwards curves in 5. In 6, we focus on two applications of binary Edwards curves to cryptography: password-based key derivation functions and a compartmented secret sharing scheme with signcryption. Finally, in Appendix 7 we discuss `e2c2`, a modern computer software library written in C++11 to perform Edwards elliptic curve cryptography built on top of Shoup's `NTL` [70]; the sourcecode for `e2c2` is included in appendix 7.4.4.

CHAPTER 2: BACKGROUND

In this chapter, we provide the background necessary to understand the cryptographic importance of binary Edwards curves. We begin with a brief discussion of elliptic curves in general. Since we are mostly interested in the application of elliptic curves and pairing computations, we will stick to a lighter summary rather than going into deep mathematical detail; i.e. we will follow the example of [79] more than [71]. We recommend these two books, along with other references in the bibliography, to readers interested in a more in-depth background.

2.1 Elliptic Curves

2.1.1 Weierstrass Curves

Broadly speaking, elliptic curves are "curves of genus one having a specified base point" ([71]). After appropriate scaling, such curves are usually written in generalized Weierstrass coordinates in the homogeneous form

$$Y^2Z + a_1XYZ + a_3YZ^2 = X^3 + a_2X^2Z + a_4XZ^2 + a_6Z^3$$

where X, Y and Z are taken to be projective coordinates from $\mathbb{P}^2$ over some base field K and $a_1, \ldots, a_6$ are scalars from the algebraic closure $\overline{K}$ (though often they're just taken to be elements of K itself). For ease of notation, we often work in nonhomogeneous affine coordinates instead, taking $x = {}^{X}/_{Z}$ and $y = {}^{Y}/_{Z}$:

$$y^2 + a_1xy + a_3y^2 = x^3 + a_2x^2 + a_4x + a_6$$

These two forms are interchangeably called the *Weierstrass* form of the curve. If $char(K) \notin \{2,3\}$, then we usually simplify further to

$$y^2 = x^3 + Ax + B$$

after a further change of coordinates (though of course we won't be able to do this when working with binary curves, i.e. curves over finite fields of characteristic two). We also specify a special point, denoted by ∞ or $\mathcal{O}$, with the projective coordinates $(0 : 1 : 0)$.[1] For fields K with $char(K) = 2$, Weierstrass curves are usually written in the form

$$y^2 + xy = x^3 + a_2x + a_6$$

We typically only work with *non-singular* curves. That is, we don't allow the curve to have multiple roots; we choose our constants such that

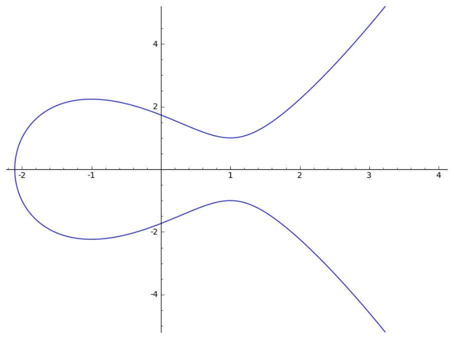
$$4A^3 + 27B^2 \neq 0$$

This inequality comes from examining the discriminant of the curve in simplified Weierstrass form, viz. $\Delta = -16(4A^3 + 27B^2)$. For more, see section III.1 of [71].

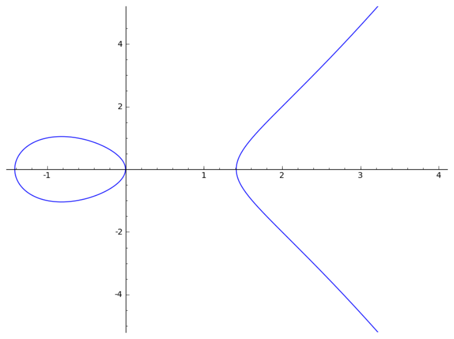
If a curve is non-singular, i.e. its discriminant Δ is nonzero, then it indeed has genus 1 and is, if taken over the complex numbers $\mathbb{C}$, isomorphic to a torus. Geometrically speaking, a non-singular curve has three distinct roots over K, so it doesn't have

[1]We'll use ∞ to refer to this point, and reserve $\mathcal{O}$ for the neutral element on an Edwards curve.

a *cusp*—which occurs when all three roots are the same—or a *node*—which occurs when only two of the roots are the same. This will be important when we define the group law next. See Figure 2.1 for the graphs of two non-singular curves, and compare them to the graphs of the singular curves in (2.2). The first curve in Figure 2.2 has a cusp, while the second has a node.

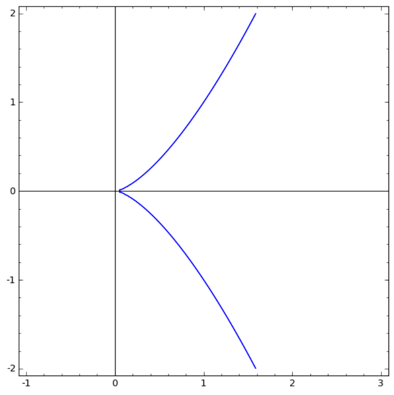


(a) $y^2 = x^3 - 3x + 3$

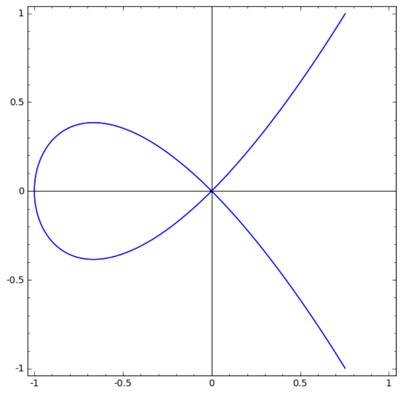


(b) $y^2 = x^3 - 2x$

Figure 2.1: Two Non-singular Elliptic Curves over $\mathbb{R}$



(a) $y^2 = x^3$



(b) $y^2 = x^3 + 2x^2$

Figure 2.2: Two Singular Elliptic Curves over $\mathbb{R}$

From here on in, we will use the notation $E(K)$ to specify an elliptic curve E over a field K, or just E if the field K is understood. That is,

$$E(K) = \{\infty\} \cup \{(x, y) \in K \times K \mid y^2 + a_1xy + a_3y^2 = x^3 + a_2x^2 + a_4x + a_6\}$$

keeping in mind that we include the "point at infinity" ∞ among the rational points as well.

2.1.2 The Group Law

As it turns out, there's a relatively easy to understand way to define a group law on $E(K)$. In this section, we'll mostly be working with $\mathbb{R}$ as our field for ease of notation, understanding, and graphing, but keep in mind that any field K works just as well. Given two points P and Q on E over $\mathbb{R}$ with rational coordinates, connect them with a line; that line will (barring a few special cases) intersect the graph of E at a third point R' with rational coordinates. Then $R = P + Q$ is defined to be the other point where the vertical line through R' intersects the graph of E. In Figure 2.3, the red line connects P to Q on the curve $E : y^2 = x^3 - 3x^2 + 4$. This line hits the curve at R' in the first quadrant; the green line connects this to $P + Q = R$.

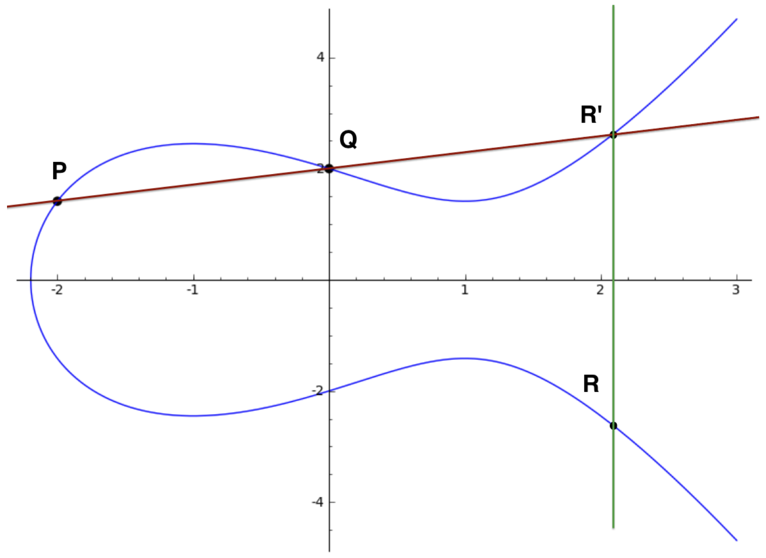


Figure 2.3: Weierstrass Group Law

To go from our "chord-and-tangent" geometric understanding to an algebraic one, we can use the slope of the red line (via a implicit differentiation of E's equation, if necessary) to find R', then the equation of the curve to find R. In special cases where the red line is vertical, we either have $P + (-P) = \infty$, the point at infinity (the identity element of our group), or $P + \infty = P$. This leads us to the following for curves given in short Weierstrass form:

Theorem 2.1 (Weierstrass Group Law). The following formulas defining the addition of $P = (x_1, y_1)$ and $Q = (x_2, y_2)$ on $E(\mathbb{R}) : y^2 = x^3 + Ax + B$ turns the points $E(\mathbb{R})$ into an abelian group:

1. If $P = \infty$, $P + Q = Q$.
2. If $Q = \infty$, $P + Q = P$
3. If $x_1 \neq x_2$, $P + Q = (x_3, y_3)$ where
$$x_3 = m^2 - x_1 - x_2, \qquad y_3 = m(x_1 - x_3) - y_1, \qquad \text{and } m = \frac{y_2 - y_1}{x_2 - x_1}$$
4. If $x_1 = x_2$ but $y_1 \neq y_2$, $P + Q = \infty$
5. If $P = Q$ and $y_1 \neq 0$, $P + Q = (x_3, y_3)$ where
$$x_3 = m^2 - 2x_1, \qquad y_3 = m(x_1 - x_3) - y_1, \qquad \text{and } m = \frac{3x_1^2 + A}{2y_1}$$
6. If $P = Q$ and $y_1 = 0$, $P + Q = \infty$

The proof of this theorem, though not terribly difficult, is a bit tedious. As such we will omit it, but it can be found in any text on elliptic curves or elliptic curve cryptography, e.g. [79], [71] (section III.2), [49], [19], etc. Again, such a proof would work for any field of characteristic not equal to two or three; in those cases, a similar set of formulas can be found. Of course, in finite fields our geometric understanding doesn't exactly hold any more—it is rather difficult to graph in $\mathbb{F}_{8675309}$, for example, but the algebraic structure still holds. Similar theorems work for fields of characteristic 2 or 3. One other thing to note about the above group law is that it involves a number of special cases; there isn't a single simple law that holds for any two points P and $Q \in E$, and the outcome of $P + Q$ highly depends on the relationships between P, Q, and ∞. This will be important to remember when we contrast Weierstrass curves with Edwards curves later on.

Over a finite field $K = \mathbb{F}_q$ where $q = p^n$ for some prime p and $n \in \mathbb{N} \setminus \{0\}$, similar algebraic work yields an abelian group of $\mathbb{F}_q$-rational points on E. This group is very nice to work with; more precisely, we have the following ([39]'s Theorem 3.12):

Theorem 2.2 (Group Structure of $E(\mathbb{F}_q)$). Let E be an elliptic curve defined over $\mathbb{F}_q$. Then $E(\mathbb{F}_q)$ is isomorphic to the direct sum of cyclic groups $\mathbb{Z}_{n_1} \oplus \mathbb{Z}_{n_2}$ for some uniquely determined n_1 and $n_2 \in \mathbb{N}$ such that $n_2 | n_1$ and $n_2 | (q - 1)$.

Since cyclic groups are generated by a single element, the fact that $E(\mathbb{F}_q)$ is isomorphic to the direct sum of two cyclic groups (or one, if $n_2 = 1$) make them very nice to work with computationally. The exact details of this isomorphism are rather difficult to specify, however. Except for specific groups that have thoroughly examined (or worked on via Schoof's Algorithm, see [75]), the best we can easily find are some bounds on $\#E(\mathbb{F}_q)$, the order of the group $E(\mathbb{F}_q)$—even though it will of course be $n_1 \cdot n_2$—given by the following theorem from the 1930s:

Theorem 2.3 (Hasse's Theorem). The order of the group $E(\mathbb{F}_q)$ satisfies the inequality

$$q + 1 - 2\sqrt{q} \leq \#E(\mathbb{F}_q) \leq q + 1 + 2\sqrt{q}$$

In some respects, the fact that the isomorphism $E(\mathbb{F}_q) \to C_{n_1} \oplus C_{n_2}$ isn't completely ironed out can seem frustrating. However, this—along with some other aspects to be discussed shortly—is exactly what makes them suitable candidates for cryptographic primitives.

2.2 Elliptic Curve Cryptography

In two groundbreaking papers ([48] and [59]) Koblitz and Miller independently suggested using the group of rational points on an elliptic curve as the basis for a public key cryptosystem. In its simplest form, elliptic curve cryptography uses the *Discrete Logarithm Problem* and the *Diffie-Hellman Problem* to hide private information in public. We'll give two descriptions of the discrete logarithm problem: one in a multiplicative group, like $\mathbb{F}_p^*$, and one in an additive one like $E(\mathbb{F}_q)$:

Problem 2.1 (DLP in multiplicative group). Given a group $(G, \cdot)$, an element $\alpha \in G$ of order n, and an element $\beta \in \langle \alpha \rangle$, find the unique element $k \in \mathbb{Z}_n$ such that $\alpha^k = \beta$.

We'll borrow [75]'s notation and say that $k = \log_\alpha \beta$.

Problem 2.2 (DLP in additive group). Given a group $(G, +)$, an element $\alpha \in G$ of order n, and an element $\beta \in \langle \alpha \rangle$, find the unique element $k \in \mathbb{Z}_n$ such that $k\alpha = \beta$.

As [75] says, "The utility of the Discrete Logarithm problem in a cryptographic setting is that finding discrete logarithms is (probably) difficult, but the inverse operation of exponentiation can be computed efficiently." While this description favors multiplicative groups, the idea is the same in an additive one: repeated applications of the group operation are (relatively) simple to compute, but taking the result of such a computation and trying to find the input that yields it is (believed to be) rather difficult. We'll say more on the "believed to be" qualification shortly.

ElGamal cryptosystems are based on the apparent difficulty of the following problem, first proposed in [24]:

Problem 2.3 (DHP in multiplicative group). Given a group $(G, \cdot)$, an element $\alpha \in G$ of order n, and two elements $\beta = \alpha^b$ and $\gamma = \alpha^c$ in $\langle \alpha \rangle$ for some integers $b, c \in \mathbb{Z}_n$, find $\delta = \alpha^{bc}$.

Problem 2.4 (DHP in additive group). Given a group $(G, +)$, an element $\alpha \in G$ of order n, and two elements $\beta = b\alpha$ and $\gamma = c\alpha$ in $\langle \alpha \rangle$ for some $b, c \in \mathbb{Z}_n$, find $\delta = (bc)\alpha$.

It's apparent that should one find a fast algorithm to compute discrete logs, the Diffie-Hellman Problem will also be rather simple to find—once $\log_\alpha \beta$ and $\log_\alpha \gamma$ are found, δ can be computed from these two values, since

$$\delta = \alpha^{bc} = \alpha^{(\log_\alpha \beta)(\log_\alpha \gamma)}$$

in multiplicative notation, or

$$\delta = (bc)\alpha = [(\log_\alpha \beta)(\log_\alpha \gamma)]\alpha$$

in additive notation. Hence the DHP is no harder than the DLP; in many groups, it is believed to be as difficult.

The above exposition of the Diffie-Hellman problem is also called the *Computational Diffie-Hellman Problem*, to contrast it with the *Decisional Diffie-Hellman Problem*:

Problem 2.5 (DDHP in additive group)*.* Given a triple (β, γ, δ) from a group $(G, +)$ of order n such that $\beta = b\alpha$ and $\gamma = c\alpha$ for some b, c chosen independently at random from $\mathbb{Z}_n$, determine which of the following two cases hold:

1. $\delta = (bc)\alpha$
2. $\delta = d\alpha$ for some other d chosen at random from $\mathbb{Z}_n$ independently from b and c.

This problem seems easier to solve at first blush than its computational counterpart, but is still rather difficult in many settings. We rely upon the difficulty of this problem to construct a cryptosystem in Chapter 6.

We now describe how these problems are used for the ElGamal public key cryptosystem in $\mathbb{F}_p^*$ using two favorite characters from cryptographic literature: Alice and Bob.[2]

Cryptosystem 2.6 (ElGamal over $\mathbb{F}_p^*$)*.* Suppose Alice wishes to set up a secure way for Bob to send her a message. They first agree on a large prime p and a group element $\alpha \in \mathbb{F}_p^*$ that is a generator of the cyclic subgroup of quadratic residues (i.e. the set of γ for which $\exists \beta \in \mathbb{F}_p^*$ such that $\beta^2 = \gamma$; this requirement is conjectured to equate the semantic security of the cryptosystem to solving the DLP). Next, Alice chooses an integer a (reduced modulo the order of α) as her secret key and publishes α and $\beta = \alpha^a$.

To send a message (that's been encoded as an integer $m < p$ in some public fashion)

[2]The following exposition is Cryptosystem 6.1 from [75].

securely to Alice, Bob chooses a secret integer b, computes

$$(\gamma, \delta) = (\alpha^b, m\beta^b)$$

and transmits them to Alice.

Once she has the pair (γ, δ), Alice computes

$$\delta(\gamma^a)^{-1} = (m\beta^b)(\alpha^{ab})^{-1} = m\alpha^{ab}\alpha^{-ab} = m$$

and recovers the message.

The security of this system, as mentioned above, is conjectured to be equivalent to the feasibility of solving the discrete logarithm problem in this group. That is, this system is secure as long as computing a from α and β is difficult. To achieve this, [75] states that because of rather efficient methods (subexponential time, but not polynomial time) for computing $\log_\alpha \beta$ in $\mathbb{F}_p^*$ like the index calculus algorithm, "p needs to be at least 2^{1880}" for this system to securely hide message until the year 2020.[3] As we'll see, the DLP is currently much more secure in $E(\mathbb{F}_q)$ for much smaller key sizes.

In elliptic curve cryptography, the size of our finite field $q = p^n$ for some prime p, the details of the curve E, and a special point P are chosen in a manner such that the order of P is a prime equal to $\#E/h$ for some small cofactor $h = 1, 2$, or 4. Typically $q = p$ (so $n = 1$) if p is a large odd prime, while n is chosen large in the binary case where $p = 2$; see [30] for more details. As above, Alice chooses a secret a (reduced modulo P's order) and publishes both P and $Q = aP$.

[3][75] was published in 2005.

However, implementers of elliptic curve cryptography are currently able to choose much smaller parameters than there counterparts in other settings, as [75] explains:

> In order for an elliptic curve discrete logarithm based cryptosystem to be secure until the year 2020, it has been suggested that one should take $p \approx 2^{160}$ [if p is odd] (or $n \approx 160$ [if $p = 2$]). In contrast, p needs to be at least 2^{1880} [in the F_p^* case] to achieve the same (predicted) level of security. The reason for this significant difference is the lack of a known index calculus attack on elliptic curve discrete logarithms.

Because the faster DLP algorithms in $\mathbb{F}_p^*$ are too specialized to work in $E(\mathbb{F}_q)$, would-be attackers are (currently) forced to use more generalized algorithms that are much slower, viz. exponential in the bit-length of our parameters. The following table from [63] describes the state of affairs as of 2009:

Symmetric Key	RSA & Diffie-Hellman Key	Elliptic Curve Key
80	1024	160
112	2048	224
128	3072	256
192	7680	384
256	15360	521

Table 2.1: NIST recommended key sizes (measured in number of bits)

As you can see from table (2.1), elliptic curves provide a strong alternative to other primitives for public key cryptosystems. They can make use of this speed and ease of implementation in the following cryptosystem to securely share a session key for, say, a communication via a faster symmetric system like AES [21]:

Cryptosystem 2.7 (Diffie-Hellman Key Exchange). Suppose Alice and Bob wish to construct a secret key with which they can communicate privately. To start, they select a finite field $\mathbb{F}_q$ and a curve $E : y^2 = x^3 + Ax + B \bmod p$ (again, $p > 3$). They also agree on a generator P of a large subgroup $\langle P \rangle$ of $E(\mathbb{F}_q)$.

On her own, Alice selects a secret random element $a \in \mathbb{F}_q$ and computes $aP = (x_a, y_a)$ and sends it to Bob over an unsecured channel. Likewise, Bob picks a secret b and computes $bP = (x_b, y_b)$ and sends it to Alice. Then, since $a(bP) = b(aP)$, they now have a shared secret key with which to communicate.

Moreover, an eavesdropper (Eve, say) would need either a or b to be able to recover their shared key. However, all Eve can see is aP and bP, so she'd need to compute a discrete logarithm to recover a or b from these multiples of P. If instead she hopes to compute abP from aP and bP, she needs to solve the Diffie-Hellman problem, which is also believed to be quite difficult.

We conclude this section with an example of using elliptic curves for a Diffie-Hellman key exchange. Computations in this section were done with the help of the Sage computer algebra system [74].

Example 2.8. To start, they follow [30]'s recommendations and select their finite field to be $\mathbb{F}_p$ where

$$p = 6277101735386680763835789423207666416083908700390324961279$$

is a prime of bit length 192. They then pick their curve E to be $y^2 = x^3 - 3x + B \bmod p$, where

$$B = 2455155546008943817740293915197451784769108058161191238065$$

Then per [30] their generator P is the point (x, y), where

$$x = 602046282375688656758213480587526111916698976636884684818$$

$$y = 174050332293622031404857552280219410364023488927386650641$$

Alice selects her secret random element $a \in \mathbb{F}_p$ to be

$$a = 559962322125394764833828889736075397576141188711443672469$$

She then computes $aP = (x_a, y_a)$ where

$$x_a = 592050352229379625430810432976474658203570824303592435263 7$$

$$y_a = 491673305705631220345147984414988575263686332486289555267 0$$

and sends it to Bob (over an unsecured channel). Likewise, Bob selects the secret random $b \in \mathbb{F}_p$ where

$$b = 218835988203114010177961198245001686548169155309222579693 3$$

and sends $bP = (x_b, y_b)$ to Alice, where

$$x_b = 763794420572335512449180732023082145057259616577775194615$$

$$y_b = 679692871035831936640161559745188598378636802338357105905$$

Their shared key is now

$$x = 171483500642289668200766892567964393143004835071470584998 2$$

$$y = 446495041758821250575576484761798123940522910742751783855 4$$

Here's the transcript of a Sage session for this computation:[4]

Listing 2.1: Diffie-Hellman Example

```
K = GF(6277101735386680763835789423207666416083908700390324961279)
B = Integer(''.join("64210519 e59c80e7 0fa7e9ab 72243049 feb8deec c146b9b1"), 16)
E = EllipticCurve(K, [-3, B])
x = Integer(''.join("188da80e b03090f6 7cbf20eb 43a18800 f4ff0afd 82ff1012"), 16)
y = Integer(''.join("07192b95 ffc8da78 631011ed 6b24cdd5 73f977a1 1e794811"), 16)
P = E(x, y)
a = Integer(K.random_element())
b = Integer(K.random_element())
print(a * (b * P) == b * (a * P))
```

The hexadecimal strings for B, x, and y, along with the choice of p, are all given in [30].

2.3 Edwards Curves

Inspired by studying the work of Euler and Gauss, Edwards found a new normal form of elliptic curves in his 2007 paper [25]. "In notes published posthumously in his *Werke*," writes Edwards, "Gauss stated explicitly the formulas Euler had hinted at earlier" for an explicit addition formula on the curve $x^2 = y^2 + x^2y^2 = 1$ which related to the integral

$$\int \frac{dx}{\sqrt{1 - x^4}}$$

[4]Note that the calls to `K.random_element()` will in all likelihood yield different values for a and b than those given above.

Gauss was studying via the change of coordinates $z = y(1+x^2)$. Gauss wrote the formulas as

$$S = \frac{sc' + s'c}{1 - ss'cc'} \qquad C = \frac{cc' - ss'}{1 + ss'cc'}$$

As Edwards notes in [25], "Gauss's choice of the letters s and c brings out the analogy with the addition laws for sines and cosines." Edwards generalized these laws to the curve

$$x^2 + y^2 = a^2 + a^2x^2y^2 \tag{2.1}$$

over a field K where $a \in K$ is a constant such that $a^5 \neq a$. An example of a curve of this type is given in Figure 2.4.

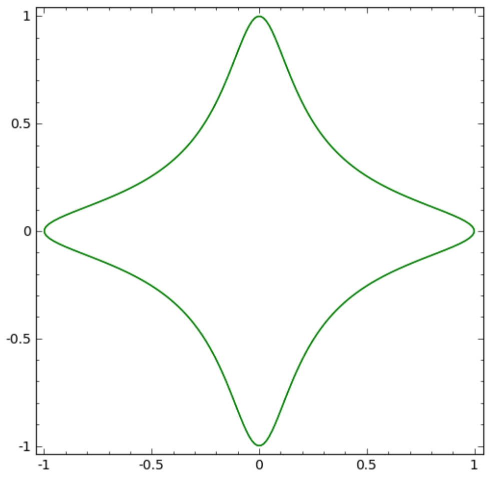


Figure 2.4: Edwards Curve

For the curve in equation 2.1, we have the following group law:

Theorem 2.4. (Edwards Addition Law) If a is a constant for which $a^5 \neq a$, the formulas

$$X = \frac{1}{a} \cdot \frac{xy' + yx'}{1 + xyx'y'} \qquad Y = \frac{1}{a} \cdot \frac{yy' - xx'}{1 - xyx'y'}$$

describe the addition formula for the elliptic curve

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

This is theorem (3.1) in [25]; the reader is referred there for a proof. By simple inspection, one can see that the neutral element for this curve is $(0, a)$. Moreover, the inverse $-P$ of the point $P = (x, y)$ is $(-x, y)$:

$$\begin{aligned}(x, y) + (-x, y) &= \left(\frac{1}{a} \cdot \frac{xy - yx}{1 - x^2y^2}, \frac{1}{a} \cdot \frac{y^2 + x^2}{1 + x^2y^2}\right) \\ &= \left(0, \frac{1}{a} \cdot \frac{a^2 + a^2x^2y^2}{1 + x^2y^2}\right) \\ &= (0, a)\end{aligned}$$

using the curve equation in the intermediate step.

As Edwards points out in his proposition (5.1), every curve of the form given in equation 2.1 is *birationally equivalent* to an elliptic curve in Weierstrass form. That is, following Edwards' advice "to abandon the notion of *points of* a curve and to work instead with *rational functions of* a curve, one can consider two curves birationally equivalent "if their fields of rational functions are isomorphic." We'll discuss this idea in more depth when we go into the details of Bernstein and Lange's exploration of Edwards curves.

The addition law given in the above theorem is much simpler than the equivalent Weierstrass one given in an earlier section. There are no special cases, no changing the rules depending on whether P is the identity element or $P = -Q$. We do of course lose the simple geometric description of Weierstrass curves[5], but that is a small price

[5]Though as we'll see in Chapter 5, there is another geometric interpretation of the group law

to pay for so simple, symmetric, and elegant a group law. As we'll see in the next section, the Edwards curve group law's superiority is more than just aesthetic: it has desirable consequences for elliptic curve cryptographic schemes that use Edwards curves as their group.

2.4 Bernstein & Lange: ECC potential

In [7], Bernstein and Lange generalize Edwards' original curve to more cases and turn their attention to cryptographic viability. First, though, we explain the necessary algebraic geometry.

2.4.1 Algebraic Geometry

Strictly speaking, a curve is "a projective variety of genus one" and dimension one with a distinguished rational point ([71]), so working more in depth with elliptic curves requires some understanding of algebraic geometry. We give a cursory sketch of the necessary pieces based upon [71] and [40]; readers who wish for more in-depth coverage of these topics are referred to these texts.

First off, in many cases it makes sense to work with projective coordinates (as mentioned above) instead of affine ones. As we saw in the case of Weierstrass curves, sometimes this is not just for convenience; we need projective coordinates to be able to discuss the point at infinity that arises from adding two points with the same x-coordinate, for example.

Definition 2.9. *Projective n-space* over a field K, denoted by $\mathbb{P}^n(K)$ or just $\mathbb{P}^n$, is the

here.

set of all $n + 1$ tuples in regular affine coordinates

$$(X_0, \ldots, X_n) \in \mathbb{A}^{n+1}$$

such that at least one X_i is nonzero, together with the following equivalence relation:

$$(X_0, \ldots, X_n) \sim (Y_0, \ldots, Y_n)$$

if $\exists \lambda \in \overline{K}^*$ such that $X_i = \lambda Y_i$ for each i. We denote an equivalence class by $(X_0 : \ldots : X_n)$, and the individual X_i are called *homogeneous* coordinates.

Next up is the concept of a *variety.* Though [71] and [40] subscribe to stricter (or at least more precise) definitions, for our purposes the ideas from [1] will suffice.

Definition 2.10. Given a polynomial f from $\mathbb{P}^n(K)$ to K, the *variety* $V(f)$ is the set of solutions of the equation $f = 0$. More formally,

$$V(f) = \{(X_0 : \ldots : X_n) \in \mathbb{P}^n \mid f(X_0 : \ldots X_n) = 0\}$$

Next we define rational maps between varieties.

Definition 2.11. Let $V_1, V_2 \subset \mathbb{P}^n$ be projective varieties. A *rational map* from V_1 to V_2 is a map of the form

$$\varphi : V_1 \to V_2, \varphi = (f_0 : \ldots : f_n)$$

where the f_i have the property that for every point $P \in V_1$ for which all of $f_0, \ldots, f_n$ are defined,

$$\varphi(P) = (f_0(P) : \ldots : f_n(P)) \in V_2$$

Note that a rational map $\varphi : V_1 \to V_2$ need not be a well-defined function at every point in V_1; however, it may be possible to replace each f_i with gf_i for some other rational function g to evaluate φ at a troublesome point $P \in V_1$.

Finally, we come to birational maps.

Definition 2.12. A *birational map* is a rational map that admits an inverse; i.e. a rational map $\varphi : V_1 \to V_2$ for which there is another rational map $\psi : V_2 \to V_1$ such that, when defined, $\varphi \circ \psi$ and $\psi \circ \varphi$ are the identity map. If there is a birational map from a variety V_1 to a variety V_2, we say that these two varieties are *birationally equivalent.*

Birational equivalence gives a looser sort of connection between two varieties (or curves, since that's what we are focused on) than strict isomorphism. Basically, two varieties are birationally equivalent if, except for a handful of points, they are isomorphic. In algebraic geometry, singularities of birational maps are typically handled by "blowing up"[6] the maps at those points to resolve them. If a point (x_0, y_0) is a singularity of a map φ, we can set $y = tx$ for some variable t and evaluate what happens as $y \to y_0$. We'll show an example of this when we discuss binary Edwards curves in Chapter 3; for more, see a text on algebraic geometry like [40].

2.4.2 Bernstein & Lange's Edwards Curves

As the authors mention in the start of [7], "Every elliptic curve over a non-binary field is birationally equivalent to a curve in Edwards form over an extension of the field, and in many cases over the original field." Because "every Edwards curve has a point of order 4," to be birationally equivalent to a curve without such a point,

[6]Think "blowing up a balloon," not "blowing up Wile E. Coyote."

such as “the NIST curves over prime fields,” may require working over an extension field. However, “to capture a larger class of elliptic curves over the original field,” Bernstein and Lange generalized the definition of Edwards curves to the following:

Definition 2.13. For a fixed field K of characteristic not equal to two, choose $c, d \in K$ such that $cd(1 - dc^4) \neq 0$ (so $c \neq 0, d \neq 0$, and $dc^4 \neq 1$). The *Edwards elliptic curve* or *Edwards curve* defined by c and d is the (affine) curve of the form

$$x^2 + y^2 = c^2(1 + dx^2y^2) \tag{2.2}$$

This definition covers “more than 1/4 of all isomorphism classes of elliptic curves over a finite field,” so it is a more useful definition for our purposes. Moreover, they show that these are isomorphic to curves where $c = 1$, we will stay with the more general form given in 2.2. From now on we will use this definition when we talk of Edwards curves. In order to distinguish it from twisted Edwards curves (next section) and binary Edwards curves (next chapter), we'll denote the Edwards curve given by equation 2.2 by $E_{O,c,d}$.

Per theorem (2.1) in [7], $E_{O,c,d}$ is birationally equivalent to the Weierstrass curve

$$\left(\frac{1}{1 - dc^4}\right) v^2 = u^3 + 2\left(\frac{1 + dc^4}{1 - dc^4}\right) u^2 + u$$

via the birational map

$$(x, y) \mapsto (u, v) = \left(\frac{1 + y}{1 - y}, \frac{2(1 + y)}{x(1 - y)}\right)$$

(since there are only finitely many points with $x(1-y) = 0$, this is indeed a birational map), with inverse

$$(u,v) \mapsto (x,y) = \left(\frac{2u}{v}, \frac{u-1}{u+1}\right)$$

(again, there are only finitely many points such that $(u+1)v = 0$).

Like Edwards's original formulation, this curve has a simple, symmetric group law.

Theorem 2.5 (Bernstein & Lange Edwards Addition Law). For two points (x_1, y_1) and (x_2, y_2) on the Edwards curve $E_{O,c,d}$ given by equation 2.2, the map

$$(x_1, y_1), (x_2, y_2) \mapsto \left(\frac{x_1y_2 + y_1x_2}{c(1 + dx_1x_2y_1y_2)}, \frac{y_1y_2 - x_1x_2}{c(1 - dx_1x_2y_1y_2)}\right)$$

turns the set of rational points on $E_{O,c,d}(K)$ into an abelian group. The neutral element for this group law is $\mathcal{O} = (0, c)$, and the inverse of the point (x, y) is $(-x, y)$.

For a proof of this, see theorems (3.1) and (3.2) in [7].

Critics may wonder why cryptographic researchers are so interested in another normal form for elliptic curves. At first blush, this new normal form may even seem less useful than the familiar Weierstrass form since it requires a point of order four. As we'll see in the last section of this chapter, however, the benefits of Edwards curves far outweigh the drawbacks. For now, though, we'll briefly touch on one other type of Edwards curves.

2.4.3 Twisted Edwards Curves

For the sake of completeness, we now define twisted Edwards curves.[7]

In [6], Bernstein et. al. introduced a generalization of Edwards curves dubbed "twisted Edwards Curves." These curves "include more curves over finite fields,"

[7]Since they have been the main focus of research for things like pairings over Edwards curves; see Chapter 6.

including "every elliptic curve in Montgomery form" (another form garnering cryptographic interest). As [3] explains, their name "comes from the fact that the set of twisted Edwards curves is invariant under quadratic twists[8] while a quadratic twist of an Edwards curve is not necessarily an Edwards curve."

Definition 2.14. For a field K with $char(K) \neq 2$, and distinct nonzero elements $a, d \in K$, the *twisted Edwards curve* $E_{T,a,d}(K)$ is the curve

$$ax^2 + y^2 = 1 + dx^2y^2$$

As you can see, if $a = 1$, then $E_{T,a,d}$ is an Edwards curve with $c = 1$. Furthermore, $E_{T,a,d}$ is a quadratic twist of the Edwards curve $E_{O,1,d/a}$

$$\overline{x}^2 + \overline{y}^2 = 1 + (d/a)\overline{x}^2\overline{y}^2$$

via the map

$$(\overline{x}, \overline{y}) \mapsto (x, y) = (\overline{x}/\sqrt{a}, \overline{y})$$

over the field extension $K(\sqrt{a})$. Of course, if a is a square in K then these curves are isomorphic over K itself.

As before, this curve also has a symmetric and elegant group law.

Theorem 2.6 (Twisted Edwards Addition Law). Let $(x_1, y_1), (x_2, y_2)$ be two points on the twisted Edwards curve $E_{T,a,d}$ given by $ax^2 + y^2 = 1 + dx^2y^2$. Then the map

$$(x_1, y_1), (x_2, y_2) \mapsto \left(\frac{x_1y_2 + y_1x_2}{1 + dx_1x_2y_1y_2}, \frac{y_1y_2 - ax_1x_2}{1 - dx_1x_2y_1y_2} \right)$$

turns the set of rational points on $E_{T,a,d}(K)$ into an abelian group. The neutral

[8] A *quadratic twist* of a curve is another curve isomorphic to it over a field extension of degree two.

element for this curve is $(0, 1)$ and the inverse of (x, y) is $(-x, y)$.

For a proof, see [6].

In the next chapter, we'll define binary Edwards curves, a form of elliptic curve that is similar in flavor to the above ones except that it is defined over fields of characteristic two. First, though, we'll explain the cryptographic appeal of Edwards curves.

2.5 Cryptographic Safety from the Mathematical Foundation

As one can see from the operation counts given for the explicit formulas for addition and doubling on Edwards and twisted Edwards curves given in [7] and [6][9], these new curves outperform Weierstrass curves with regards to pure speed. For abelian groups that form the basis of cryptographic protocols, faster computations and more efficiency are certainly very important. However, binary Edwards curves produce a group law that in pure operation counts is a bit slower than its Weierstrass counterpart.[10] As it turns out, the Edwards family of curves is cryptographically interesting for a different reason: their groups laws are *unified* and *complete*, which leads to implementations that are safer against certain types of attacks from the very start; they have greater security "baked into them" from their mathematical foundation, as it were.

As we mentioned in a previous section, elliptic curve cryptography offers a lot of security for relatively low cost because of the lack of subexponential algorithms for calculating discrete logarithms. As such, attackers trying to break ECC implementa-

[9]Which we incorporate into `e2c2`; see Appendix 7.4.4.

[10]Slower only if we don't take validity checks into account; if we do, per [62], then the binary Edwards group law is still competitive.

tions tend to focus on the technical details of a specific implementation rather than any mathematical or algorithmic attacks which may take too long. Indeed, "the mathematically proved security of a cryptosystem does not imply its implementation [has] security against side-channel attacks," as [67] explains. A side-channel attack on a cryptosystem implementation is one that attempts to gain secret information via measuring some aspect of the implementation's performance that, perhaps unknown to its designers or users, leaks such information. Examples for ECC implementations include "those that monitor the power consumption and/or the electromagnetic emanations of a device," [67] expands, "and can infer important information about the instructions being executed or the operands being manipulated at a specific instant of interest." Elliptic curve cryptography with Weierstrass curves is certainly quite vulnerable to such attacks;[11] The group law as presented in theorem 2.1 has a number of special cases one must check for; any implementation needs to check whether either of the points it's trying to add is ∞, if the two points have the same x-coordinate but are different points, if they are the same but have an x-coordinate of zero (so they lie on the same vertical line), or if the two points are equal and have a nonzero x-coordinate.

There are a multitude of papers detailing such attacks against ECC or trying to safeguard systems against them; see [9, 12, 14, 16, 20, 34, 45, 44, 46, 47, 61], and [64] just to name a few. With all that energy expended on attacking elliptic curve cryptosystems from the implementation side, it would certainly be advantageous for a system to have a group law that protects against such attacks from the start; this

[11] At least in the "textbook" version we've presented, of course.

is where Edwards curves come in.

As [7] proves, the group law for an Edwards curve $E_{O,c,d}$ is *unified*, since it can also be used to double a point. That eliminates any need to check whether $P = Q$ when trying to add $P + Q$. Moreover, this same law works for the neutral element and for inverses; this eliminates even more special cases. Finally, if d isn't a square in K then the addition law is *complete*; i.e. it works for all pairs of inputs, and there are no special cases to check for at all. Twisted Edwards curves also share the same cryptographic benefits—the group law works for doubling, so it is unified, and is complete if a and d are both nonsquares in K (i.e. $\sqrt{a}, \sqrt{d} \notin K$). To reiterate, this strengthens ECC implementations based on these types of curves against side-channel analysis and attacks from the start; the elegance of their mathematical theory leads to safer, more easily implemented cryptography. As we'll see in the next chapter, binary Edwards curves also have these desirable properties.

CHAPTER 3: BINARY EDWARDS CURVES

In this chapter we discuss binary Edwards curves. We'll start with a discussion of the work of [8], the first paper to lay out an "Edwards-like" elliptic curve over a field of characteristic two. Then, we'll look at the practical improvements provided by [62].

Bernstein, Lange, and Farashahi's paper [8] presented "the first complete addition formulas for binary elliptic curves." As such, it was a huge milestone in the field; binary elliptic curves are very attractive from an implementation standpoint because, after all, computers work in binary. Until this paper came along, ECC implementations over a binary finite field were inherently vulnerable to the types of side-channel attacks mentioned in the previous chapter. Moloney, O'Mahony, and Laurent's paper [62] extended this work, presenting algorithms and practical measurements of things like code complexity that matter to implementors of cryptographic primitives.

3.1 Bernstein, Lange, & Farashahi

Unfortunately for cryptographers, the Edwards curve equation $x^2 + y^2 = c^2(1 + dx^2y^2)$ is not elliptic over fields with characteristic two; if it were, one could just use Edwards curves (or twisted Edwards curves) over these fields and reap the same benefits that we did over non-binary fields. In 2008, Bernstein, Lange, and Farashahi came up with a normal form for elliptic curves over binary fields that is reminiscent of Edwards curves which they dubbed *binary Edwards curves.*

Definition 3.1. Let K be a field with $char(K) = 2$, and $d_1, d_2 \in K$ such that $d_1 \neq 0$ and $d_2 \neq d_1^2 + d_1$. The *binary Edwards curve* E_{B,d_1,d_2} is the affine curve

$$d_1(x + y) + d_2(x^2 + y^2) = xy(x + 1)(y + 1)$$

As you can see from the definition, E_{B,d_1,d_2} is symmetric in x and y, so if (x, y) is a point on E_{B,d_1,d_2} then so is (y, x); this will soon yield our negation law. There are only two points on the curve that are invariant under this law: $(0, 0)$ and $(1, 1)$. As we'll see shortly, the former will be our neutral element, while the latter will be a point of order two.

In their theorem (2.2), the authors of [8] show that the affine form of E_{B,d_1,d_2} is nonsingular. Shortly after, they look at singularities of the projective closure

$$d_1(X + Y)Z^3 + d_2(X^2 + Y^2)Z^2 = XY(X + Z)(Y + Z)$$

of which there are two: $\Omega_1 = (1 : 0 : 0)$ and $\Omega_2 = (0 : 1 : 0)$. We'll expand on their work to show that the first of these blows up to two projective points, and use their same appeal to symmetry to cover the second.

To study E_{B,d_1,d_2} around Ω_1, consider the affine curve E_{Ω_1}:

$$d_1(1 + y)z^3 + d_2(1 + y^2)z^2 + y(1 + z)(y + z) = 0$$

If we take the partial derivatives of this curve with respect to y and z, we get

$$\begin{aligned}\frac{\partial E_{\Omega_1}}{\partial y} &= d_1z^3 + 2d_2yz^2 + (z+1)(y+z) + (z+1)y \\ &= d_1z^3 + 2d_2yz^2 + 2yz + z^2 + 2y + z \\ &= d_1z^3 + z^2 + z\end{aligned}$$

and

$$\begin{aligned}\frac{\partial E_{\Omega_1}}{\partial z} &= 3(y+1)d_1z^2 + 2(y^2+1)d_2z + (z+1)y + (y+z)y \\ &= 3d_1yz^2 + 2d_2y^2z + 3d_1z^2 + 2d_2z + y^2 + 2yz + y \\ &= d_1(1+y)z^2 + y^2 + y\end{aligned}$$

Evaluating these at the point $(y,z) = (0,0)$, we see that E_{Ω_1} is indeed singular; we can "blow up" this singularity by substituting $y = tz$ into E_{Ω_1} and dividing through by z^2, getting the following curve E_t:

$$d_1(1+tz)z + d_2(1+t^2z^2) + t(1+t)(1+z) = 0$$

If we substitute in $z = 0$, E_t becomes $t^2 + t + d_2 = 0$ which has two distinct roots in $\overline{K}$. To see that these two points are nonsingular, consider the partial derivatives

$$\begin{aligned}\frac{\partial E_t}{\partial t} &= 2d_2tz^2 + d_1z^2 + (z+1)(t+1) + (z+1)t \\ &= 2d_2tz^2 + d_1z^2 + 2tz + 2t + z + 1 \\ &= d_1z^2 + z + 1\end{aligned}$$

and

$$\begin{aligned}\frac{\partial E_t}{\partial z} &= 2d_2t^2z + d_1tz + (t+1)t + (tz+1)d_1 \\ &= 2d_2t^2z + 2d_1tz + t^2 + d_1 + t \\ &= t^2 + d_1 + t\end{aligned}$$

Neither of these partial derivatives vanish at the point $(z,t) = (0,0)$, so these blowups are nonsingular. As [8] says, they are "defined over the smallest extension of K in which $d_2 + t + t^2 = 0$ has roots."

The authors provide the following birational equivalence: the map

$$\begin{aligned}&(x,y) \mapsto (u,v) \\ &\quad = \left(\frac{d_1(d_1^2+d_1+d_2)(x+y)}{xy+d_1(x+y)}, d_1(d_1^2+d_1+d_2)\left[\frac{x}{xy+d_1(x+y)} + d_1 + 1\right]\right)\end{aligned}$$

is a birational equivalence[12] between E_{B,d_1,d_2} and the binary elliptic curve W[13]

$$v^2 + uv = u^3 + (d_1^2 + d_2)u^2 + d_1^4(d_1^4 + d_1^2 + d_2^2)$$

This map has inverse

$$(u,v) \mapsto (x,y) = \left(\frac{d_1(u+d_1^2+d_1+d_2)}{u+v+(d_1^2+d_1)(d_1^2+d_1+d_2)}, \frac{d_1(u+d_1^2+d_1+d_2)}{v+(d_1^2+d_1)(d_1^2+d_1+d_2)}\right)$$

This map is undefined at the point $(0,0)$; if we define $(0,0) \mapsto \infty$, then this becomes an isomorphism between the curves.

The addition law on a binary Edwards curve is just as symmetric as its ordinary and twisted counterparts, if a little more complicated:

[12]Though we'll use the equivalence given in [62] ourselves.

[13]In shorter Weierstrass form for binary curves

Theorem 3.1 (Binary Edwards Addition Law). If (x_1, y_1) and (x_2, y_2) are two points on the binary Edwards curve E_{B,d_1,d_2}, then the mapping $(x_1, y_1), (x_2, y_2) \mapsto (x_3, y_3)$ turns the rational points on this curve into an abelian group, where

$$x_3 = \frac{d_1(x_1 + x_2) + d_2(x_1 + y_1)(x_2 + y_2) + (x_1 + x_1^2)(x_2(y_1 + y_2 + 1) + y_1 y_2)}{d_1 + (x_1 + x_1^2)(x_2 + y_2)}$$
$$y_3 = \frac{d_1(y_1 + y_2) + d_2(x_1 + y_1)(x_2 + y_2) + (y_1 + y_1^2)(y_2(x_1 + x_2 + 1) + x_1 x_2)}{d_1 + (y_1 + y_1^2)(x_2 + y_2)}$$

as long as the denominators in the above fractions are nonzero.

Substituting $(0, 0)$ for either (x_1, y_1) or (x_2, y_2) in the above law, we see that $(0, 0)$ is the neutral element. Moreover, $(x, y) + (y, x) = (0, 0)$, so the inverse of a point (x, y) is (y, x) as we said before. When defined, this addition law is unified; it can be used for doubling as well. For a proof of this law, see section 3 of [8]; in that section, the authors demonstrate that this addition law corresponds to the addition law on the equivalent Weierstrass curve, so the birational map is indeed a isomorphism.

Astute readers will notice the caveat "as long as the denominators in the above fractions are nonzero" in the previous theorem. We could try and list all the cases where those fractions don't exist and piece together a group law that takes these special cases into account, like the Weierstrass group law does. However, Bernstein, Lange, and Farashahi offer us another very helpful theorem.

Theorem 3.2 (Complete Binary Edwards Curves). Let K be a field with $char(K) = 2$ and $d_1, d_2 \in K$ such that $d_1 \neq 0$ and no element $t \in K$ satisfies $t^2+t+d_2 = 0$. Then the addition law on the binary Edwards curve $E_{B,d_1,d_2}(K)$ is complete. Moreover, every ordinary elliptic curve over the finite field $\mathbb{F}_{2^n}$ for $n \geq 3$ is birationally equivalent over

$\mathbb{F}_{2^n}$ to a complete binary Edwards curve.

See theorems (4.1) and (4.3) for proofs of these claims. Since elliptic curve cryptography typically involves finite binary fields of degree n at least 160, the above theorem tells us that we can use a binary Edwards curve and reap the benefits of a complete and unified group law.

Despite their extremely thorough treatment in [8], Bernstein, Lange, and Farashahi did leave some small room for improvement. In trying to find an equivalent complete binary Edwards curve for a given Weierstrass curve, they left some nondeterminism in finding d_1. Though they could find an appropriate d_1 easily enough experimentally, they didn't have a deterministic algorithm for it.

3.2 Moloney, O'Mahony, & Laurent

In 2010, Moloney, O'Mahony, and Laurent posted [62] online. In it, they perform a practical, implementation-focused analysis of binary Edwards curves, and come up with some very useful results.

First, they offer a modified birational equivalence. Recall the usual trace function

$$Tr : \mathbb{F}_{2^n} \to \mathbb{F}_2, \qquad \alpha \mapsto \sum_{i=0}^{n-1} \alpha^{2^i}$$

and define the half-trace function

$$H : \mathbb{F}_{2^n} \to \mathbb{F}_2, \qquad \alpha \mapsto \sum_{i=0}^{(n-1)/2} \alpha^{2^{2^i}}$$

(noting that n must be odd). If given a_2 and a_6 for a Weierstrass curve, suppose we found a suitable d_1; we can then calculate

$$d_2 = d_1^2 + d_1 + \sqrt{a_6}/d_1^2$$

We will also make use of b which satisfies $b^2 + b = d_1^2 + d_2 + a_2$; it can be directly calculated as

$$b = H(d_1^2 + d_2 + a_2)$$

The authors show that $(u, v) \mapsto (x, y)$ is another birational equivalence from the Weierstrass curve to E_{B,d_1,d_2}, where

$$x = \frac{d_1(bu + v + (d_1^2 + d_1)(d_1^2 + d_1 + d_2))}{u^2 + d_1u + d_1^2(d_1^2 + d_1 + d_2)}$$
$$y = \frac{d_1((b+1)u + v + (d_1^2 + d_1)(d_1^2 + d_1 + d_2))}{u^2 + d_1u + d_1^2(d_1^2 + d_1 + d_2)}$$

Though there is no difference between this equivalence and the one presented by Bernstein et.al., the calculation of this equivalence involves fewer field inversions. Field inversions tend to be very costly to calculate, so the fewer the better.[14]

Secondly, and perhaps more importantly, the authors present two deterministic algorithms to find a suitable d_1 given $n \geq 3$, a_2, and a_6 determining a Weierstrass curve over the finite field $\mathbb{F}_{2^n}$. We reproduce the first algorithm here, since that's what is used in our software library `e2c2`. Precompute $t = Tr(a_2)$, $r = Tr(a_6)$, and $w = x + Tr(x)$ where x is the indeterminant used to define our field extension $\mathbb{F}_{2^n}$. This algorithm "terminates with guaranteed success in a finite number of steps, except in the case $t = r = 0$." Fortunately, "this case does not appear in any of the standards (e.g. NIST) of which the authors are aware."

[14] `e2c2` uses this birational equivalence.

Finally, [62] offers some valuable measurements and comparisons between implementations of Weierstrass and binary Edwards curves. They note that “implementing ECC from the textbooks leaves us with incredibly complex code,” while implementations of binary Edwards curves have lower complexity. The symmetric, unified, and complete group law takes a lot of the burden off of potential developers and implementors. More interestingly, despite the larger operation count for the binary Edwards addition law, the fact Weierstrass implementations must constantly check for special cases slows them down considerably. The cost measurements commonly mentioned in the literature “do not take into account the cost of checking” if an operation “is attempting to double the point at infinity,” for example. Moreover, “performance is significantly different if implemented on a different processor.” Integrating binary Edwards code into an existing ECC library, they found on one processor that, as may be expected, the binary Edwards curve code was slower. However, on a different processor that pipelined instructions, the implementation could take advantage of the fact that the binary Edwards curve addition law involves no conditionals; “due to the fact that we do not have to break the pipeline with checks for the point of infinity,” along with some other, more esoteric technical work on the part of the authors, “we are able to increase the performance of [binary Edwards curves] such that it is approximately 25% faster than the equivalent Weierstrass version.”

```
function MOLalg1($n, p, t, r, a_6, w$)
    if $t = 0$ and $r = 1$ then
        $d_1 \leftarrow 1$
    else
        if $t = 1$ and $r = 0$ then
            $d_1 \leftarrow \sqrt[4]{a_6}$
        else
            if $t = r = 1$ and $a_6 \neq 1$ then
                if $Tr({}^1\!/_{a_6+1}) = 1$ then
                    $d_1 \leftarrow \sqrt{a_6} + \sqrt[4]{a_6}$
                else
                    $d_1 \leftarrow \sqrt[4]{a_6} + 1$
                end if
            else
                if $t = 1$ and $a_6 = 1$ then
                    if $Tr({}^1\!/_w) = 1$ then
                        $d_1 \leftarrow w$
                    else
                        if $Tr({}^1\!/_{(w+1)}) = 0$ then
                            $d_1 \leftarrow {}^1\!/_{(w+1)}$
                        else
                            $d_1 \leftarrow 1 = {}^1\!/_{(w+1)}$
                        end if
                    end if
                else
                    if $t = r = 0$ then
                        if $Tr({}^1\!/_{(a_6+1)}) = 0$ then
                            $d_1 \leftarrow \sqrt[4]{a_6} + 1$
                        else
                            $i \leftarrow 1$
                            $s \leftarrow \sqrt{a_6}$
                            while $Tr(a_6^{2^i+1}) = 0$ do
                                $s \leftarrow s^2$
                                $i \leftarrow i + 1$
                            end while
                            $d_1 \leftarrow {}^1\!/_{(s+1)}$
                        end if
                    end if
                end if
            end if
        end if
    end if
    return $d_1$
end function
```

Algorithm 3.1: Moloney, O'Mahony, & Laurent's first d_1 finder

CHAPTER 4: PRACTICAL CONSIDERATIONS

Binary Edwards curves specifically, and Edwards curves in general, have generated a lot of excitement in the cryptographic field. As such, a number variations, adaptations, and entirely new normal forms have been proposed in recent years. Many of them are promising and have interesting mathematical properties; that doesn't mean, unfortunately, that they are "ready for primetime" as far as cryptographic implementation is concerned. In this chapter, we show that four new normal forms for elliptic curves, despite being mathematically interesting and involving some quite nice theory, do not measure up to the cryptographic standard set by binary Edwards curves.[15] As we shall see, these constructions exhibit weaknesses that fall into one of two categories: either their group law is not symmetric, so commutativity is hard to see (though of course still present), or their atypical choice of neutral point obfuscates the result of adding a point and the neutral element. In both cases, one has to resort to working modulo the curve equation (or more precisely, modulo the ideal generated by the curve equation in the appropriate polynomial ring) to see that these computations behave as expected. This means that elementary operations, the results of which should be immediately apparent, cannot be implemented programmatically in a simple way; even simple work must involve unnecessary checks and reductions. This

[15]A previous version of this chapter has been posted to the International Association for Cryptologic Research's cryptology eprint archive (`http://eprint.iacr.org/2013/015`) and has been submitted to IACR's CRYPTO 2013 conference (`http://www.iacr.org/conferences/crypto2013/`).

extra work will at best slow down a cryptosystem, and at worst could leak enough side-channel information to severely weaken the system.

4.1 Two Weaknesses & How Edwards Curves Avoid Them

Recall that Edwards curves, originally presented by Edwards in [25] and expanded upon by Bernstein and Lange in [7], are elliptic curves over a field of characteristic not equal to two of the form

$$x^2 + y^2 = c^2(1 + dx^2y^2)$$

with some restrictions on c and d and have the affine group law

$$(x_1, y_1) + (x_2, y_2) = \left(\frac{x_1y_2 + y_1x_2}{c(1 + dx_1x_2y_1y_2)}, \frac{y_1y_2 - x_1x_2}{c(1 - dx_1x_2y_1y_2)}\right) \tag{4.3}$$

Next, twisted Edwards curves can be taken over any non-binary field, have the form

$$ax^2 + y^2 = 1 + dx^2y^2$$

and have affine group law

$$(x_1, y_1) + (x_2, y_2) = \left(\frac{x_1y_2 + y_1x_2}{1 + dx_1x_2y_1y_2}, \frac{y_1y_2 - ax_1x_2}{1 - dx_1x_2y_1y_2}\right) \tag{4.4}$$

Finally, binary Edwards curves take the form

$$d_1(x + y) + d_2(x^2 + y^2) = (x + x^2)(y + y^2)$$

over a field of characteristic two, and have the (slightly more complicated but still symmetric) group law $(x_1, y_1) + (x_2, y_2) = (x_3, y_3)$ where

$$\begin{aligned} x_3 &= \frac{d_1(x_1 + x_2) + d_2(x_1 + y_1)(x_2 + y_2) + (x_1 + x_1^2)(x_2(y_1 + y_2 + 1) + y_1y_2)}{d_1 + (x_1 + x_1^2)(x_2 + y_2)} \\ y_3 &= \frac{d_1(y_1 + y_2) + d_2(x_1 + y_1)(x_2 + y_2) + (y_1 + y_1^2)(y_2(x_1 + x_2 + 1) + x_1x_2)}{d_1 + (y_1 + y_1^2)(x_2 + y_2)} \end{aligned} \tag{4.5}$$

All of four of the normal forms we examine have group laws that are purported to be unified and complete (at least on a specified subgroup). They fail to live up to the Edwards standard in other ways, however. A few of these normal forms have group laws that are asymmetric; that is, the equations for adding two points P and Q involve their coordinates in such a fashion that it's not obvious that $P + Q$ is the same as $Q + P$, even though addition of two rational points on an elliptic curve is commutative. None of the three major Edwards curve types—the original one put forward in [25] and [7], binary curves presented in [8], or twisted curves from [6]—exhibit this flaw. All three of the Edwards group laws—Edwards curves in equation 4.3, twisted Edwards curves in equation 4.4, and binary Edwards curves in 4.5—are symmetric with respect to their inputs; one can clearly see that $(x_1, y_1)+(x_2, y_2)$ is the same as $(x_2, y_2)+(x_1, y_1)$ without any extra work simply because of the commutativity of field addition and multiplication. This means that any implementation of these laws in computer code will be much less complex than they otherwise could be if extra work were needed to demonstrate this simple fact.

The other weakness exhibited by some of the normal forms we examine is their atypical choice of neutral element. For some, the neutral element choice makes it unclear that $\mathcal{O} + P = P + \mathcal{O} = P$. For Edwards curves, the neutral element is $(0, 1)$. It's clear that this can be substituted into the Edwards group law in either position and the result will always be the other point; that is, it's immediately clear that $(0, 1)$ is indeed the neutral element for this law. Similarly, twisted Edwards curves have neutral point $(0, 1)$, while binary Edwards curves have neutral point $(0, 0)$. Substituting these into either position in their group laws clearly demonstrates that they are the correct neutral elements. For some of the variations, it is not so apparent that the stated neutral element is correct; we again need to resort to reducing modulo the ideal generated by the curve equation in order to see that this is the case.

4.2 Farashahi & Joye

The first curve we'll consider is Farashahi and Joye's Generalized Hessian curve presented in [26]. This curve has the form

$$H_{c,d} : x^3 + y^3 + c = dxy$$

or, in projective coordinates,

$$\mathbf{H}_{c,d} : X^3 + Y^3 + cZ^3 = dXYZ$$

over an arbitrary field. The group of rational points on this curve has neutral element $\mathcal{O} = (1 : -1 : 0)$.

The authors present some unified addition formulas for $\mathbf{H}_{c,d}$ (equations (9) and (10) in [26]). If we let $P = (X_1 : Y_1 : Z_1)$ and $Q = (X_2 : Y_2 : Z_2)$ be two points on

$\mathbf{H}_{c,d}$, then according to their first equation we have $P + Q = (X_3 : Y_3 : Z_3)$ where

$$X_3 = cY_2Z_1^2Z_2 - X_1X_2^2Y_1$$

$$Y_3 = X_2Y_1^2Y_2 - cX_1Z_1Z_2^2$$

$$Z_3 = X_1^2X_2Z_2 - Y_1Y_2^2Z_1$$

Using these formulas, we can calculate $\mathcal{O} + P = (X_1^2 : X_1Y_1 : X_1Z_1)$; while at first this may not seem to be the same as P, projective points are really equivalence classes, so this is of course the same point as we would get dividing all three positions by X_1,[16] viz. $(X_1 : Y_1 : Z_1) = P$ provided, of course, that $X_1 \neq 0$. Similarly, $P + \mathcal{O} = (-X_1Y_1 : -Y_1^2 : -Y_1Z_1) \equiv (X_1 : Y_1 : Z_1) = P$.

The real trouble with this construction, however, comes from comparing $P + Q$ with $Q + P$. Let $(X_4 : Y_4 : Z_4) = Q + P$, so

$$X_4 = cY_1Z_1Z_2^2 - X_1^2X_2Y_2$$

$$Y_4 = X_1Y_1Y_2^2 - cX_2Z_1^2Z_2$$

$$Z_4 = X_1X_2^2Z_1 - Y_1^2Y_2Z_2$$

Since we need point addition to be commutative, this should be equal (or at least equivalent in the projective point sense) to $P + Q$. Suppose that all of $P, Q, P + Q$, and $Q + P$ are finite points, so their Z coordinate is nonzero. Then we need the following:

$$\frac{X_3}{Z_3} = \frac{cY_2Z_1^2Z_2 - X_1X_2^2Y_1}{X_1^2X_2Z_2 - Y_1Y_2^2Z_1} = \frac{cY_1Z_1Z_2^2 - X_1^2X_2Y_2}{X_1X_2^2Z_1 - Y_1^2Y_2Z_2} = \frac{X_4}{Z_4}$$

[16] X_1 cannot be zero, or else $\mathcal{O} + P$ would be a singular point on $\mathbf{H}_{c,d}$, something which the authors show is impossible.

This is true if and only if $X_3Z_4 - X_4Z_3 = 0$; i.e. if and only if the following is zero:

$$-X_1X_2\left(cX_1Y_1Z_1Z_2^3 - cX_2Y_2Z_2Z_1^3 - X_1^3X_2Y_2Z_2 + X_1Y_1Z_1X_2^3 + X_1Y_1Z_1Y_2^3 - X_2Y_2Z_2^3\right) \tag{4.6}$$

Suppose furthermore that $X_1X_2 \neq 0$; then we need the larger factor to be zero, which isn't immediately apparent. Factoring and simplifying, this larger factor becomes

$$(X_1Y_1Z_1)(X_2^3 + Y_2^3 + cZ_2^3) - (X_2Y_2Z_2)(X_1^3 + Y_2^3 + cZ_1^3)$$

Working modulo the curve equation, we know $X^3 + Y^3 + cZ^3 = dXYZ$, which implies our work simplifies to

$$(X_1Y_1Z_1)(dX_2Y_2Z_2) - (X_2Y_2Z_2)(dX_1Y_1Z_1)$$

which is, at last, zero.

Similarly,

$$Y_3Z_4 - Y_4Z_3 =$$

$$(X_1^2X_2Z_3 - Y_1Y_2^2Z_1)(cX_2Z_1^2Z_2 - X_1Y_1Y_2^2) - (X_1X_2^2Z_1 - Y_1^2Y_2Z_2)(cX_1Z_1Z_2^2 - X_2Y_1^2Y_2) =$$

$$Y_1Y_2(cX_1Y_1Z_1Z_2^3 - cX_2Y_2Z_1^3Z_2 + X_1X_2^3Y_1Z_1 - X_1^3X_2Y_2Z_2 + X_1Y_1Y_2^3Z_1 - X_2Y_1^3Y_2Z_2) =$$

$$Y_1Y_2\left[(X_1Y_1Z_1)(X_2^3 + Y_2^3 + cZ_2^3) - (X_2Y_2Z_2)(X_1^3 + Y_1^3 + cZ_1^3)\right]$$

If $Y_1Y_2 \neq 0$, then this can only be zero if we resort to the curve equation, getting

$$Y_1Y_2\left[(X_1Y_1Z_1)(dX_2Y_2Z_2) - (X_2Y_2Z_2)(dX_1Y_1Z_1)\right]$$

Thus $P + Q$ does indeed equal $Q + P$; note, however, that in order to reach this conclusion we had to perform substitutions using $\mathbf{H}_{c,d}$'s equation. This equality was not apparent from the outset but rather required working modulo the ideal generated by the curve equation in the appropriate polynomial ring. This addition is true, and even mathematically pleasing, but not cryptographically viable. Such reductions would complicate any computer code implementation of this group—at best leading to slow execution speed, and at worst causing side-channel leaks that could potentially lead to a break of the implementation. This elliptic curve is not as safe as Edwards curves when it comes to the concerns of cryptographic implementation.

4.3 Wang, Tang, & Yang

In [78], the authors explore the curve

$$M_d : x^2y + xy^2 + dxy + 1 = 0$$

and its homogeneous projective version

$$\widetilde{M_d} : X^2Y + XY^2 + dXYZ + Z^3 = 0$$

over a field of characteristic greater than three.[17] The neutral element of the group of rational points on this curve is $(1 : -1 : 0)$ Though their affine group law seems to have little trouble in the symmetry department, the projective group law is where the real trouble lies. Per the law given in [78], the sum of two points $(X_1 : Y_1 : Z_1)$

[17]Of course characteristic greater than three means that this curve is not a direct competitor to binary Edwards curves as such. However, it attempts to have a unified group law like Edwards curves do and fails for reasons similar to the other normal forms we analyze; these reasons make it worth including in our discussion.

and $(X_2 : Y_2 : Z_2)$ is $(X_3 : Y_3 : Z_3)$ where

$$X_3 = X_1X_2(Y_1Z_2 - Y_2Z_1)^2$$

$$Y_3 = Y_1Y_2(X_1Z_2 - X_2Z_1)^2$$

$$Z_3 = (X_1Z_2 - X_2Z_1)(Y_1Z_2 - Y_2Z_1)(X_2Y_2Z_1^2 - X_1Y_1Z_2^2)$$

is problematic with regards to the neutral element. Suppose we wished to add the point $P = (X : Y : Z)$ (a finite point, so $Z \neq 0$) and the neutral element $(1 : -1 : 0)$; then we'd have

$$X_3 = X \cdot 1(Y \cdot 0 - (-1) \cdot Z)^2$$

$$Y_3 = Y \cdot (-1)(X \cdot 0 - 1 \cdot Z)^2$$

$$Z_3 = (X \cdot 0 - 1 \cdot Z)(Y \cdot 0 - (-1) \cdot Z)(1 \cdot (-1) \cdot Z^2 - X \cdot Y \cdot 0^2)$$

which simplifies to $(XZ^2 : -YZ^2 : Z^4) \equiv (X : -Y : Z^2)$. Except in very special circumstances, this is of course not equal to $(X : Y : Z)$; moreover, it's not apparent how resorting to the curve equation will even help here.

There are even more problems here, though. For example, $\mathcal{O} + P = (XZ^2 : -YZ^2 : -Z^4) \equiv (X : -Y : -Z^2)$, $\mathcal{O} + \mathcal{O} = (0 : 0 : 0)$, and $P + P = (0 : 0 : 0)$, so this law is not unified (contrary to the claims of [78]). These problems can be seen by running the following Sage [74] script:

Listing 4.2: Arithmetic on Wang et.al.'s curve

```
var('d x y z')
R.<d, x, y, z> = GF(17^17, 'a')[]
S = R.quotient([x^2 * y + x * y^2 + d * x * y * z + z^3])
```

```

def add((x1, y1, z1), (x2, y2, z2)):
    x3 = S(x1 * x2 * (y1 * z2 - y2 * z1)^2)
    y3 = S(y1 * y2 * (x1 * z2 - x2 * z1)^2)
    z3 = S((x1 * z2 - x2 * z1) *
           (y1 * z2 - y2 * z1) *
           (x2 * y2 * z1^2 - x1 * y1 * z2^2))
    return (x3, y3, z3)

o, p = (1, -1, 0), (x, y, z)

for pair in cartesian_product_iterator([(o, p)] * 2):
    print "add{0}\t=\t{1}".format(pair, add(*pair))
```

Hence this curve is not a suitable candidate for cryptographic implementation.

4.4 Wu, Tang, & Feng

In [80], presented at INDOCRYPT 2012, Wu, Tang, & Feng introduce the curve

$$S_t : x^2y + xy^2 + txy + x + y = 0$$

and its projective version

$$X^2Y + XY^2 + tXYZ + XZ^2 + YZ^2 = 0$$

and study its properties over a binary field. In their paper, they define the projective point $\mathcal{O} = (1 : 1 : 0)$ as the neutral element.

Suppose that we wish to add the finite projective point $(X : Y : 1)$ to $\mathcal{O}$ using the formulas given in [80] to obtain the point $(X_3 : Y_3 : Z_3)$; moreover, suppose that $X \neq Y$ and both are nonzero. Then

$$\begin{aligned}
X_3 &= (Y \cdot 1 + 1 \cdot 0)\,[(X \cdot 1 + Y \cdot 1)(Y \cdot 0 + 1 \cdot 1) + t \cdot Y \cdot 1 \cdot (1 \cdot 0 + X \cdot 1)] \\
&= X\,[(X + Y) + tXY] \\
&= X(X + Y + tXY) \\
Y_3 &= (Y \cdot 1 + 1 \cdot 0)\,[(X \cdot 1 + Y \cdot 1)(X \cdot 0 + 1 \cdot 1) + t \cdot X \cdot 1(1 \cdot 0 + Y \cdot 1)] \\
&= Y\,[(X + Y) + tXY] \\
&= Y(X + Y + tXY) \\
Z_3 &= (X \cdot 1 + Y \cdot 1)(X \cdot 1 + 1 \cdot 0)(Y \cdot 1 + 1 \cdot 0) \\
&= XY(X + Y)
\end{aligned}$$

Therefore $(X_3 : Y_3 : Z_3)$ is equivalent to

$$\left(\frac{X(X + Y + tXY)}{XY(X + Y)} : \frac{Y(X + Y + tXY)}{XY(X + Y)} : 1\right) = \\ \left(\frac{X + Y + tXY}{Y(X + Y)} : \frac{X + Y + tXY}{X(X + Y)} : 1\right)$$

From the curve equation, we know that $X+Y+tXY = X^2Y+XY^2 = XY(X+Y)$, so $(X_3 : Y_3 : Z_3)$ is indeed equal to $(X : Y : 1)$. Note, however, that this result only occurs if we take into account the curve equation. For something as simple as adding a point to the neutral element, having to modulo the curve equation to show that $(X : Y : 1) + \mathcal{O} = (X : Y : 1)$ is unnecessarily complicated.

4.5 Diao & Fouotsa

In [23], presented at "Journées C2: Codage et Cryptographie" in September 2012, Diao & Fouotsa introduce the curve

$$\mathcal{E}_\lambda : 1 + x^2 + y^2 + x^2y^2 = \lambda xy$$

which is valid over a field of any characteristic. Their paper is very detailed, and the construction involves some interesting work with Theta functions. Unfortunately, this construction also falls short of the cryptographic applicability of Edwards curves due to the asymmetry of the group law they present.

Suppose we wished to add two points (x_1, y_1) and (x_2, y_2); it shouldn't matter in which order we add them, because the group law should be commutative. By the work in [23], we have

$$(x_1, y_1) + (x_2, y_2) = \left(\frac{x_1 + y_1x_2y_2}{y_2 + x_1y_1x_2}, \frac{x_1x_2 + y_1y_2}{1 + x_1x_2y_1y_2} \right)$$

while

$$(x_2, y_2) + (x_1, y_1) = \left(\frac{x_2 + x_1y_1y_2}{y_1 + x_1x_2y_2}, \frac{x_1x_2 + y_1y_2}{1 + x_1x_2y_1y_2} \right)$$

The second coordinates of these points are obviously equal to each other, but we also need the first ones to be equal. This is the case if and only if

$$\frac{x_1 + y_1x_2y_2}{y_2 + x_1y_1x_2} = \frac{x_2 + x_1y_1y_2}{y_1 + x_1x_2y_2} \iff$$

$$(x_1 + y_1x_2y_2)(y_1 + x_1x_2y_2) = (x_2 + x_1y_1y_2)(y_2 + x_1x_2y_1) \iff$$

$$x_1y_1 + x_2(x_1^2 + y_1^2 + x_1y_1x_2y_2) = x_2y_2 + x_1y_1(x_2^2 + y_2^2 + x_1x_2y_1y_2)$$

Using the curve equation this is true if and only if

$$x_1y_1 + x_2y_2(1 + x_1^2y_1^2 + x_1x_2y_1y_2) = x_2y_2 + x_1y_1(1 + x_2^2y_2^2 + x_1x_2y_1y_2) \iff$$

$$x_1y_1 + x_2y_2 + x_1^2x_2y_1^2y_2 + x_1x_2^2y_1y_2^2 = x_2y_2 + x_1y_1 + x_1x_2^2y_1y_2^2 + x_1^2x_2y_1^2y_2$$

So it is true that $(x_1, y_1) + (x_2, y_2) = (x_2, y_2) + (x_1, y_1)$ as we required. Note that proving this simple fact again required resorting to working modulo the curve equation (i.e. modulo the ideal generated by the curve equation in the polynomial ring $\mathbb{F}_2^n[x_1, x_2, y_1, y_2]$).

4.6 Conclusions

Following the excitement regarding the various types of Edwards curves, normal forms for elliptic curves have been presented and explored with an eye to improving upon one characteristic or another of Edwards curves while maintaining the same safety and security afforded by their complete and unified group laws. It turns out that there is more to being as safe as Edwards curves than just being complete (on a subgroup or over the whole group) and unified, however. As we have demonstrated, four recently proposed normal forms exhibit weaknesses that don't show up in Edwards curves: either their group laws are not symmetric or they use an unusual choice of neutral element.[18] Both of these weaknesses mean that we must reduce modulo their curve equations to demonstrate even elementary facts, like $\mathcal{O} + P = P$ or $P + Q = Q + P$. This extra work will complicate any computer implementation, leading to slower execution speed and perhaps leakage of information through side channels. The main advantage Edwards curves have for implementation is their in-

[18]In fact, one normal form's troubles extend even deeper.

corporating safety and security from the ground up; these newer normal forms do not measure up when it comes to suitability for cryptographic implementation.

CHAPTER 5: PAIRINGS

One area of cryptography that we have yet to touch on is *pairing based cryptography.* Pairings are bilinear forms over specific points on an elliptic curve (more on that in a moment), and were actually first used in cryptography to attack cryptosystems rather than implement them—see Chapter 6 for more details. In this chapter we'll discuss the mathematics of pairings, beginning with the necessary background information. From there, we'll discuss one way to compute an important function, dubbed a *Miller function*, over a binary Edwards curve. Finally, we'll discuss some interesting directions for future work, including a preliminary result that may help pave the way.

5.1 Background

5.1.1 Preliminaries

To start with we will discuss bilinear maps in a somewhat general setting, though of course we will eventually focus on those taking as input rational points on an elliptic curve over a finite field. Let G_1 be a cyclic group written additively and G_2 be a cyclic group written multiplicatively (with identity element 1) such that both have the same prime order n. A *bilinear map* or *pairing* is a function $\widehat{e} : G_1 \times G_1 \to G_2$ that satisfies the following properties:

1. *Bilinearity.* For any $P, Q \in G_1$ and $\alpha, \beta \in \mathbb{Z}_n^*$, we have $\widehat{e}(\alpha P, \beta Q) = \widehat{e}(P, Q)^{\alpha\beta}$

2. *Non-degeneracy.* There exists $P, Q \in G_1$ such that $\widehat{e}(P, Q) \neq 1$; ergo if $\langle P \rangle = G_1$ then $\langle \widehat{e}(P, P) \rangle = G_2$.

3. *Efficient Computability.* For all $P, Q \in G_1$, the pairing $\widehat{e}(P, Q)$ can be computed efficiently (say, in polynomial time).

In the elliptic curve settings, two popular bilinear maps are the Weil pairing and the Tate pairing; see [18, 71, 79] for details. The Weil pairing was used in Boneh & Franklin's scheme in [10] that gave a solution to the problem originally posed by Shamir in [69]. We'll take a cue from the literature and focus on the Tate pairing[19] in what follows.

5.1.2 The Tate Pairing

Suppose E is some elliptic curve over a finite field $\mathbb{F}_q$ with identity $\mathcal{O}$.[20] To compute the pairing of two points P and Q on E, we'll need to first understand the notion of a *divisor*. We off some definitions; for a more in depth coverage, see [36, 71].

Definition 5.1. The *divisor group* of E, denoted by $Div(E)$, is the free abelian group generated by the points of E.[71] An element of $Div(E)$ is a formal sum $D = \sum_{P \in E} n_P(P)$ where (P) is the so called "place" associated with the point P. The *degree* of a divisor is the sum $deg(D) = \sum_{P \in E} n_P$, and the divisors of degree zero form a subgroup $Div^0(E)$.

There are a special set of divisors that correspond to rational functions over E, which we'll now discuss. To get there, we'll borrow a few definitions from [36]. Recall

[19]Also known as the "Tate-Lichtenbaum pairing" in [71] or the "Reduced Tate pairing" in [3].

[20]We won't focus on exactly what form the elliptic curve is in, currently, but of course we are focusing on binary Edwards curves.

that E is an algebraic variety over our field $\mathbb{F}_q$.[21]

Definition 5.2. Let $E \subseteq \overline{K}^n$ be an algebraic variety. A *polynomial function* of E is a mapping of E into $\overline{K}$ induced by a polynomial in $K[x_1, \ldots, x_n]$. The *coordinate ring* of E is the ring of all such mappings. We define the *function field* of $K(E)$ to be the field induced by ratios of polynomials from the coordinate ring.

Next, we define the divisor of a function.

Definition 5.3. For a rational function f from a given function field $K(E)$ of an algebraic variety E over a field K, factor f completely over $\overline{K}$:

$$f(x) = \alpha \prod (x - P)^{e_P}$$

for some $\alpha \in \overline{K}$, expressing f as the ratio of powers of *zeroes* and *poles*; here $e_P \in \mathbb{Z}$. Given a point $P \in E$, write $f(x)$ as $(x - P)^{e_P} g(x)$, where P is neither a zero nor a pole of g (so $(x - P)$ divides neither the numerator nor the denominator of g). The *order* of f at P, denoted by $ord_P(f)$, is the exponent e_P. The *divisor* of f is the element of $Div(E)$ given by

$$div(f) = \sum_{P \in E} ord_P(f)(P)$$

A divisor is *principal* if it is the divisor of a rational function.

We can now define the Tate pairing; we'll borrow [3]'s definition.

Definition 5.4 (The Tate Pairing). Let

- $E(\mathbb{F}_q)$ be an elliptic curve over $\mathbb{F}_q$ with neutral element $\mathcal{O}$;

[21]In the language of [36], it's an *algebraic set*, but the difference isn't important for our purposes.

- $n|\#E$ be a prime divisor of the group order and $k > 1$ be the *embedding degree* of E with respect to n, i.e. k is the smallest natural number such that $n|q^k - 1$;
- $P \in E(\mathbb{F}_q)[n]$, the torsion group of order n (so $nP = \mathcal{O}$);
- and $f \in \mathbb{F}_q(E)$ be such that $div_P(f) = n(P) - n(\mathcal{O})$.

For ease of notation, denote by μ_n the group of nth roots of unity in $\mathbb{F}^*_{q^k}$, so $\mu_n = \mathbb{F}^*_{q^k}/\mathbb{F}^{*n}_{q^k}$. The *Tate pairing*

$$\tau_n : E(\mathbb{F}_q)[n] \times E(\mathbb{F}_{q^k})/nE(\mathbb{F}_{q^k}) \longrightarrow \mu_n$$

is given by

$$(P, Q) \mapsto f(Q)^{(q^k - 1)/n}$$

As [52] says, "it is well known" that τ_n "is a non-degenerate bilinear pairing;" see also [29].

5.1.3 Miller's Algorithm

Given the definition for τ_n, it isn't immediately obvious how one might go about computing such a thing; thankfully Miller gave an efficient algorithm to compute it in [60]. Following [3], let $n = (n_{k-1}, \ldots, n_0)_2$ be the binary representation of n (so $n_{k-1} = 1$, and n is k bits long). Let $g_{R,S} \in \mathbb{F}_q(E)$ be "the function arising in the addition of two points R and S on E; i.e. $g_{R,S}$ is a function with $div(g_{R,S}) = (R) + (S) - (R + S) - (\mathcal{O})$." [3] Observe that in the Weierstrass case, this function can be thought of as the ratio of two lines; ℓ_1 through R and S (or tangent to R if they're equal) and one vertical one through the third point of intersection $-(R + S)$

of ℓ_1 and E. In Figure 5.5, ℓ_1 is the red diagonal line, and ℓ_2 is the green vertical one.

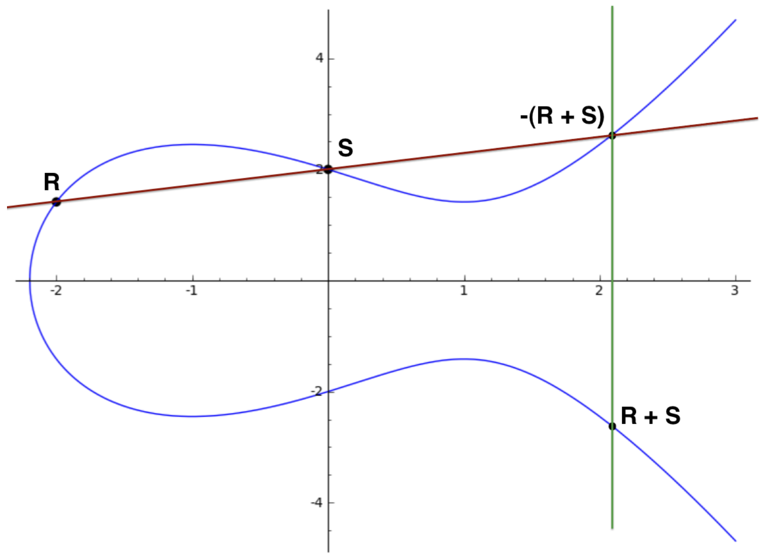


Figure 5.5: ℓ_1 and ℓ_2 over a Weierstrass curve

We can see the above claim that $g_{R,S} = {}^{\ell_1}/_{\ell_2}$ since

$$div(\ell_1) = (R) + (S) + (-(R+S)) - 3(\mathcal{O})$$

because it has zeroes at R, S, and $-(R+S)$; since principal divisors have degree zero (E is a smooth curve; see [71], section II), we know that ℓ_1 must have $\mathcal{O}$ as a pole of order three. Similarly,

$$div(\ell_2) = (-(R+S)) + (R+S) - 2(\mathcal{O})$$

since it intersects E only at $R+S$, $-(R+S)$, and $\mathcal{O}$. Finally,

$$\begin{aligned} div\left(\frac{\ell_1}{\ell_2}\right) &= (R) + (S) + (-(R+S)) - 3(\mathcal{O}) - [(R+S) + (-(R+S)) - 2(\mathcal{O})] \\ &= (R) + (S) + (R+S) - (\mathcal{O}) \\ &= div(g_{R,S}) \end{aligned}$$

```
function MILLER($P, Q$)
    $f \leftarrow 1$
    $R \leftarrow P$
    for $i \leftarrow k - 2$ downto 0 do
        $f \leftarrow f^2 \cdot g_{R,R}(Q)$
        $R \leftarrow 2R$                                        ▷ Doubling Step
        if $n_i = 1$ then
            $f \leftarrow f \cdot g_{R,P}(Q)$
            $R \leftarrow R + P$                                 ▷ Adding Step
        end if
    end for
    $f \leftarrow f^{(q^k - 1)/n}$
    return $f$
end function
```

Algorithm 5.2: Miller's Algorithm for computing τ_n

as claimed.

This leads us to defining *Miller functions* which arise in Miller's iterative algorithm for computing τ_n. We compute the needed f with $div_P(f) = n(P) - n(\mathcal{O})$ with a take on the familiar "double-and-and" routine. At each iteration we have some intermediate function f_i with divisor $i(P) - (iP) - (i-1)(\mathcal{O})$; after we complete our iterations, we have f since $nP = \mathcal{O}$. This gives us Algorithm 5.2.

This algorithm is very efficient and the Weierstrass implementation has a nice geometric intuition behind it—the "chord and tangent" rule. To compute pairings for a given normal form of an elliptic curve, it's enough to figure out what a Miller function looks like for that form, since this function "forms the backbone for pairing computation." [22] Ergo in order to implement this for Edwards curves, we'll need a way to compute some Miller function $g_{R,S}$ with $div(g_{R,S}) = (R) + (S) - (R+S) - (\mathcal{O})$. So far pairing computations in the Edwards arena have been done for twisted Edwards curves, since they cover more cases of elliptic curves. In trying to find Miller functions for binary Edwards curves, my research followed the same route that work has for twisted ones, beginning with the work of Das and Sarkar [22].

5.2 Following Das & Sarkar

In [22], the authors give a way of explicitly evaluating a pairing over twisted Edwards curves by using the birational map to and from a more familiar normal form. We do the same here, but for binary Edwards curves.[22]

Let Φ be the birational map[23] from our binary Edwards curve E to the corresponding Weierstrass curve

$$W : v^2 + uv = u^3 + a_2u^2 + a_6$$

Following the notation and work from [62], Φ is given by

$$u = \sqrt{a_6}\left(\frac{(X+Y)Z}{d_1XY + d_1^2(X+Y)Z}\right) \qquad v = \sqrt{a_6}\left(\frac{(b+1)XZ + bYZ}{d_1XY + d_1^2(X+Y)Z} + 1 + \frac{1}{d_1}\right)$$

where b is chosen such that $b^2 + b = d_1^2 + d_2 + a_2$; i.e., b is the half trace of $d_1^2 + d_2 + a_2$ (assuming that the degree n of our binary field $\mathbb{F}_{2^n}$ is odd). We also have $\Phi^{-1} : W \to E$ given by the projective coordinates

$$X = d_1(bu + v + (d-1^2 + d_1)(d_1^2 + d_1 + d_2))$$

$$Y = X + d_1u$$

$$z = u^2 + d_1u + d_1^2(d_1^2 + d_1 + d_2)$$

In order to calculate the pairing of P_1 and P_2 on E, we need to find a point P_3 and

[22]In what follows, we shift our notation slightly to follow that of [22] instead of [3] since it makes our work slightly easier in the current setting.

[23]It's really a group isomorphism when we extend it to the neutral elements, but nobody computes the pairing of a neutral element with another point in practice anyway.

a rational function $h \in \mathbb{F}_{2^n}(E)$ such that

$$\operatorname{div}(h) = (P_1) + (P_2) - (P_3) - \mathcal{O}$$

To do so, we'll map our points to $Q_1, Q_2, Q_3 \in W$ via Φ, use the function

$$g(u, v) = \frac{\ell_1(u, v)}{\ell_2(u, v)} = \frac{v + \lambda u + \theta}{u + u_3}$$

for lines ℓ_1 through Q_1 and Q_2 and ℓ_2 through Q_3 and $-Q_3$, then map back to E via Φ^{-1}. The values λ and θ are the slope and intercept of ℓ_1, which is given via straight calculation. Putting it all together, we have the following

Theorem 5.1. Let E be a binary Edwards curve over $\mathbb{F}_{2^n}$ for n odd, and $P_1 = (X_1 : Y_1 : Z_1)$ and $P_2 = (X_2 : Y_2 : Z_2)$ be two points on E with sum $P_3 = (X_3 : Y_3 : Z_3)$. Then the Miller function $h(x, y)$ such that

$$\operatorname{div}(h) = (P_1) + (P_2) - (P_3) - \mathcal{O}$$

is given by $^N/_D$, where

$$D = (u1 + u2)(u_3 d_1(d_1 XZ + d_1 YZ + XY) + \sqrt{a_6} Z(X + Y))$$

and the value of N depends on whether P_1 and P_2 are equal:

1. If $P_1 \neq P_2$, then

$$\begin{aligned} N = {} & Z(X+Y)d_1^2(v_1u_2 + u_1v_2 + u_1\sqrt{a_6} + u_2\sqrt{a_6}) \\ & + \sqrt{a_6}(u_1+u_2)d_1(XY + XZ + YZ) \\ & + YXd_1(v_1u_2 + u_1v_2) \\ & + \sqrt{a_6}(XZu_1b + YZu_1b + XZu_2b + YZu_2b \\ & + XYu_1 + XZu_1 + XZv_1 + YZv_1 + XYu_2 + XZu_2 \\ & + XZv_2 + YZv_2) \end{aligned}$$

2. If $P_1 = P_2$, then

$$\begin{aligned} N = {} & u_1Z(X+Y)d_1^2(u_1^2 + \sqrt{a_6}) \\ & + u_1d_1(XYu_1^2 + XY\sqrt{a_6} + XZ\sqrt{a_6} + YZ\sqrt{a_6}) \\ & + \sqrt{a_6}(XZu_1^2 + YZu_1^2 + XZu_1b + YZu_1b + XYu_1 + XZu_1 + XZv_1 + YZv_1) \end{aligned}$$

Proof. Given $\Phi(X,Y,Z) = (u,v)$ via the definition above, our function h is $g\left(\Phi^{-1}(u,v)\right)$ per [22]. That is, we have

$$h = g\left(\sqrt{a_6}\left(\frac{(X+Y)Z}{d_1XY + d_1^2(X+Y)Z}\right), \sqrt{a_6}\left(\frac{(b+1)XZ + bYZ}{d_1XY + d_1^2(X+Y)Z} + 1 + \frac{1}{d_1}\right)\right)$$

which is equal to

$$\frac{\sqrt{a_6}\left(\left[1 + \frac{1}{d_1} + \frac{(b+1)XZ+bYZ}{d_1XY+d_1^2(X+Y)Z}\right] + \lambda\left[\frac{(X+Y)Z}{d_1XY+d_1^2(X+Y)Z}\right]\right) + \theta}{\sqrt{a_6}\left[\frac{(X+Y)Z}{d_1XY+d_1^2(X+Y)Z} + \frac{(X_3+Y_3)Z_3}{d_1X_3Y_3+d_1^2(X_3+Y_3)Z_3}\right]} \tag{5.7}$$

where λ and θ are determined by the line ℓ_1. Observe that if $P_1 \neq P_2$, then λ is the slope of the line between them; if, on the other hand, $P_1 = P_2$, then it's the slope of the tangent line at P_1. In either case, $\theta = v_1 + \lambda u_1$.

If $P_1 \neq P_2$, a straightforward calculation yields

$$\lambda = \frac{v_2 + v_1}{u_2 + u_1} \tag{5.8}$$

(since we're in characteristic two, addition and subtraction are the same). If not, we use implicit differentiation on the equation for W to find $\lambda = \frac{dv}{du}$:

$$\begin{aligned} v^2 + uv = u^3 + a_6 u^2 + a_2 &\implies 2v\lambda + u\lambda + v = 3u^2 + 2a_6 u \\ &\implies \lambda\Big|_{(u_1,v_1)} = \frac{u_1^2 + v_1}{u_1} \end{aligned} \tag{5.9}$$

remembering again that we're in characteristic two.

Replacing λ with (5.8) and (5.9), in turn, and taking $\theta = v_1 + \lambda u_1$ yields the desired result after some tedious calculation. Rather than show all the work, we include the following Sage script that will do the heavy lifting for us:

Listing 5.3: Calculations for binary Edwards Pairings

```
var('d1 d2 x y z u v u1 v1 u2 v2 u3 v3 b a6 sqrtA6')

R.<d1, d2, x, y, z, u, v, u1, v1, u2, v2, u3, v2, b, a6, sqrtA6> = GF(2)[]

def U_V(x, y, z):
    tmp = d1 * x * y + d1^2 * (x + y) * z
    u = sqrtA6 * ((x + y) * z / tmp)
    v = sqrtA6 * (((b + 1) * x * z + b * y * z) / tmp + 1 + 1 / d1)
    return u, v
```

```

def Lambda_Theta(case):
    if case == 1:
        Lambda = (v2 + v1) / (u2 + u1)
    elif case == 2:
        Lambda = (u1^2 + v1) / u1
    Theta = v1 + Lambda * u1
    return Lambda, Theta

def g(case, (u, v)):
    Lambda, Theta = Lambda_Theta(case)
    return (v + Lambda * u + Theta) / (u + u3)

h_1 = g(1, U_V(x, y, z))
N_1, D_1 = h_1.numerator(), h_1.denominator()

h_2 = g(2, U_V(x, y, z))
N_2, D_2 = h_2.numerator(), h_1.denominator()

print(D_1 ==
      D_2 ==
      (u1 + u2) * (u3 * d1 * (d1*x*z + d1*y*z + x*y) + sqrtA6 * z * (x + y)))

print(N_1 ==
           z * (x + y) * d1^2 * (v1*u2 + u1*v2 + u1*sqrtA6 + u2*sqrtA6)
       +   sqrtA6 * (u1 + u2) * d1 * (x*y + x*z + y*z)
       +   y * x * d1 * (v1*u2 + u1*v2)
```

```
        +   sqrtA6 * (x*z*u1*b + y*z*u1*b + x*z*u2*b + y*z*u2*b + x*y*u1 +
            x*z*u1 + x*z*v1 + y*z*v1 + x*y*u2 + x*z*u2 + x*z*v2 + y*z*v2))

print(N_2 ==
            u1 * z * (x + y) * d1^2 * (u1^2 + sqrtA6)
        +   u1 * d1 * (x*y*u1^2 + x*y*sqrtA6 + x*z*sqrtA6 + y*z*sqrtA6)
        +   sqrtA6 * (x*z*u1^2 + y*z*u1^2 + x*z*u1*b + y*z*u1*b + x*y*u1 +
            x*z*u1 + x*z*v1 + y*z*v1))
```

□

5.3 Directions for Future Work

Clearly the preceding theorem, though perfectly adequate, leaves something to be desired; a more elegant, cleaner solution for the Miller function would be nice. The most clear way to such a solution, as evidenced by [3] and [54], is to get a better understanding of the geometry of binary Edwards curves. To that end, we show how to extend one of the theorems of [3] from twisted Edwards curves to binary Edwards curves. We intend this theorem to be a stepping stone towards a full result similar to the main idea of [3], wherein they give a new geometric formulation of the twisted Edwards curve group law (among many other interesting results).

In [3], the authors give a new interpretation of the group law similar in its geometric flavor to the familiar "chord-and-tangent" law of Weierstrass curves. They show that the sum of two points on a twisted Edwards curve $E_{T,a,d}$ can be given by a special conic: quoting their *Remark 3*, "we see that $P_1 + P_2$ is obtained as the mirror image with respect to the y-axis of the eighth intersection point of $E_{T,a,d}$ and the conic passing through $\Omega_1, \Omega_2, \mathcal{O}', P_1$, and P_2," where Ω_1 and Ω_2 are the two points at infinity $(1:0:0)$ and $(0:1:0)$, respectively, and $\mathcal{O}'$ is the point $(0,-1)$ of order 2. We offer an analogous result to their main theorem (1) that paves the way for their geometric interpretation in the hopes that it will inspire similar results for binary Edwards curves.

Let $\mathcal{O}' = (1,1) = (1:1:1)$; recall that this point has order 2. Furthermore, let $\varphi(X,Y,Z) = c_{X^2}X^2 + c_{Y^2}Y^2 + c_{Z^2}Z^2 + c_{XY}XY + c_{XZ}XZ + c_{YZ}YZ \in K[X,Y,Z]$ be a homogeneous polynomial of degree 2 and $C : \varphi(X,Y,Z) = 0$ be the associated plane (possibly degenerate) conic. Like in [3], since the points Ω_1, Ω_2, and $\mathcal{O}'$ do not lie on a line, a conic C passing through these points cannot be a double line and φ represents C uniquely up to multiplication by a scalar. By evaluating φ at Ω_1, Ω_2, and $\mathcal{O}'$, we get the following:

$$\begin{aligned}
\Omega_1 :& \quad \varphi(1:0:0) = c_{X^2} \\
\Omega_2 :& \quad \varphi(0:1:0) = c_{Y^2} \\
\mathcal{O}' :& \quad \varphi(1,0,0) = c_{X^2} + c_{Y^2} + c_{Z^2} + c_{XY} + c_{XZ} + c_{YZ}
\end{aligned}$$

Hence $c_{X^2} = c_{Y^2} = 0$ and $c_{Z^2} = c_{XY} + c_{XZ} + c_{YZ}$. Therefore C must have the form:

$$c_{XY}(XY + Z^2) + c_{XZ}(XZ + Z^2) + c_{YZ}(YZ + Z^2) \tag{5.10}$$

Next we have the following analogous result to Theorem 1 of [3]:

Theorem 5.2. Let E_{B,d_1,d_2} be a binary Edwards curve over K and let $P_1 = (X_1 : Y_1 : Z_1)$ and $P_2 = (X_2 : Y_2 : Z_2)$ be two affine, not necessarily distinct, points on $E_{B,d_1,d_2}(K)$. Let C be the conic passing through $\Omega_1, \Omega_2, \mathcal{O}', P_1$, and P_2 which must have the form (5.10). If some of the above points are equal, we consider C and E_{B,d_1,d_2} to intersect with at least that multiplicity at the corresponding point. Then the coefficients in (5.10) of the equation φ of the conic C are uniquely determined (up to scalars) as follows:

1. If $P_1 \neq P_2, P_1 \neq \mathcal{O}'$, and $P_2 \neq \mathcal{O}'$, then

$$\begin{aligned} c_{XY} &= Z_1 Z_2 \left[X_1(Y_2 + Z_2) + Y_1(X_2 + Z_2) + Z_1(X_2 + Y_2)\right] \\ c_{XZ} &= Y_1 Z_2 (X_1 Y_2 + X_1 Z_2 + Z_1 Z_2) + Y_2 Z_1 (Y_1 X_1 + Z_1 X_1 + Z_1 Z_2) \\ c_{YZ} &= X_1 Z_2 (Y_1 X_2 + Y_1 Z_2 + Z_1 Z_1) + X_2 Z_1 (X_1 Y_2 + Z_1 Y_2 + Z_1 Z_2) \end{aligned}$$

2. If $P_1 \neq P_2 = \mathcal{O}'$, then $c_{XY} = Z_1, c_{XZ} = Z_1$, and $c_{YZ} = X_1$
3. If $P_1 = P_2$, then

$$\begin{aligned} c_{XY} &= X_1^2 Y_1 + X_1 Y_1^2 + d_1 X_1 Z_1^2 + X_1^2 Z_1 + d_1 Y_1 Z_1^2 + Y_1^2 Z_1 + X_1 Z_1^2 + Y_1 Z_1^2 \\ c_{XZ} &= X_1^2 Y_1 + d_1 Y_1^2 Z_1 + X_1 Y_1^2 + d_1 X_1 Z_1^2 + X_1^2 Z_1 \\ &\quad + d_1 Y_1 Z_1^2 + X_1 Y_1 Z_1 + d_1 Z_1^3 + Y_1 Z_1^2 \\ c_{YZ} &= d_1 X_1^2 Z_1 + d_2 X_1^2 Z_1 + d_2 Y_1^2 Z_1 + X_1^2 Z_1 + d_1 Z_1^3 + X_1 Z_1^2 \end{aligned}$$

Proof. We tackle each case separately.

1. If the points P_1 and P_2 are distinct, evaluating equation 5.10 at the two points yields two linear equations in the coefficients:

$$\begin{aligned} c_{XY}(X_1 Y_1 + Z_1^2) + c_{XZ}(X_1 Z_1 + Z_1^2) + c_{YZ}(Y_1 Z_1 + Z_1^2) &= 0 \\ c_{XY}(X_2 Y_2 + Z_2^2) + c_{XZ}(X_2 Z_2 + Z_2^2) + c_{YZ}(Y_2 Z_2 + Z_2^2) &= 0 \end{aligned}$$

Since we're working in projective coordinates, two equations is enough to uniquely determine the three unknowns, and we get the following solutions:

$$c_{XY} = \begin{vmatrix} X_1Z_1 + Z_1^2 & Y_1Z_1 + Z_1^2 \\ X_2Z_2 + Z_2^2 & Y_2Z_2 + Z_2^2 \end{vmatrix}$$

$$= (X_1Z_1 + Z_1^2)(Y_2Z_2 + Z_2^2) + (X_2Z_2 + Z_2^2)(Y_1Z_1 + Z_1^2)$$

$$= Z_1Z_2\left[X_1(Y_2 + Z_2) + Y_1(X_2 + Z_2) + Z_1(X_2 + Y_2)\right]$$

$$c_{XZ} = \begin{vmatrix} X_1Y_1 + Z_1^2 & Y_1Z_1 + Z_1^2 \\ X_2Y_2 + Z_2^2 & Y_2Z_2 + Z_2^2 \end{vmatrix}$$

$$= (X_1Y_1 + Z_1^2)(Y_2Z_2 + Z_2^2) + (X_2Y_2 + Z_2^2)(Y_1Z_1 + Z_1^2)$$

$$= X_1^2Y_1 + d_1Y_1^2Z_1 + X_1Y_1^2 + d_1X_1Z_1^2 + X_1^2Z_1$$

$$+ d_1Y_1Z_1^2 + X_1Y_1Z_1 + d_1Z_1^3 + Y_1Z_1^2$$

$$c_{YZ} = \begin{vmatrix} X_1Y_1 + Z_1^2 & X_1Z_1 + Z_1^2 \\ X_2Y_2 + Z_2^2 & X_2Z_2 + Z_2^2 \end{vmatrix}$$

$$= (X_1Y_1 + Z_1^2)(X_2Z_2 + Z_2^2) + (X_2Y_2 + Z_2^2)(X_1Z_1 + Z_1^2)$$

$$= d_1X_1^2Z_1 + d_2X_1^2Z_1 + d_2Y_1^2Z_1 + X_1^2Z_1 + d_1Z_1^3 + X_1Z_1^2$$

as claimed.

2. Note that C is tangent to the curve E_{B,d_1,d_2} at the point $\mathcal{O}'$ if and only if $0 = ({}^{\partial\varphi}/_{\partial x})\,(1:1:1) = c_{XY} + c_{XZ}$, i.e. iff $c_{XY} = c_{XZ}$. Then

$$\varphi = c_{XY}(XY + Z^2 + XZ + Z^2) + c_{YZ}(YZ + Z^2) = (Y + Z)(c_{XY}X + c_{YZ}Z)$$

Since $P_1 \neq \mathcal{O}'$, it doesn't lie on the line $Y + Z = 0$, so $c_{XY}X_1 + c_{YZ}Z_1 = 0$.

Then together $c_{XY} = c_{XZ}$ and $c_{XY}X_1 = c_{YZ}Z_1$ imply the result.

3. In the final case, let $Z = Z_1 = 1$ in our equations. The tangent vectors at $P_1 = (X_1 : Y_1 : 1) = (X_1, Y_1)$ of E_{B,d_1,d_2} and C are

$$\begin{pmatrix} {}^{\partial E}\!/_{\partial Y} \\ \\ {}^{\partial E}\!/_{\partial X} \end{pmatrix} = \begin{pmatrix} d_1 + X_1 + X_1^2 \\ \\ d_1 + Y_1 + Y_1^2 \end{pmatrix} \qquad \begin{pmatrix} {}^{\partial C}\!/_{\partial Y} \\ \\ {}^{\partial C}\!/_{\partial X} \end{pmatrix} \begin{pmatrix} c_{XY}X_1 + c_{YZ} \\ \\ c_{XY}Y_1 + c_{XZ} \end{pmatrix}$$

(note that we can drop the usual negative signs because $\mathrm{char}(K) = 2$). These vectors are collinear if and only if

$$0 = \begin{vmatrix} d_1 + X_1 + X_1^2 & c_{XY}X + c_{YZ} \\ \\ d_1 + Y_1 + Y_1^2 & c_{XY}Y + c_{XZ} \end{vmatrix}$$

$$= c_{XY}(d_1Y_1 + X_1Y_1 + X_1^2Y_1 + d1X_1 + X_1Y_1 + X_1Y_1^2)$$

$$+ c_{XZ}(d_1 + X_1 + X_1^2) + c_{YZ}(d_1 + Y_1 + Y_1^2)$$

$$= c_{XY}(d_2(X_1^2 + Y_1^2) + X_1^2Y_1^2) + c_{XZ}(d_1 + X_1 + X_1^2) + c_{YZ}(d_1 + Y_1 + Y_1^2)$$

using the curve equation to simplify. We also know that

$$0 = \varphi(X_1, Y_1, 1) = c_{XY}(X_1Y_1 + 1) + c_{XZ}(X_1 + 1) + c_{YZ}(Y_1 + 1)$$

These two equations can be solved in the same manner as our work in the first case, yielding

$$\begin{aligned}
c_{XY} &= \begin{vmatrix} d_1 + X_1 + X_1^2 & d_1 + Y_1 + Y_1^2 \\ X_1 + 1 & Y_1 + 1 \end{vmatrix} \\
&= d_1(X_1 + Y_1) + X_1 + Y_1 + X_1^2 + Y_1^2 + X_1^2Y_1 + X_1Y_1^2 \\
&= (X_1 + Y_1)(X_1Y_1 + X_1 + Y_1 + d_1 + 1)
\end{aligned}$$

for the first coefficient,

$$\begin{aligned}
c_{XZ} &= \begin{vmatrix} d_2(X_1^2 + Y_1^2) + X_1^2Y_1^2 & d_1 + Y_1 + Y_1^2 \\ X_1Y_1 + 1 & Y_1 + 1 \end{vmatrix} \\
&= d_1(X_1Y_1 + 1) + d_2(X_1^2Y_1 + Y_1^3 + X_1^2 + Y_1^2) + X_1^2 \\
&+ Y_1^2 + X_1Y_1^2 + X_1Y_1^3 + Y_1 + Y_1^2 + X_1^2Y_1^3 \\
&= X_1^2Y_1 + d_1Y_1^2 + X_1Y_1^2 + d_1X_1 + X_1^2 + d_1Y_1 + X_1Y_1 + d_1 + Y
\end{aligned}$$

for the second, and

$$\begin{aligned}
c_{YZ} &= \begin{vmatrix} d_2(X_1^2 + Y_1^2) + X_1^2Y_1^2 & d_1 + X_1 + X_1^2 \\ X_1Y_1 + 1 & X_1 + 1 \end{vmatrix} \\
&= d_1(X_1Y_1 + 1) + d_2(X_1^3 + X_1^2 + X_1Y_1^2 + Y_1^2) + X_1^34Y_1^2 \\
&+ X_1^2Y_1^2 + X_1^2Y_1 + X_1^3Y_1 + X_1 + X_1^2 \\
&= d_1X_1^2 + d_2X_1^2 + d_2Y_1^2 + X_1^2 + X_1 + d_1
\end{aligned}$$

for the third, using the curve equation to simplify. Homogenizing yields the stated result. Note that the same formulas work if $P_1 = \mathcal{O}'$, since then we still have $\varphi = 0$. $\square$

Unfortunately, it's not entirely clear where to go from here; the geometry of twisted Edwards curves is different from the geometry of binary Edwards curves. Moreover, though working in characteristic two has some benefits to arithmetic, more often than not it seems to complicate calculations. Therefore, theorem 5.2 is offered as a possible starting point for future research instead of an end in and of itself. If we were to continue to mirror the progression of results for pairings on twisted Edwards curves, the next step would be to expand on a result similar to [3]'s to get one similar to [54]'s. Again, because the geometry of binary Edwards curves differs so much from that of twisted means the results of this paper don't directly apply, but they do offer an intriguing possibility for another direction. Such a result would involve not only reinterpreting the geometry of binary Edwards curves, but would also involve working in extended coordinates (four instead of the usual three for projective space).

CHAPTER 6: APPLICATIONS

In this chapter we discuss two applications of elliptic curve cryptography, both of which benefit from the added security granted by the binary Edwards group law. Moreover, they may be more attractive to implementors because they use binary Edwards curves rather than some other type; computers do work in binary, after all, so binary Edwards curves can lend themselves to efficient implementation in software or even hardware (e.g. [15], [50], [51]).

6.1 Password Based Key Derivation

6.1.1 Background

Our first application is a password based key derivation function, or PBKDF. Password safety is paramount in today's interconnected world; users log in to multiple workstations, websites, and services for communication, work, banking—the list goes on. Despite its importance, password safety still a tricky technical subject, one that even experts get wrong sometimes; for example, according to [33], the IEEE exposed plaintext passwords in a public FTP directory "for over a month." One way to securely store passwords is, somewhat paradoxically, to not store them at all. Instead, a system can use a PBKDF to store different information derived from a user's login credentials to authenticate them.

To quote [65],

> Password-based key derivation functions are used for two primary purposes: First, to hash passwords so that an attacker who gains access to a password file does not immediately possess the passwords contained therein; and second, to generate cryptographic keys to be used for encrypting and/or authenticating data. ...Since all modern key derivation functions are constructed from hashes against which no non-trivial preimage attacks are known, attacking the key derivation function directly is infeasible; consequently, the best attack in either case is to iterate through likely passwords and apply the key derivation function to each in turn. Unfortunately, this form of "brute force" attack is quite liable to succeed. Users often select passwords which have far less entropy than is typically required of cryptographic keys; a recent study found that even for web sites such as `paypal.com`, where—since accounts are often linked to credit cards and bank accounts—one would expect users to make an effort to use strong passwords, the average password has an estimated entropy of 42.02 bits, while only a very small fraction had more than 64 bits of entropy.[24] This is where a properly designed PBKDF comes in.

As [77] says, "the main idea of a PBKDF is to slow dictionary or brute force attacks on the passwords by increasing the time needed to test each password." Ideally, it should behave like a random mapping from passwords to possible data, which we'll called *password hashes*, though this term is somewhat problematic.[25] To slightly

[24]The cited study is [27].

[25]Using "hashes" may lead one to think that using a general-purpose hash function like SHA-256 as a PBKDF is a good idea; as we'll see, this is not the case.

borrow some of [57]'s exposition, "a password, associated with each user (entity), is typically a string of 6 to 10 or more characters the user is capable of committing to memory." In order to authenticate herself to the system in question, "the user enters a (userid, password) pair" to the system, which then uses this information in some way to compute the hash. Once this computation is complete, the system checks the hash against the credentials it has stored for the supplied userid; if the hash matches the one on file, the user is granted access to the system.

Clearly a string of 6 to 10 memorable characters may not have enough entropy to qualify as cryptographically secure; therefore, a PBKDF should be designed to make the hash output look as random as possible. Randomness alone isn't enough, however; as [65] mentions above, cryptographic security can be compromised if the PBKDF is too computationally simple to perform; consider the following example.

Example 6.1. Suppose an eavesdropper Eve manages to get her hands on the table of

$$[\text{userid}, \text{hash}(\text{password})]$$

pairs for Alice's system. If Eve's desire to break into Alice's system isn't particularly time sensitive, she can simply grab a large file of likely passwords and hash them all until she finds a match in the second column of the table. If Alice chose to use a general purpose message hashing algorithm for her PBKDF like SHA-3 ([5]), Eve may have the computational power to break into Alice's system soon enough to cause severe damage.

One way to combat this "dictionary attack" is to widen the search space by *salting* the hashes; for each userid, a system may instead store $\text{hash}(\text{password} * \text{salt})$ for some operation $*$, typically string concatenation or bit exclusive-or, where the salt is a secret value known only to the system. In this setup, a potential attacker would have to brute-force over all possible salts as well as all possible passwords, greatly increasing the work involved. Even with salting, however, the speed of hash can still be an issue given today's technology. There are a number of recent publications regarding cracking password hashes by brute force, many of which use the advanced parallel computing power granted by today's GPUs—see [2, 32, 55], and [83].

To make matters worse, the inevitable increase in computing power we experience as technology changes means that attackers will be able to attack any fixed PBKDF more and more easily as time goes by. This means that a successful PBKDF should be tunable to meet this rising power available to would-be password crackers; as [66] puts it, we are looking for a future-adaptable password scheme to "keep up with hardware speeds." A secure PBKDF's computational cost "must increase as hardware improves."

One other successful PBKDF is [66]'s `bcrypt`. However, it isn't the final answer to the PBKDF problem; `bcrypt` is already under some scrutiny, and some alternatives have been proposed, the most notable of which is probably [65]'s `scrypt`. Like `scrypt`, our proposed PBKDF will incorporate a pseudorandom number generator. It will also make use of binary Edwards curves; although they can be implemented rather efficiently, the inherent complexity of binary Edwards curves compared to the computer primitives of which typical hash functions are built adds to the security of our PBKDF.

6.1.2 Proposed PBKDF

To further research into and development of secure PBKDFs, I propose we take a cue from NIST: look at a candidate scheme that is very different from what currently exists, so they won't (necessarily) be vulnerable to the same attacks, like NIST did with the recent SHA-3 competition. Explaining some of the reasons for declaring `Keccak` the winner, they wrote

> "Keccak has the added advantage of not being vulnerable in the same ways SHA-2 might be,"says NIST computer security expert Tim Polk. "An attack that could work on SHA-2 most likely would not work on Keccak because the two algorithms are designed so differently." [5]

In fact, we'll take even more from NIST's recent competition and present a variation on the Elliptic Curve Only Hash (ECOH, [13]) which was submitted to the SHA-3 competition in 2008. This variation makes some changes to the original algorithm to combat the main weakness that lost it the competition, viz. the second pre-image attack found in [37, 38]. At the same time, these changes ensure that while several different instances of our PBKDF can be parallelized, the algorithm itself is inherently serial and thus resists any further attempts at parallelization.

Specifically, the points P_i and the values X_1 and X_2 presented in this section rely on the state of the algorithm at every step. That is, each P_i for $i > 0$ relies upon the previous P_{i-1}, and both X_1 and X_2 rely on the all of the P_i. We also incorporate a pseudorandom number generator, not unlike other PBKDFs like `scrypt` [65]. As mentioned, these changes effectively ameliorate the second pre-image attack from [38];

they also have force the different stages of the hash to be computed serially, removing any chance of parallelization. This makes our PBKDF more resilient in the face of processors that are no longer scaling up to greater speeds but out to more and more cores; for an offline attack, only multiple instances of our PBKDF can be run on a parallel machine or GPU. The PBKDF itself can't be sped up.

Finally, we make some changes to the parameters involved. Rather than sticking to an Elliptic curve in Weierstrass form, we make use of a binary Edwards curve; this makes the PBKDF more resistant to side-channel analysis. We also remove the nondeterministic part of the *Search* step of ECOH, instead using the ideas from [41] to deterministically map a field element to a point on our curve. In doing so, we incorporate the birational map from [62].

Our PBKDF, called `ECOH's Echo`, takes in a salt s and password p and makes use of the following parameters:

- a pseudorandom number generator PRNG that can be seeded (primed with an initial state)
- a block size $blen$, number of blocks/rounds $numblocks$, length $ilen$ for integer bit representations, and an output length
- a finite field $\mathbb{F}_{2^n}$, an algebraic extension of $\mathbb{F}_2$ of degree n
- a_2 and a_6, two elements of $\mathbb{F}_{2^n}$ determining an elliptic curve in Weierstrass form
- E, the birationally equivalent binary Edwards curve
- G, a base point on E

Note that our parameters are not set in stone, but rather can be scaled up to larger sizes as potential attacking computing power increases. We can increase our degree n as well as the other parameters as the need arises.

We also make use of a number of maps:

- π, a deterministic map from K to E (see [41] and [62]) that takes a fixed number of steps

- A mapping $\omega : E \times \{X, Y\} \to K$, given by

  $$(Q, Z) \mapsto \frac{\left\lfloor Q + \left\lfloor \frac{Q.z}{2} \right\rfloor G \right\rfloor .z}{2} \bmod 2^{blen}$$

  where $z = x$ if $Z = X$ and y if $Z = Y$ (i.e., Z picks out the coordinate). This is an extension of the output mapping from the original ECOH submission ([13]) to work with either of a point's coordinates.

- A mapping $\varphi : E \to K$ given by $\varphi(Q) = \omega(Q, X)$ (not strictly needed, of course, this is just a specialization of ω)

- A mapping $\psi : E \to \mathbb{F}_2$ given by $\psi(Q) = \omega(Q, Y)\&1$, the least significant bit of $\omega(Q, Y)$

Before we move on, let's flesh out the details of the map π. Take [41]'s $f_{a_2,a_6} : \mathbb{F}_{2^n} \to W(\mathbb{F}_{2^n})$ where W is an elliptic curve in the Weierstrass form

$$v^2 + uv = u^3 + a_2u^2 + a_6$$

defined by $z \mapsto (u, v)$ such that

$$\alpha = a_2 + z + z^2$$

$$u = (\alpha^4 + \alpha^3 + a_6)^{1/3} + \alpha$$

$$v = zu + \alpha^2$$

and combine it with [62]'s birational equivalence to create π via $z \mapsto (u, v) \mapsto (x, y)$.[26] This gives us a direct mapping $\pi : \mathbb{F}_{2^n} \to E(\mathbb{F}_{2^n})$ that inherits all of the cryptographically desirable properties of f_{a_2,a_6} since the birational equivalence has no exceptional points save ∞. Moreover, it's more secure than the *Search* step of ECOH, since this was a nondeterministic map that would take an indeterminate amount of steps. As always, we strive to minimize the leakage of extra information through side-channels.

We'll first give a more naïve version of `ECOH's Echo`.

6.1.2.1 Naïve Version

Execution proceeds as follows: first, our PRNG is seeded with the exclusive-or (XOR) of the password and salt.[27] Each block O_i is generated in the following way: given an integer i, append the *ilen*-bit representation of i to a $blen - ilen$ long stream of pseudorandom bits from PRNG. Second, we initialize a point P_0 from the initial block O_0 by setting it to be $\pi(O_0)$. We then iterate for $numblocks - 1$ rounds, setting

[26]We use this one instead of the original given in [8] due to its having only one exceptional point (∞ which won't show up here) and being more efficient in implementation.

[27]Strictly speaking, there is an encoding involved in making an integer value out of the password. In our reference implementation below, we make the usual choice of treating the password as a string of ASCII-encoded bytes and build an integer out of that.

P_i to be $\pi(O_i \oplus i \oplus \varphi(P_{i-1}))$. Next we update *numblocks*, saving

$$\mathit{numblocks} \oplus \left(\sum_0^{\mathit{numblocks}-1} 2^i \psi(P_i)\right)$$

in its place. After that, X_1 and X_2 are defined to be $\pi(\mathit{numblocks})$ and

$$\pi\left(\mathit{numblocks} \oplus \left[\bigoplus_0^{\mathit{numblocks}-1} O_i\right]\right)$$

respectively. Finally, we define Q to be the sum

$$X_1 + X_2 + \sum_0^{\mathit{numblocks}-1} P_i$$

and output $\varphi(Q)$.

6.1.2.2 Memory Efficient Version

Though it's simpler to follow, the above version of our PBKDF is not as efficient with regards to memory as it could be. We needn't store all the intermediate points P_i, for instance, if we instead introduce two temporary variables Y_1 and Y_2 that will be used to build X_1 and X_2 later. After seeding PRNG as before, let $(O, Y_1, Y_2) = (O_0, \mathit{numblocks}, \mathit{numblocks})$ and set $Q = P = \pi(O)$. Then we iterate for $i \in \{1, \ldots, \mathit{numblocks} - 1\}$, setting

$$O = O_i$$

$$P = \pi(O \oplus i \oplus \varphi(P))$$

$$Y_1 = Y_1 + 2^i \psi(P)$$

$$Y_2 = Y_2 \oplus O$$

$$Q = Q + P$$

```
function ECOH'S ECHO$(p, s)$
    PRNG.salt$(s \oplus p)$
    $P_0 \leftarrow \pi(O_0)$
    for $i \leftarrow 1$ to $numblocks - 1$ do
        $P_i \leftarrow \pi\left(O_i \oplus i \oplus \varphi(P_{i-1})\right)$
    end for
    $numblocks \leftarrow numblocks \oplus \left(\sum_0^{numblocks-1} 2^i \psi(P_i)\right)$
    $X_1 \leftarrow \pi(numblocks)$
    $X_2 \leftarrow \pi\left(numblocks \oplus \left[\bigoplus_0^{numblocks-1} O_i\right]\right)$
    $Q \leftarrow X_1 + X_2 + \sum_0^{numblocks-1} P_i$
    return $\varphi(Q)$
end function
```

Algorithm 6.3: ECOH's Echo, Naïve version

at each step of the loop. Finally, let $X_1 = \pi(Y_1)$, $X_2 = \pi(Y_1 \oplus Y_2)$, and $Q = Q + X_1 + X_2$. As before, we output $\varphi(Q)$.

6.1.3 Algorithm Pseudocode and Diagrams

We give a description of `ECOH's Echo` in algorithms (6.3) and (6.4). Diagrams of the flow of the rounds in the more efficient description are given in Figures (6.6) and (6.7).

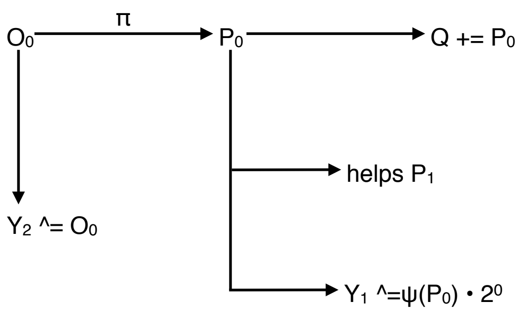


Figure 6.6: First round of `ECOH's Echo`

```
function ECOH's Echo$(p, s)$
    PRNG.salt$(s \oplus p)$
    $O \leftarrow O_0$
    $Y_1 \leftarrow numblocks$
    $Y_2 \leftarrow numblocks$
    $P \leftarrow \pi(O)$
    $Q \leftarrow P$
    for $i \leftarrow 1$ to $numblocks - 1$ do
        $O \leftarrow O_i$
        $P \leftarrow \pi\left(O \oplus i \oplus \varphi(P)\right)$
        $Y_1 \leftarrow Y_1 \oplus 2^i \psi(P)$
        $Y_2 \leftarrow Y_2 \oplus O$
        $Q \leftarrow Q + P$
    end for
    $X_1 \leftarrow \pi(Y_1)$
    $X_2 \leftarrow \pi\left(Y_1 \oplus Y_2\right)$
    $Q \leftarrow Q + X_1 + X_2$
    return $\varphi(Q)$
end function
```

Algorithm 6.4: ECOH's Echo, Memory-Efficient Version

6.1.4 `ECOH's Echo` Reference Implementation

Finally, we provide a reference implementation of `ECOH's Echo` using our software library `e2c2`. For more on `e2c2`, see Appendix 7.4.4.

Listing 6.4: ECOH's Echo

```
/**
 * @file ecoh_echo.cc
 * @brief Elliptic Curve only key derivation function
 * @author Graham Enos
 *
 */
#include <algorithm>
#include <iostream>
#include <sstream>
```

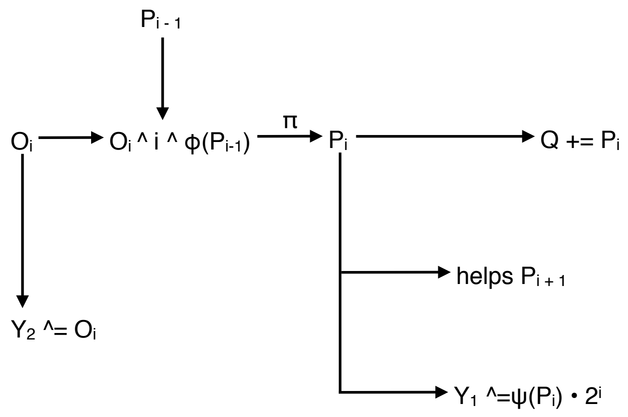


Figure 6.7: Subsequent rounds of `ECOH's Echo`

```
#include "e2c2.h"

using namespace std;
using namespace NTL;
using namespace e2c2;

//------- Constants --------//
/// Number of blocks
const size_t NUM_BLOCKS = 32;
/// Length of each block
const size_t B_LEN = 192;
/// Bit length of integer representation
const size_t I_LEN = 64;


//------- Utilities -------//
```

```
/// Helps omega() decide which coordinate to use
enum class Coord { X, Y };

string ZZtoHex(const ZZ& z) {
    /// The hex representation of z
    static const size_t out_len = (B_LEN + I_LEN) >> 4;
    static const string digits = "0123456789abcdef";
    stringstream ss;
    if (z <= 0) {
        ss << "0x";
        while (ss.str().length() < out_len + 2) {
            ss << "0";
        }
        return ss.str();
    } else {
        auto zz = z;
        while (zz > 0) {
            ss << digits.at(to_long(zz & 0xf));
            zz  >>= 4;
        }
        while (ss.str().length() < out_len) {
            ss << "0";
        }
        ss << "x0";
        auto s = ss.str();
        reverse(s.begin(), s.end());
        return s;
    }
}

ZZ StoZZ(const string& s) {
    /// Treat a string of characters as a bunch of bytes (ASCII), then use
    /// those bytes to build a ZZ
    size_t i = 0;
    auto z = ZZ::zero();
    for (auto c : s) {
        ++i;
        z <<= 8;
        z |= to_ZZ(c);
    }
    /// Extra NOPS (to protect against side-channel/timing attacks)
    while (i < (B_LEN >> 3)) {
        ++i;
    }
    return z;
}

ZZ GF2EtoZZ(const GF2E& x) {
    /// Use the bit representation of x to create a ZZ
    GF2X::HexOutput = false;
    auto z = ZZ::zero();
    stringstream ss;
    ss << x;
    auto s = ss.str();
    /// Only leave behind 0s and 1s (remove spaces and brackets)
    for (auto c : string("[ ]")) {
        auto i = remove(s.begin(), s.end(), c);
        while (i != s.end()) {
            s.erase(i);
            i = remove(s.begin(), s.end(), c);
        }
    }
    /// Set bits where there are 1s
    for (auto c : s) {
        z <<= 1;
        z |= to_ZZ(c == '1') & 0x1 ;
    }
    return z;
}

GF2E ZZtoGF2E(const ZZ& z) {
    /// Use the bit representation of z to create a GF2E
    auto x = GF2E::zero();
    stringstream ss;
    ss << "[";
    for (auto i = 0; i < NumBits(z); ++i) {
        ss << bit(z, i) << " ";
    }
    ss << "]";
    ss >> x;
    return x;
```

```
}



//------- ECOH's Echo Helpers -------//
ZZ block(const size_t i) {
    /// The ith block O_i is (random)^{B_LEN - I_LEN} || {I_LEN bit repr of i},
    /// similar to the original ECOH submission
    auto O = (RandomBits_ZZ(B_LEN - I_LEN)) << I_LEN;
    O |= to_ZZ(i) & ((to_ZZ(1) << I_LEN) - 1);
    return O;
}

ZZ omega(const BinaryAff& Q, const BinaryAff& G, const Coord z) {
    /// Output, based on coordinate, of bit-length B_LEN
    if (z == Coord::X) {
        return GF2EtoZZ((Q + (GF2EtoZZ(Q.x) / 2) * G).x) / 2 %
            power2_ZZ(B_LEN);
    } else { /// z == Coord::Y
        return GF2EtoZZ((Q + (GF2EtoZZ(Q.y) / 2) * G).y) / 2 %
            power2_ZZ(B_LEN);
    }
}

BinaryAff pi(const GF2E& z, const GF2E& a2, const GF2E& a6,
        const BinaryCurve& E) {
    /// Uses Icart's f from "How to Hash into Elliptic Curves" and Moloney,
    /// O'Mahony, & Laurent's birational map from "Efficient Implementation of
    /// Elliptic Curve Point Operations Using Binary Edwards Curves"
    const auto alpha = a2 + z + sqr(z);
    const auto u = power(power(alpha, 4) + power(alpha, 3) + a6,
                         (2 * GF2E::cardinality() - 1) / 3) + alpha;
    const auto v = z * u + sqr(alpha);
    return birMapAff(u, v, a2, E);
}


//------- ECOH's Echo -------//
ZZ ecoh_echo(const ZZ& password, const ZZ& salt, const GF2E& a2,
        const GF2E& a6, const BinaryCurve& E, const BinaryAff& G) {
    /// Our PBKDF

    /// Helpers defined as lambda expressions
    auto phi = [&G](const BinaryAff& P){ return omega(P, G, Coord::X); };
    auto psi = [&G](const BinaryAff& P){ return omega(P, G, Coord::Y) & 1; };

    /// Memory efficient version
    SetSeed(salt ^ password);
    auto O = block(0);
    auto X_1_tmp = to_ZZ(NUM_BLOCKS);
    auto X_2_tmp = to_ZZ(NUM_BLOCKS);
    auto P = pi(ZZtoGF2E(O), a2, a6, E);
    auto Q = P;
    for (size_t i = 1; i < NUM_BLOCKS; ++i) {
        O = block(i);
        P = pi(ZZtoGF2E(O ^ i ^ phi(P)), a2, a6, E);
        X_1_tmp ^= (psi(P) << i);
        X_2_tmp ^= O;
        Q += P;
    }
    auto X1 = pi(ZZtoGF2E(X_1_tmp), a2, a6, E);
    auto X2 = pi(ZZtoGF2E(X_1_tmp ^ X_2_tmp), a2, a6, E);
    Q += X1 + X2;
    return phi(Q);
}


//------- An Example -------//
int main(int argc, char *argv[]) {
    //------- Error Checking -------//
    if (argc != 3) {
        cerr << "usage: " << argv[0] << " salt password" << endl;
        return EXIT_FAILURE;
    }

    //------- Setup -------//
    /// Our irred. polynomial is x^163 + x^7 + x^6 + x^3 + 1, per FIPS 186-3
    GF2E::init(GF2X(163, 1) + GF2X(7, 1) + GF2X(6, 1) + GF2X(3, 1) +
               GF2X(0, 1));
    GF2X::HexOutput = true; /// more compact output
    auto a2 = to_GF2E(1), a6 = GF2E::zero();
    /// a6 = b in Fips 186-3 language
    set_parameter(a6, "20a601907b8c953ca1481eb10512f78744a3205fd", true);
    auto E = from_weierstrass(163,
            to_ZZ("5846006549323611672814742442876390689256843201587"),
            a2,
            a6);
    auto u = GF2E::zero(), v = GF2E::zero();
    set_parameter(u, "3f0eba16286a2d57ea0991168d4994637e8343e36", true);
    set_parameter(v, "0d51fbc6c71a0094fa2cdd545b11c5c0c797324f1", true);
    auto x = GF2E::zero(), y = GF2E::zero();
    BinaryAff G = birMapAff(u, v, a2, E);

    //------- Run ECOH's ECHO -------//
    auto salt = to_ZZ(argv[1]);
    auto password = StoZZ(argv[2]);
    cout << ZZtoHex(ecoh_echo(password, salt, a2, a6, E, G)) << endl;

    return EXIT_SUCCESS;
}
```

6.2 Compartmented ID-Based Secret Sharing and Signcryption

Another application of elliptic curves and elliptic curve cryptography is using pairings for identity-based schemes, like those first suggested in [10].[28] In [53], Li, Xin, and Hu describe an ID-based signcryption scheme that uses a bilinear map to accomplish (t, n) shared unsigncryption with the help of Shamir's secret sharing scheme.

[28]A previous of this section has been posted to the International Association for Cryptologic Research's cryptology eprint archive (http://eprint.iacr.org/2012/528) and has been submitted for consideration to the Information Processing Letters Journal for consideration for publication.

Here we describe a way to extend Li et. al.'s construction into a *compartmented scheme*. For our compartmented scheme, suppose the organization $\mathcal{O}$ is split into several compartments $\mathcal{C}_i$, $i \in \{1, \ldots, t\}$. In order to unsigncrypt a message sent to $\mathcal{O}$, at least one member of each of the t compartments must participate; without the cooperation of at least one member from each compartment, the message cannot be unsigncrypted. What's more, each member $\mathcal{M}_{ij} \in \mathcal{C}_i$ gets different information; therefore, although any $\mathcal{M}_{ij}$ can participate equally, the compartment $\mathcal{C}_i$ is in fact split up so that all of its potential participants have something unique to contribute.

In what follows, we will make the following changes to the terminology and notation of [53]: uppercase letters will denote points on an elliptic curve E over a predetermined finite field K, lowercase letters will denote elements in the multiplicative group μ_n of nth roots of unity, Greek letters are used for elements of $\mathbb{F}_q$, and script letters generally denote compartments or members thereof. Moreover, $\widehat{e}$ is a pairing function from $E \times E \to \mu_n = \mathbb{F}^*_{q^k}/\mathbb{F}^{*n}_{q^k}$.

6.2.1 Preliminaries

Here we briefly discuss the basic tools needed for our scheme, namely

1. Bilinear Diffie-Hellman Problems
2. Identity-based encryption
3. Shamir's threshold scheme
4. Signcryption
5. Baek & Zheng's zero knowledge proof for the equality of two discrete logarithms

based on a bilinear map

We also cite relevant references for readers who would like more in-depth coverage of these interesting topics.

6.2.1.1 Bilinear Diffie-Hellman Problems

As [57] writes, bilinear maps were first used in cryptography to weaken systems rather than create them. In [56], the authors showed that “the discrete logarithm problem for an elliptic curve over a finite field $\mathbb{F}_q$ can be reduced to the discrete logarithm problem in some extension field $\mathbb{F}_q^k$.” For a particular class of curves called *supersingular* curves, this was a particularly devastating attack. Fortunately for elliptic curve cryptography, not all curves are supersingular.

The basic idea behind this attack was that if $Q = \ell P$, then

$$\widehat{e}(P, Q) = \widehat{e}(P, P)^{\ell}$$

so we can solve the resulting discrete logarithm problem (or Diffie-Hellman problem) in a different group instead, one where logarithms might be computed more easily. As such, we need to pick our groups E and μ_n such that the Decisional Diffie-Hellman Problem is difficult in μ_n and the following problems are difficult in $(E, \mu_n, \widehat{e})$:

Problem 6.2 (Computational Diffie-Hellman Problem). Suppose P is a generator of a large subgroup of E and $\alpha \in \mu_n = \widehat{e}(P, P)$. Given $Q, R \in E$ such that $Q = bP$ and $R = cP$, compute bc via $\beta = \alpha^b = \widehat{e}(P, Q)$ and $\gamma = \alpha^c = \widehat{e}(P, R)$.

Problem 6.3 (Decisional Diffie-Hellman Problem). Suppose P is a generator of a large subgroup of E and $\alpha \in \mu_n = \widehat{e}(P, P)$. Given $Q, R, S \in E$ such that $Q = bP$

and $R = cP$, determine which of the following is true via $\beta = \alpha^b = \widehat{e}(P, Q)$ and $\gamma = \alpha^c = \widehat{e}(P, R)$:

1. $S = bcP$ (so $\delta = \widehat{e}(P, S)$ is equal to α^{bc})

2. $S = dP$ for some d chosen uniformly at random from $\mathbb{Z}_n$ independently of b and c.

These problems are currently believed to be difficult if the size n of our groups is chosen large enough.

6.2.1.2 Identity-Based Encryption

In [69], Shamir proposed an interesting problem: he “asked for a public key encryption scheme in which the public key can be an arbitrary string.” [10] Shamir wished to simplify the management of digital certificates in e-mail and other systems; as the authors of [10] write, he wished to have a system such that “when Alice sends mail to Bob at `bob@hotmail.com` she simply encrypts her message using the public key string ‘`bob@hotmail.com`’,” thereby removing the need for interacting with any sort of management or external cryptographic infrastructure on a per-message basis. One of the most satisfying solutions to date comes from Boneh and Franklin. The Weil pairing was used in Boneh and Franklin’s scheme in [10]; readers are referred to this paper for more.

In [69] and [10], the protocols and schemes consist of four stages: *Setup*, *Extract*, *Encrypt*, and *Decrypt*. We follow the notation of [53] in the description of our scheme, though, with the four stages *Setup*, *Extraction*, *Signcryption*, and *Unsigncryption*. Before we get to the protocol, however, there is more background to cover.

6.2.1.3 Shamir's Threshold Scheme

In [68], Shamir developed a simple and elegant method to share a secret piece of information amongst n people such that no less than some threshold value t of them must cooperate to recover that secret. This scheme uses polynomial interpolation over a finite field; if we suppose that the secret piece of information s is encoded as some element of the field, we then construct a random polynomial f of degree $t-1$ such that $f(0) = s$ (so s is the constant term). If we give the pair $(i, f(i))$ to the ith person in our scheme, for $1 \leq i \leq n$, then via Lagrange interpolation any group of t people can first reconstruct the polynomial f and then evaluate $f(0)$ to recover s. Furthermore, because no group of $t-1$ or less people will suffice to recover the polynomial, this scheme is information-theoretically secure. For more, see the original paper [68]; [72] extends the idea of secret sharing to multipartite and compartmented schemes, while [11, 31, 42] and [43] discuss some ways to share secrets in various settings. To our knowledge, however, none of these include identity-based encryption and the next topic: signcryption.

6.2.1.4 Signcryption

In [81], the author put forth a new idea that combines the steps of digitally signing and encrypting a message—traditionally two separate procedures—that drastically reduces the computational and communication costs involved. Later work extended this idea to include other cryptographically desirable features such as non-repudiation, public-verifiability, and forward security (see [53]). Recently made into an international standard, signcryption has gained increasing popularity with researchers and implementers alike. For more information, see the original [81]; [82] demonstrates how to implement signcryption using rational points on elliptic curves over finite fields. Even more information, including an extensive bibliography, can be found online at `signcryption.org`.

6.2.1.5 Baek & Zheng's zero knowledge proof

In this section, $\mathcal{O}$ stands for the neutral element on an Edwards curve E instead of the organization in question. Per [4] and [53], the zero knowledge proof of membership for the language

$$L_{\mathtt{EDLog}^{E}_{P,\widetilde{P}}} \overset{\text{def}}{=} \{(x, \widetilde{x}) \in \mu_n \times \mu_n \mid \log_g x = \log_{\widetilde{g}} \widetilde{x}\}$$

(where $g = \widehat{e}(P, P)$ and $\widetilde{g} = \widehat{e}\left(\widetilde{P}, \widetilde{P}\right)$ for generators P and $\widetilde{P}$ of a large additive cyclic subgroup $E(K)$ of order $\#E = n$) ensure the robustness of our threshold decryption. Provided that the Decisional Diffie-Hellman problem is hard in μ_n and the Computational and Decisional Bilinear Diffie-Hellman Problems are difficult in $(E, \mu_n, \widehat{e})$, the basic idea is as follows: suppose both the Prover and the Verifier receive the tuple $\left(P, \widetilde{P}, g, \widetilde{g}\right)$ and the pair $(k, \widetilde{k}) \in L_{\mathtt{EDLog}^{\mu_n}_{P,\widetilde{P}}}$. Moreover, suppose the Prover knows a secret $S \in E \setminus \{\mathcal{O}\}$ such that $k = \widehat{e}(S, P)$ and $\widetilde{k} = \widehat{e}(S, \widetilde{P})$; then

1. The Prover chooses at random an element $R \in E \setminus \{\mathcal{O}\}$ computes $a = \widehat{e}(R, P)$ and $\widetilde{a} = \widehat{e}(R, \widetilde{P})$, and sends a and $\widetilde{a}$ to the Verifier.

2. The verifier picks $\gamma \in \mathbb{F}_q^*$ at random and sends it to the Prover.

3. The Prover computes $T = R + \gamma S$ and sends it to the Verifier. If (and only if)

the two equalities

$$ak^\gamma = \widehat{e}(T, P) \qquad \widetilde{ak}^\gamma = \widehat{e}(T, \widetilde{P})$$

hold, the Verifier believes that the Prover knows the secret S since

$$\widehat{e}(T, P) = \widehat{e}(R + \gamma S, P) = \widehat{e}(R, P)\widehat{e}(S, P)^\gamma = ak^\gamma$$

and

$$\widehat{e}(T, \widetilde{P}) = \widehat{e}(R + \gamma S, \widetilde{P}) = \widehat{e}(R, \widetilde{P})\widehat{e}(S, \widetilde{P})^\gamma = \widetilde{ak}^\gamma$$

For more, including how to adapt the above into a non-interactive zero knowledge proof, see [4].

6.2.2 The Proposed Compartmented Scheme

Suppose we have an organization $\mathcal{O}$ consisting of n people split into t compartments $\mathcal{C}_i$, each consisting of members $\mathcal{M}_{ij}$. In addition, we have a Private Key Generator ($\mathcal{P}$) who acts as the trusted authority and a sender Alice ($\mathcal{A}$) who wishes to send a message to the compartments $\mathcal{C}_i \subset \mathcal{O}$. There are four stages: *Setup*, *Extraction*, *Signcryption*, and *Unsigncryption*.

6.2.2.1 Setup

$\mathcal{P}$ first chooses our two groups of large prime order n: E and μ_n. $\mathcal{P}$ also picks a generator P of E and a number of hash functions:[29]

$$H_1 : \{0,1\}^* \to E$$

$$H_2 : \mu_n \to \{0,1\}^*$$

$$H_3 : \{0,1\}^* \times \mu_n \to \mathbb{F}_q^*$$

Finally, $\mathcal{P}$ chooses a secret master key $s \in \mathbb{F}_q^*$, computes $P_{pub} = sP$, and publishes the tuple

$$(E, \mu_n, n, \widehat{e}, P, P_{pub}, H_1, H_2, H_3, E, D)$$

where E and D are the encryption and decryption steps of some fast symmetric key cipher (like AES; see [21]).

6.2.2.2 Extraction

In what follows, given an ID (identifying information considered as a bit string), the public key $\mathcal{P}$ generates for that ID is $Q_{ID} = H_1(ID)$, the private signcryption key is $S_{ID} = s^{-1}Q_{ID}$, and the private decryption key is $D_{ID} = sQ_{ID}$.

Since $\mathcal{P}$ uses $ID_{\mathcal{O}}$ to compute $Q_{\mathcal{O}}$, $S_{\mathcal{O}}$, and $D_{\mathcal{O}}$ and wishes to pass information to each $\mathcal{C}_i$ in such a way that some cooperation is required to put $D_{\mathcal{O}}$ back together, she randomly picks $R_k \in E \setminus \{\mathcal{O}\}$, $k \in \{1, \ldots, t-1\}$, and constructs a function $f : \{0,1\}^* \to E$ via $f(u) = D_{\mathcal{O}} + \sum_1^{t-1} u^k R_k$ (treating u as the binary representation

[29] H_1 can be the one from [41]

of some positive integer). Then, for each $\mathcal{C}_i \subset \mathcal{O}$, $\mathcal{P}$:

1. Computes $D_i = f(ID_i)$, the private decryption key for $\mathcal{C}_i$
2. Computes $y_i = \widehat{e}(D_i, P)$, the public verification key for $\mathcal{C}_i$
3. For each $\mathcal{M}_{ij} \in \mathcal{C}_i$, $\mathcal{P}$:
    (a) Chooses a random $\mu_{ij} \in \mathbb{F}_q^*$
    (b) Privately sends $\mathcal{M}_{ij}$ the triple $(D_i, P_{ij}, y_{ij}) = \big(D_i, (1+\mu_{ij})D_i, y_i^{\mu_{ij}}\big)$
4. Finally, $\mathcal{P}$ publishes the table

$$\begin{aligned}\{(ID_i, y_i, \{(ID_{ij}, y_{ij})\}\} = {} & (ID_1, y_1, (ID_{1,1}, y_{1,1}), (ID_{1,2}, y_{1,2}), \ldots \\ & (ID_2, y_2, (ID_{2,1}, y_{2,1}), (ID_{2,2}, y_{2,2}), \ldots \\ & \vdots\end{aligned}$$

6.2.2.3 Signcryption

To send the message m to $\mathcal{O}$, Alice computes the signcrypted text (c, r, S) as follows:

1. She chooses a random $x \in \mathbb{F}_q^*$
2. $k_1 = \widehat{e}(P, Q_{\mathcal{A}})^x$
3. $k_2 = H_2(\widehat{e}(Q_{\mathcal{A}}, Q_{\mathcal{O}})^x)$
4. $c = E_{k_2}(m)$
5. $r = H_3(c, k_1)$

6. $S = (x - r)S_{\mathcal{A}}$

6.2.2.4 Unsigncryption

After at least one member $\mathcal{M}_{ij}$ from each of the t compartments $\mathcal{C}_i$ assemble, they first verify Alice's signature; then each $\mathcal{M}_{ij}$ individually

1. Computes $k_1' = \widehat{e}(S, P_{pub})\widehat{e}(Q_{\mathcal{A}}, P)^r$

2. Accepts Alice's signature if and only if $r = H_3(c, k_1')$

Next, each $\mathcal{M}_{ij}$ picks two random points $B_{ij}, T_{ij} \in E$ and uses B_{ij} to certify that they belong to $\mathcal{C}_i$ and T_{ij} to certify their decryption share. While the latter is accomplished in exactly the same manner as in [53], $\mathcal{M}_{ij}$ does the former as follows:

3. Construct credentials κ_{ij} using B_{ij}, where

$$\kappa_{ij} = (\widetilde{P}_{ij}, z_{ij}) = (P_{ij} + B_{ij}, y_{ij}\widehat{e}(B_{ij}, P))$$

4. Send credentials κ_{ij} to each of the other $\mathcal{M}_{k\ell}$

5. Check each of $\mathcal{M}_{k\ell}$'s credentials by testing whether

$$y_k = \frac{\widehat{e}(\widetilde{P}_{k\ell}, P)}{z_{k\ell}}$$

Once everyone's credentials are established, the rest of unsigncryption continues as in [53].

6.2.3 Analysis of Scheme

We discuss the effects to correctness, security, and efficiency of the changes we have made to [53]'s original scheme. As such, our analysis is based on that of [53], especially where it makes use of [4]'s zero knowledge proof of membership.

6.2.3.1 Correctness

Observe that

$$\begin{aligned}\frac{\widehat{e}(\widetilde{P}_{ij}, P)}{z_{ij}} &= \frac{\widehat{e}(P_{ij}, P)\widehat{E}(B_{ij}, P)}{y_{ij}\widehat{e}(B_{ij}, P)} \\ &= \frac{\widehat{e}\left((1+\mu_{ij})D_k, P\right)}{y_k^{\mu_{ij}}} \\ &= \frac{\widehat{e}(D_k, P)\widehat{e}(D_k, P)^{\mu_{ij}}}{y_k^{\mu_{ij}}} \\ &= \widehat{e}(D_k, P) \\ &= y_k\end{aligned}$$

So κ_{ij} does indeed certify that $\mathcal{M}_{ij}$ belongs to and can speak for the compartment $\mathcal{C}_k$. The correctness of the rest of our scheme can be proven in exactly the same manner as [53].

6.2.3.2 Security

Because the signcryption process in our scheme is the same as in [53] (which in turn is the same as in [17]), our scheme has the same existential unforgeability against chosen plaintext attacks in the random oracle model as those schemes, provided that the Computational Bilinear Diffie-Hellman Problem is difficult in the groups and pairing underlying the implementation of our scheme.

What's more, our scheme doesn't change the level of confidentiality either; assuming the Decisional Bilinear Diffie-Hellman Problem is hard in $(E, \mu_n, \widehat{e})$, our scheme enjoys the same indistinguishability against adaptive chosen ciphertext attacks in the random oracle model. During unsigncryption, no less than t cooperating members of different compartments suffice to recover the key k_2 (and hence the message). Giving different randomly obfuscated versions of the same information to members of the same compartment does nothing to lessen this fact. Recovery of $D_{\mathcal{O}}$ is also computationally infeasible due to the difficulty of inverting the pairing $\widehat{e}$. Finally, the use of Baek and Zheng's zero knowledge proof ensures that each member participating in unsigncryption is protected against the possibility of dishonesty from any of the others.

The public verifiability of our extended scheme remains intact, since any third party can verify the signature via the first two steps of the *Unsigncryption* stage.

We also still keep forward security, since it remains difficult to compute k_2' without $D_{\mathcal{O}}$, even if $S_{\mathcal{A}}$ is leaked.

6.2.3.3 Efficiency

With a slight modification to [53]'s notation, let T_p, T_m, and T_e be the computing time required for calculating a pairing, point multiplication, and exponentiation, respectively. Note that our scheme still requires $2T_p + T_m + 2T_e$ for signcryption and $(2t+4)T_p + T_m + (3t-1)T_e$ for $\mathcal{M}_{ij}$, just like the original scheme. The main bottleneck in this scheme is the random point choices performed by $\mathcal{P}$; if we assume that $\mathcal{P}$ has a fast pseudorandom number generator, then the time this takes is essentially $(2n+1)T_m$, just like in [53].

The efficiency picture can be improved, though; instead of having $\mathcal{P}$ choose each $\mathcal{M}_{ij}$'s point, it could instead choose t points and send them to t secondary generators

P_i, one for each compartment. These secondary generators can then randomize those points and distribute the relevant information to the members of their respective compartments. Though this doesn't reduce the work involved (and it requires having more trusted authorities, or rather semi-trusted authorities), it does allow our scheme to parallelize one of its major, one-time steps. Hence our scheme lends itself better to implementation using modern computing methods (i.e. parallel computation) than does [53].

6.2.4 Conclusion

In this section we demonstrated how a small modification to Li, Xin, and Hu's scheme ([53]) extends it into a compartmented scheme, allowing a sender to address a message to an organization $\mathcal{O}$ and requiring different compartments $\mathcal{C}_i \subset \mathcal{O}$ to cooperate for the message's recovery. In doing so, we do not lose any of the security or efficiency features of [53]'s scheme—in fact, we can even parallelize one of the main stages. To our knowledge, this scheme is the first that combines identity-based encryption, Shamir's secret sharing, and signcryption into a compartmented sharing scheme that can be implemented with available algorithms and software.

This scheme incorporates a naturally parallelizable step, and is likewise naturally applicable to modern situations. For instance, this scheme could very easily be used in cloud computing to synchronize information passed to different groups or clusters from a single host. As another example, one could use this scheme for authenticated and signcrypted communication in a business setting; the shared secret could be an expected return message to acknowledge receipt of an important document or the scheduling of an important meeting. There is still room for future work. We hope to investigate deeper into questions such as increasing the efficiency of our scheme or reducing the reliance upon the trusted private key generator $\mathcal{P}$.

CHAPTER 7: `E2C2`: A C++11 LIBRARY FOR EDWARDS ECC

In order to explore the theory and implementation of Edwards curves for Elliptic Curve Cryptography, I've created a modern C++11 software library called `e2c2`. In this chapter, we'll discuss the design and rationale behind `e2c2`, its organization, and a few examples of its functionality.

7.1 Rationale and Design Choices

`e2c2` was designed to be a software library that was simple enough to use but close enough to a practical implementation that software engineers could use it as a template or a stand-in for a practical cryptographic library. As such, it strikes a balance between usability and speed and between clarity of exposition and being closer to "production-level code."

7.1.1 C++11

Probably the most obvious design choice was picking C++, specifically the newest standard C++11, for the programming language. Though not as low-level as C, C++ still gives us decent enough control over the underlying specifics of the machine. Since it's a compiled language with a long history of implementation, it creates very fast binaries. Furthermore, C++ is available on a number of platforms, so portability is less of an issue. These characteristics combine to make C++ a viable choice for a cryptography implementation seeking to bridge the gap between mathematical theory and programming practice.

Even more helpful was the expressiveness of the new C++ language standard, C++11[28]. This new standard adds interesting and useful constructs to the language, many of which are used in our library. These improvements—like type inference, lambda functions and expressions, and range-based for-loops—make C++ an easier language in which to implement complex mathematical and cryptographic concepts. What's more, this new level of expressiveness comes at no obvious loss of execution speed.

7.1.2 NTL and GMP

The next important choice was choosing to use software libraries that abstract away the details of arbitrary precision integers and foundational number theory concepts. The GNU Multiple Precision Arithmetic Library, better known as GNU MP or GMP, is a free and open-source library for arbitrary-precision arithmetic with integers and rational numbers.[35] Though a true full-blown production-level cryptographic library might choose to implement its own big integer arithmetic for ultimate control, speed, and safety, GMP is tried and tested enough after over 15 years of development to be included here in our library that's trying to be a mix of proof-of-concept and implementation guide.

I used Victor Shoup's Number Theory Library—NTL for short—for the same reasons, only more so.[70] NTL is a C++ library for doing number theory; to quote its website,

NTL is a high-performance, portable C++ library providing data struc-

> tures and algorithms for manipulating signed, arbitrary length integers, and for vectors, matrices, and polynomials over the integers and over finite fields. NTL provides high quality implementations of state-of-the-art algorithms for:
>
> - arbitrary length integer arithmetic and arbitrary precision floating point arithmetic;
> - polynomial arithmetic over the integers and finite fields including basic arithmetic, polynomial factorization, irreducibility testing, computation of minimal polynomials, traces, norms, and more;
> - lattice basis reduction, including very robust and fast implementations of Schnorr-Euchner, block Korkin-Zolotarev reduction, and the new Schnorr-Horner pruning heuristic for block Korkin-Zolotarev;
> - basic linear algebra over the integers, finite fields, and arbitrary precision floating point numbers.
>
> NTL's polynomial arithmetic is one of the fastest available anywhere, and has been used to set "world records" for polynomial factorization and determining orders of elliptic curves.

NTL does most of the heavy lifting when it comes to finite field computations in `e2c2`; it is only linked against GMP at compile time to take advantage of some of its code for enhanced performance. To quote the project's website again, the main reason for choosing NTL is because "It provides a good environment for easily and quickly implementing new number-theoretic algorithms, without sacrificing performance."

In some regards, `e2c2` can be seen as an extension of NTL. It uses portions of NTL to implement Edwards curves in a way that blends well with the rest of NTL, and the line between where NTL stops and `e2c2` starts need not be of much concern to users of this library.[30]

7.1.3 Template Specialization Instead of Inheritance

The last design choice we'll discuss will probably only be of concern to aficionados of C++ (or perhaps another object-oriented language). Though curves and points are implemented as classes in `e2c2`, the library is written using template specialization instead of class inheritance. This gave me the ability to use the illusion of inheritance in constructing `e2c2`, sharing common functionality between different types of curves or different types of points, without the added runtime penalties that can be associated with virtual function lookup. Moreover, C++ templates are expanded at compile time (and much can be inlined by a compiler), thereby only charging the programmer for what they use. Another added bonus of using template specialization over inheritance is ease of portability and use; `e2c2` consists entirely of C++ header files, four specifying the implementation and one single `e2c2.h` interface header to be included in projects. This means that `e2c2` is quite compact, and it is simple to share or extend the sourcecode. It's compact enough that the entire source for `e2c2` and examples of its usage is included in Appendix 7.4.4.

Readers interested in learning more about C++ templates should consult [76].

[30]Especially if C++ projects using `e2c2` prefaces any major portions of code with the directives "`using namespace NTL`" and "`using namespace e2c2`," as in the examples we'll show later.

7.2 Curves and Points

We now get into the actual details of `e2c2`'s code. First up is a description of its fundamental objects: curves and points.

7.2.1 `curves.h`

The code for curves in contained in the header file `curves.h`. Curves are implemented as templates based on two parameters: an element type (one of NTL's "ZZ_pE" or "GF2E" datatypes) and a C++ enumeration[31] called "CurveID" (to help distinguish between two types of curves that have the same base element type). Each curve type has two field elements called c and d as members; these correspond to the c and d of Edwards curves, a and d in twisted Edwards curves, and d_1 and d_2 in binary Edwards curves. They also have an NTL ZZ m that gives the cardinality of the curve (i.e., the number of rational points on the curve over the field in question).

Besides the usual C++ member functions (i.e. constructors and destructors), curves have four member functions. Through C++'s operator overloading, curves can be compared for equality or inequality with the usual operators (`==` and `!=`). They can also tell you their cardinality via calls to `cardinality()` and information about them can be printed to an output stream via the `<<` operator.

Each curve specialization—OddCurve, TwistedCurve, and BinaryCurve—provides four things. The first is an alias typedef from the verbose template name to the shorted curve name for ease of use. Then each specialization provides three functions that are used in their member functions:

[31] Actually it's a C++11 strongly typed enumeration for added typesafety peace of mind.

- `getName`, when given a curve as an argument, returns a human readable string describing the type of curve; this is used when the curve information is printed to an output stream

- `parametersValid`, when given a curve as an argument, returns a boolean value which checks whether the curve parameters c and d are valid; this is used in curve construction to ensure that the curve being built matches the requirements given in the various papers about it

- `curveEquation`, when given a curve and an x and y coordinate, returns a boolean value that specifies whether the two coordinates describe a point lying on the curve or not; this is used when points are constructed (to be described shortly)

To construct a curve, one first builds the appropriate field via NTL, then calls the specific Curve constructor with c, d, and m specified. For an example, see `curves_test.cc` (described below).

7.2.2 `points.h`

As you might expect, points are a little more complicated. There are two base class templates for points: "Affine" and "Projective." Each point has a curve to which it is assigned; in addition, Affine points have two coordinates (x and y), while Projective points have three coordinates (x, y, and z). Though Affine points can only interact with Affine points at this time, and likewise Projective, there is a copy constructor from Affine to Projective; i.e. if `a` is a BinaryAff, one can write the following:

```
auto p = BinaryProj(a);
```

to create a projective version of `a`.[32]

Beyond the obvious difference in the number of coordinates, Affine and Projective points have all the same member functions. Points can be compared for equality and inequality, tested for whether they're the identity element, added or subtracted (both with `+`, `-` and `+=`, `-=`), doubled, and multiplied by a scalar. Scalar multiplication is implemented with Montgomery's ladder, and incorporates the suggested "wrap-around" to counteract Brumley and Tuveri's timing attack from [14].

An example of point functionality is given in `points_test.cc`, which we will soon discuss.

7.3 Utilities and Subroutines

`e2c2` has two scaffolding files that support the real work, the first of which is the header `mol.h`.

7.3.1 `mol.h`

This header file is named for the authors of [62]. In it there are a number of functions and routines that implement the ideas from that paper. Some are only used by other functions in the same file, but probably most important is `mol_alg_1`. If fed a long n, representing the degree of the extension field $\mathbb{F}_{2^n}$, and two elements of this field a_2 and a_6 that specify a binary elliptic curve in Weierstrass form, this algorithm computes the d_1 value that specifies the birationally equivalent binary Edwards curve.[33] This function is in turn used to implement other functions in

[32]This copy constructor has been declared `explicit`, so there shouldn't be any surprises about when this conversion takes place.

[33]d_1 is called c in `e2c2`.

`mol.h`.

`mol_alg_1` is definitely too low-level for typical use, while other functions in this file are probably more user friendly; programmers interested in the `e2c2` library will find most useful

- `from_weierstrass`, which takes n, m, a_2, and a_6 as arguments and returns a BinaryCurve
- `mol_bm_aff`, which implements the birational map from [62] for Affine points
- `mol_bm_proj`, which implements the birational map from [62] for Projective points

7.3.2 `utilities.h`

The other utility file is called, rather aptly, `utilities.h`. This file contains a routine to set a parameter given a string in hex or not; this routine, called `set_parameter`, is helpful in constructing the curves specified in various standards like [30]. `utilities.h` also specifies the various C++ exceptions created to signal fatal error conditions to users of `e2c2`:

- `InvalidParametersException`, which is thrown when one tries to construct a curve with parameters that are invalid
- `DifferentCurvesException`, which is thrown when attempting to perform an operation involving two points from different curves
- `NotImplementedException`, which is left over from `e2c2`'s development; it was used (and can be used again, as development of this library continues) to indi-

cate that the edge of current implementation had been reached but the functionality in question was planned for future work

7.4 Examples

Probably the best exposition we can have of `e2c2` is to see it in action; as such, we present four example C++ files that can be compiled and run to demonstrate the library's functionality and usage. All examples have been compiled and tested with the following command and options:[34]

Listing 7.5: Compiler Options

```
g++-4.7 -Wall -Wextra -Weffc++ -pedantic -O3 -m64 -std=c++11 -lntl -lgmp
```

Specifically, all this code was compiled with the GNU C++ compiler version 4.7 (see [73]), though any compiler and standard library that implements most of the C++11 standard should work.

7.4.1 `curves_test.cc`

The first example file is `curves_test.cc`. In this program, there are three different subroutines that can all be called by the main routine, depending on user input. Each specifies a type of Edwards curve—odd, binary, or twisted—to construct and output some information about. The binary example is slightly more involved, since it builds a curve from the Weierstrass equivalent and later intentionally crashes by trying to make a curve with invalid parameters.

[34]The `-Weffc++` option tells the compiler to warn about C++ code that doesn't meet the high standards set by [58].

7.4.2 `points_test.cc`

The next example file is similar; it shows the creation and usage of points, both affine and projective, over all three types of Edwards curves. Again, the user can specify a specific type of curve to work over; once that is done, the program

1. builds the appropriate curve
2. constructs an affine identity element on that curve and outputs it
3. constructs two projective points, one of which is the neutral element and one which is not
4. demonstrates point addition, negation, and scalar multiplication with these points

7.4.3 `key_demo.cc`

The third example file is a little more involved. This program gives a short Diffie-Hellman key exchange demonstration. After some setup, we join our friends Alice and Bob as they try to construct a shared key so as to communicate in private. After deciding (in public) on a base point P, Alice and Bob pick private random scalars a and b, respectively. Then Alice computes and publishes aP while Bob does the same with bP. Their shared key is $a(bP) = b(aP)$; the code double-checks that all calculations went according to plan.

Here is the output of an example run of this program, slightly reformatted:

Listing 7.6: Output of key_demo.cc

```
Alice and Bob wish to communicate in private, so they need a shared secret key.


Alice first picks a secret random number a =
10582230386582042969237721342986301254891720029263
with the same number of bits as m (the size of our group)...

...then sends Bob the point pA = a * P (the generator) =
    (0x325292b6786653bc201777d7ce8f78ad2b7293bf5 :
     0xadc4e7fb280edc5b490bf5a0200716435af252064 :
     0x1)

Bob also picks a secret random number b =
6842965894516648772563843478507619750018805216122
Then he sends Alice the point bP = b * P =
    (0x0d48849159e4d4af5fbfe0d25841ebf5da8200793 :
     0xac71cbbf9c6b620a45a52ec50cdd4c52f10e07a05 :
     0x1)

Then Alice takes a and multiplies it by bP to get key_a =
    (0x9b003af9bf9ff81875b26c8873ac866413199742 :
     0x77bb1c163e1fdacd2eefba7941a9dc5b7de4838c2 :
     0x1)
Similarly, Bob calculates key_b = b * aP =
    (0x9b003af9bf9ff81875b26c8873ac866413199742 :
    0x77bb1c163e1fdacd2eefba7941a9dc5b7de4838c2 :
    0x1)

```

```
Are these in fact the same key?

Yes!

So now they share a secret key, and can communicate securely.
```

7.4.4 ecoh_echo.cc

As a final example, we draw the reader's attention to the sourcecode listing in the first section of Chapter 6. In it we provide `ecoh_echo.cc`, a reference implementation of our password based key derivation function described in that section. It is probably the best non-trivial example of using `e2c2` for cryptographic exploration and implementation.

`e2c2` Source

Listing 7.7: curves.h

```
/**
 * @file curves.h
 * @brief Edwards Curves over finite fields of prime characteristic
 * @author Graham Enos
 *
 * This file contains the interface and implementation (since this base "class"
 * is really a template) of Edwards Curves over finite fields of prime
 * characteristic, viz @f$ \mathbf{F}_{p^n} @f$ in C++
 */

#ifndef _CURVES_H
#define _CURVES_H

#include <iostream>      // Readable output
#include <NTL/ZZ.h>      // Arbitrarily large integers
#include <NTL/ZZ_pE.h>   // Field elements from @f$ \mathbf{F}_{p^n} @f$
#include <NTL/GF2E.h>    // Field elements from @f$ \mathbf{F}_{2^n} @f$
#include "utilities.h"   // Utilities header for e2c2 project


/// Namespace for our library
namespace e2c2 {

    //------- Class Skeletons -------//

    /**
     * @p CurveID
     * @brief quick typesafe identifiers for curves
 *
 * @details Used in curve definitions, made an enum class for typesafety
 * and speed; used to differentiate curves that are defined similarly
 */
enum class CurveID {
    Odd,
    Binary,
    Twisted
};

/**
 * @p Curve
 * @brief Base "class" (really class template) for Edwards Curves
 *
 * @tparam Elt        field element
 * @tparam ID         id of curve
 *
 * @details Collects all relevant information about the curve and provides
 * basic functionality
 */
template <class Elt, CurveID ID>
class Curve {
public:
    /// Parameter #1 of curve; @f$ c @f$ in papers on odd curves,
    /// @f$ d_1 @f$ in binary papers, @f$ a @f$ in twisted papers,
    Elt c;

    /// Parameter #2 of curve; @f$ d @f$ in papers on odd and twisted
    /// curves, @f$ d_2 @f$ in binary papers
    Elt d;

    /// Cardinality of curve
    NTL::ZZ m;

    /// Default constructor
    Curve() : c(), d(), m() {}

    /// Destructor
    ~Curve() {}

    /// Constructor given full tuple as input
    Curve(const Elt& c, const Elt& d, const NTL::ZZ& m) :
            c(c), d(d), m(m) {
        if (!parametersValid(*this))
            throw InvalidParametersException();
    }

    /// Equality test
    bool operator==(const Curve& that) const {
        return (c == that.c) && (d == that.d) && (m == that.m);
    }

    /// Inequality test
    bool operator!=(const Curve& that) const {
        return not(*this == that);
    }

    /// Number of rational points on curve over field
    NTL::ZZ cardinality() const {
        return m;
    }

    /// Output of curve's information
    friend std::ostream& operator<<(std::ostream& out,
            const Curve& curve) {
        out << getName(curve) << std::endl
            << "Curve parameter #1: " << curve.c << std::endl
            << "Curve parameter #2: " << curve.d << std::endl
            << "Cardinality: " << curve.m;
        return out;
    }
};


//------- Edwards Curves -------//

/**
 * @var OddCurve
 * @brief Edwards Curve over finite field with odd characteristic
 *
 */
using OddCurve = Curve<NTL::ZZ_pE, CurveID::Odd>;


/**
 * @p getName(const OddCurve&)
 * @brief output of name
 */
inline std::string getName(const OddCurve&) {
    return std::string("Edwards Curve over odd field");
}

/**
 * @p parametersValid(const OddCurve& curve)
 * @brief Parameter validation
 */
inline bool parametersValid(const OddCurve& curve) {
    return (curve.c != 0 && NTL::sqr(curve.c) + curve.c != curve.d);
}

/**
 * @p curveEquation(const OddCurve& curve, const NTL::ZZ_pE& x,
 *      const NTL::ZZ_pE& y)
 * @brief Validation of point coordinates
 */
inline bool curveEquation(const OddCurve& curve, const NTL::ZZ_pE& x,
        const NTL::ZZ_pE& y) {
    auto xx = NTL::sqr(x), yy = NTL::sqr(y);
    return (xx + yy == NTL::sqr(curve.c) * (1 + curve.d * xx * yy));
}


/**
 * @var BinaryCurve
 * @brief Edwards Curve over finite field of characteristic two
 */
using BinaryCurve = Curve<NTL::GF2E, CurveID::Binary>;

/**
 * @p getName(const BinaryCurve&)
 * @brief output of name
 */
inline std::string getName(const BinaryCurve&) {
    return "Binary Edwards Curve";
}

/**
 * @p parametersValid(const BinaryCurve& curve)
 * @brief Parameter validation
 */
inline bool parametersValid(const BinaryCurve& curve) {
    return (curve.c * curve.d * (1 - curve.d * NTL::power(curve.c, 4)) !=
            0);
}

/**
 * @p curveEquation(const BinaryCurve& curve, const NTL::GF2E& x,
 *     const NTL::GF2E& y)
 * @brief Validation of point coordinates
 */
inline bool curveEquation(const BinaryCurve& curve, const NTL::GF2E& x,
        const NTL::GF2E& y) {
    auto xx = NTL::sqr(x), yy = NTL::sqr(y);
    return (curve.c * (x + y) + curve.d * (xx + yy)
            == (x + xx) * (y + yy));
}


/**
 * @var TwistedCurve
 * @brief Twisted Edwards Curve over a non-binary field
 */
using TwistedCurve = Curve<NTL::ZZ_pE, CurveID::Twisted>;

/**
 * @p getName(const TwistedCurve&)
 * @brief output of name
 */
inline std::string getName(const TwistedCurve&) {
    return "Twisted Edwards Curve";
}
```

```
    /**
     * @p parametersValid(const TwistedCurve& curve)
     * @brief Parameter validation
     */
    inline bool parametersValid(const TwistedCurve& curve) {
        return (curve.c != curve.d) || ((curve.c == 0) || (curve.d == 0));
    }

    /**
     * @p curveEquation(const TwistedCurve& curve, const NTL::ZZ_pE& x,
     *      const NTL::ZZ_pE& y)
     * @brief Validation of point coordinates
     */
    inline bool curveEquation(const TwistedCurve& curve, const NTL::ZZ_pE& x,
            const NTL::ZZ_pE& y) {
        auto xx = NTL::sqr(x), yy = NTL::sqr(y);
        return (curve.c * xx + yy == 1 + curve.d * xx * yy);
    }
}

#endif // _CURVES_H
```

Listing 7.8: points.h

```
/**
 * @file points.h
 * @brief Affine points on Edwards Curves
 *
 * This file contains the interface and implementation (since this base "class"
 * is really a template) of affine and projective points on Edwards Curves.
 */

#ifndef _POINTS_H
#define _POINTS_H

#include <iostream>      // Readable output
#include <NTL/ZZ.h>      // Arbitrarily large integers
#include <NTL/ZZ_pE.h>   // Field elements from @f$ \mathbf{F}_{p^n} @f$
#include <NTL/GF2E.h>    // Field elements from @f$ \mathbf{F}_{2^n} @f$
#include "utilities.h"   // Utilities header for e2c2 project
#include "curves.h"      // e2c2 curve interface and implementation
#include "mol.h"         // Birational map and utilities from MOL paper
#include "utilities.h"   // e2c2 utilities


/// Namespace for our library
namespace e2c2 {
    //------- Helper Function -------//

    /**
     * @p counterBTTiming(const NTL::ZZ& k, const NTL::ZZ& m)
     * @brief This function counteracts the timing attack outlined in Brumley &
     * Tuveri's paper
     */
```

```
inline NTL::ZZ counterBTTiming(const NTL::ZZ& k, const NTL::ZZ& m) {
   if (NTL::NumBits(k + m) == NTL::NumBits(m))
      return k + 2 * m;
   else
      return k + m;
}


//------- Class Skeletons -------//

/**
 * @class Affine
 * @brief Class skeleton for affine points on Edwards Curves
 *
 * Collects relevant information and functionality for affine points on
 * Edwards Curves
 */
template <class Elt, class Curve>
class Affine {
public:
   /// x-coordinate
   Elt x;

   /// y-coordinate
   Elt y;

   /// curve to which this point belongs
   Curve curve;

   /// Default constructor
   Affine() : x(), y(), curve() {}

   /// Destructor
   ~Affine() {}

   /// Constructor given all information
   Affine(const Elt& x, const Elt& y, const Curve& curve) :
         x(x), y(y), curve(curve) {
      if (!curveEquation(curve, x, y))
         throw InvalidParametersException();
   }

   /// Constructor if given just a curve (left to specific classes)
   Affine(const Curve& curve) : x(), y(), curve(curve) {
      *this = aff_id(curve);
   }

   /// Checking whether we have the neutral element
   bool isID() const {
      return *this == Affine(this->curve);
   }

   /// Two affine points are equal if all relevant info is the same...
   bool operator==(const Affine& that) const {
        return (x == that.x) && (y == that.y) && (curve == that.curve);
    }

    /// ...and are not equal otherwise
    bool operator !=(const Affine& that) const {
        return !(*this == that);
    }

    /// Assignment by addition; left to fleshed-out classes
    Affine& operator+=(const Affine& that) {
        if (this->curve != that.curve)
            throw DifferentCurvesException();

        *this = aff_add(*this, that);
        return *this;
    }

    /// Negation of a point; left to full class
    Affine operator-() const {
        return aff_neg(*this);
    }

    /// Assignment by subtraction; makes use of += and -
    Affine& operator-=(const Affine& that) {
        return *this += -that;
    }

    /// Addition via +=
    Affine operator+(const Affine& that) const {
        return Affine(*this) += that;
    }

    /// Subtraction via -=
    Affine operator-(const Affine& that) const {
        return Affine(*this) -= that;
    }

    /// Point doubling; left to full class
    Affine pointDouble() const {
        return aff_double(*this);
    }

    /// Montgomery Ladder for scalar multiplication
    Affine montgomery(const NTL::ZZ& k) const {
        // Work with positive scalars
        if (k < 0)
            return (-(*this)).montgomery(-k);

        // Counteract Brumley & Tuveri's timing attack
        NTL::ZZ kk = counterBTTiming(k, curve.cardinality());

        Affine aTmp(curve); // identity element
        Affine bTmp(*this);
        for (auto i = NumBits(kk) - 1; i >= 0; i--) {
            if (bit(k, i)) {
                aTmp += bTmp;
                bTmp = bTmp.pointDouble();
            } else {
                bTmp += aTmp;
                aTmp = aTmp.pointDouble();
            }
        }
        return aTmp;
    }

    /// Assignment by scalar multiplication (using Montgomery Ladder)
    Affine& operator*=(const NTL::ZZ& k) {
        return *this = montgomery(k);
    }

    /// Assignment by scalar multiplication (using Montgomery Ladder)
    template <class N>
    Affine& operator*=(const N& k) {
        return *this = montgomery(NTL::to_ZZ(k));
    }

    /// Scalar multiplication (using Montgomery Ladder), via *= (e.g. k *
    /// point)
    friend Affine operator*(const NTL::ZZ& k, const Affine& point) {
        return Affine(point) *= k;
    }

    /// Scalar multiplication (using Montgomery Ladder), via *= (e.g. k *
    /// point)
    template <class N>
    friend Affine operator*(const N& k, const Affine& point) {
        return Affine(point) *= NTL::to_ZZ(k);
    }

    /// Output
    friend std::ostream& operator<<(std::ostream& out, const Affine&
                                    point) {
        return (out <<"(" << point.x << ", " << point.y << ")");
    }
};


/**
 * @class Projective
 * @brief Class skeleton for projective points on Edwards Curves
 *
 * Collects relevant information and functionality for projective points on
 * Edwards Curves
 */
template <class Elt, class Curve>
class Projective {
public:
    /// x-coordinate
    Elt x;

    /// y-coordinate
    Elt y;

    /// z-coordinate
    Elt z;

    /// curve to which this point belongs
    Curve curve;

    /// Default constructor
    Projective() : x(), y(), z(), curve() {}

    /// Destructor
    ~Projective() {}

    /// Constructor given all information
    Projective(const Elt& x, const Elt& y, const Elt& z,
            const Curve& curve) : x(x), y(y), z(z), curve(curve) {
        if (!curveEquation(curve, x / z, y / z))
            throw InvalidParametersException();
    }

    /// Constructor if given just a curve (left to specific class)
    Projective(const Curve& curve) : x(), y(), z(), curve(curve) {
        *this = proj_id(curve);
    }

    /// Constructor from an affine point of the same type
    explicit Projective(const Affine<Elt, Curve>& a) : x(a.x), y(a.y), z(),
            curve(a.curve) {
        z = 1;
        if (!curveEquation(curve, x / z, y / z))
            throw InvalidParametersException();
    }

    /// Equivalence class representative: z = 1
    Projective equivalenceClassRep() const {
        Elt one;
        one = 1;
        return Projective(x / z, y / z, one, curve);
    }

    /// Checking whether we have the neutral element
    bool isID() const {
        return *this == Projective(this->curve);
    }

    /// Two projective points are equal iff all relevant info is the
    /// same...
    bool operator==(const Projective& that) const {
        return (x / z == that.x / that.z) && (y / z == that.y / that.z) &&
            (curve == that.curve);
    }

    /// ...and are not equal otherwise
    bool operator !=(const Projective& that) const {
        return !(*this == that);
    }

    /// Assignment by addition; left to fleshed-out class specificities
    Projective& operator+=(const Projective& that) {
        if (this->curve != that.curve)
            throw DifferentCurvesException();

        *this = proj_add(*this, that);
        return *this;
    }

    /// Negation of a point; left to class specificities
    Projective operator-() const {
        return proj_neg(*this);
    }

    /// Assignment by subtraction; makes use of += and -
    Projective& operator-=(const Projective& that) {
        return *this += -that;
    }

    /// Addition via +=
    Projective operator+(const Projective& that) const {
        return Projective(*this) += that;
    }

    /// Subtraction via -=
    Projective operator-(const Projective& that) const {
        return Projective(*this) -= that;
    }

    /// Point doubling; left to class specifics
    Projective pointDouble() const {
        return proj_double(*this);
    }

    /// Montgomery Ladder for scalar multiplication
    Projective montgomery(const NTL::ZZ& k) const {
        // Work with positive scalars
        if (k < 0)
            return (-(*this)).montgomery(-k);

        // Counteract Brumley & Tuveri's timing attack
        NTL::ZZ kk = counterBTTiming(k, curve.cardinality());

        Projective aTmp(curve); // identity element
        Projective bTmp(*this);
        for (auto i = NumBits(kk) - 1; i >= 0; i--) {
            if (bit(k, i)) {
                aTmp += bTmp;
                bTmp = bTmp.pointDouble();
            } else {
                bTmp += aTmp;
                aTmp = aTmp.pointDouble();
            }
        }
        return aTmp;
    }

    /// Assignment by scalar multiplication (using Montgomery Ladder)
    Projective& operator*=(const NTL::ZZ& k) {
        return *this = montgomery(k);
    }

    /// Assignment by scalar multiplication (using Montgomery Ladder)
    template <class N>
    Projective& operator*=(const N& k) {
        return *this = montgomery(NTL::to_ZZ(k));
    }

    /// Scalar multiplication (using Montgomery Ladder), via *= (e.g. k *
    /// point)
    friend Projective operator*(const NTL::ZZ& k,
        const Projective& point) {
        return Projective(point) *= k;
    }

    /// Scalar multiplication (using Montgomery Ladder), via *= (e.g. k *
    /// point)
    template <class N>
    friend Projective operator*(const N& k, const Projective& point) {
        return Projective(point) *= NTL::to_ZZ(k);
    }


    /// Output
    friend std::ostream& operator<<(std::ostream& out,
            const Projective& point) {
        Projective tmp = point.equivalenceClassRep();
        return (out << "(" << tmp.x << " : " << tmp.y << " : " << tmp.z <<
                ")" );
    }

};


//------- Affine Specialization Functions -------//

/**
 * @var OddAff
 * @brief Affine points on an odd curve
 *
 * This typedef builds the four functions necessary to flesh out our Affine
```

```
 * template for odd affine points.
 */
using OddAff = Affine<NTL::ZZ_pE, OddCurve>;

/**
 * @p aff_id(const OddCurve& curve)
 * @brief Affine neutral element on odd curve.
 */
inline OddAff aff_id(const OddCurve& curve) {
    return OddAff(NTL::ZZ_pE::zero(), curve.c, curve);
}

/**
 * @p aff_add(const OddAff& a1, const OddAff& a2)
 * @brief Affine addition for points on an odd curve.
 */
inline OddAff aff_add(const OddAff& a1, const OddAff& a2) {
    NTL::ZZ_pE w, num_x, num_y, den_x, den_y;
    w = a1.curve.d * a1.x * a2.x * a1.y * a2.y;
    num_x = a1.x * a2.y + a1.y * a2.x;
    num_y = a1.y * a2.y - a1.x * a2.x;
    den_x = a1.curve.c * (1 + w);
    den_y = a1.curve.c * (1 - w);

    return OddAff(num_x / den_x,
                  num_y / den_y,
                  a1.curve);
}

/**
 * @p aff_neg(const OddAff& a)
 * @brief Negation of an affine point on an odd curve.
 */
inline OddAff aff_neg(const OddAff& a) {
    return OddAff(-a.x, a.y, a.curve);
}

/**
 * @p aff_double(const OddAff& a)
 * @brief Affine point doubling on odd curve.
 */
inline OddAff aff_double(const OddAff& a) {
    auto xx = NTL::sqr(a.x);
    auto yy = NTL::sqr(a.y);
    auto num_x = 2 * a.x * a.y * a.curve.c;
    auto num_y = (yy - xx) * a.curve.c;
    auto den_x = xx + yy;
    auto den_y = 2 * NTL::sqr(a.curve.c) - (xx + yy);

    return OddAff(num_x / den_x, num_y / den_y, a.curve);
}


/**
```

```
 * @var BinaryAff
 * @brief Affine points on a binary curve
 *
 * This typedef builds the four functions necessary to flesh out our Afine
 * template for binary affine points.
 */
using BinaryAff = Affine<NTL::GF2E, BinaryCurve>;

/**
 * @p aff_id(const BinaryCurve& curve)
 * @brief Affine neutral element on binary curve.
 */
inline BinaryAff aff_id(const BinaryCurve& curve) {
    return BinaryAff(NTL::GF2E::zero(), NTL::GF2E::zero(), curve);
}

/**
 * @p aff_add(const BinaryAff& a1, const BinaryAff& a2)
 * @brief Affine addition for points on a binary curve.
 */
inline BinaryAff aff_add(const BinaryAff& a1, const BinaryAff& a2) {
    auto w1 = a1.x + a1.y;
    auto w2 = a2.x + a2.y;
    auto a = NTL::sqr(a1.x) + a1.x;
    auto b = NTL::sqr(a1.y) + a1.y;
    auto c = a1.curve.d * w1 * w2;
    auto d = a2.x * a2.y;

    return BinaryAff(
            a1.y + (c + a1.curve.c * (w1 + a2.x) + a * (d + a2.x)) /
                (a1.curve.c + a*w2),
            a1.x + (c + a1.curve.c * (w1 + a2.y) + b * (d + a2.y)) /
                (a1.curve.c + b * w2),
            a1.curve);
}

/**
 * @p aff_neg(const BinaryAff& a)
 * @brief Negation of an affine point on a binary curve.
 */
inline BinaryAff aff_neg(const BinaryAff& a) {
    return BinaryAff(a.y, a.x, a.curve);
}

/**
 * @p aff_double(const BinaryAff& a)
 * @brief Affine point doubling on binary curve.
 */
inline BinaryAff aff_double(const BinaryAff& a) {
    auto aa = NTL::sqr(a.x);
    auto b = NTL::sqr(aa);
    auto c = NTL::sqr(a.y);
    auto d = NTL::sqr(c);
    auto f = a.curve.c;
    auto g = (a.curve.d / a.curve.c) * (b + d);
    auto j = aa + c;
    auto k = g + a.curve.d * j;
    auto z = f + j + g;

    return BinaryAff((k + aa + d) / z,
                     (k + c + b) / z,
                     a.curve);
}

/**
 * @p birMapAff(const NTL::GF2E& u, const NTL::GF2E& v,
 * const BinaryCurve& curve)
 * @brief Birational Map from Weierstrass curve to Binary Edwards curve.
 */
inline BinaryAff birMapAff(const NTL::GF2E& u, const NTL::GF2E& v,
        const NTL::GF2E& a2, const BinaryCurve& curve) {
    NTL::GF2E x, y;
    mol_bm_aff(x, y, u, v, NTL::GF2E::degree(), curve.c, curve.d, a2);
    return BinaryAff(x, y, curve);
}


/**
 * @var TwistedAff
 * @brief Affine points on a twisted curve
 *
 * This typedef builds the four functions necessary to flesh out our Affine
 * template for twisted affine points.
 */
using TwistedAff = Affine<NTL::ZZ_pE, TwistedCurve>;

/**
 * @p aff_id(const TwistedCurve& curve)
 * @brief Affine neutral element on twisted curve.
 */
inline TwistedAff aff_id(const TwistedCurve& curve) {
    return TwistedAff(NTL::ZZ_pE::zero(), NTL::to_ZZ_pE(1), curve);
}

/**
 * @p aff_add(const TwistedAff& a1, const TwistedAff& a2)
 * @brief Affine addition for points on a twisted curve.
 */
inline TwistedAff aff_add(const TwistedAff& a1, const TwistedAff& a2) {
    auto w = a1.curve.d * a1.x * a2.x * a1.y * a2.y;
    auto num_x = a1.x * a2.y + a1.y * a2.x;
    auto num_y = a1.y * a2.y - a1.curve.d * a1.x * a2.x;
    auto den_x = 1 + w;
    auto den_y = 1 - w;

    return TwistedAff(num_x / den_x, num_y / den_y, a1.curve);
}

```

```
/**
 * @p aff_neg(const TwistedAff& a)
 * @brief Negation of an affine point on a twisted curve.
 */
inline TwistedAff aff_neg(const TwistedAff& a) {
   return TwistedAff(-a.x, a.y, a.curve);
}

/**
 * @p aff_double(const TwistedAff& a)
 * @brief Affine point doubling on a twisted curve.
 */
inline TwistedAff aff_double(const TwistedAff& a) {
   auto b = NTL::sqr(a.x + a.y);
   auto c = NTL::sqr(a.x);
   auto d = NTL::sqr(a.y);
   auto e = a.curve.c * c;
   auto f = e + d;
   auto j = f - 2;
   auto z = f * j;

   return TwistedAff(((b - c - d) * j) / z,
                     (f * (e - d)) / z,
                     a.curve);
}


//------- Projective Specialization Functions -------//

/**
 * @var OddProj
 * @brief Projective points on an odd curve
 *
 * This typedef builds the four functions necessary to flesh out our
 * Projective template for odd affine points.
 */
using OddProj = Projective<NTL::ZZ_pE, OddCurve>;

/**
 * @p proj_id(const OddCurve& curve)
 * @brief Projective neutral element on odd curve.
 */
inline OddProj proj_id(const OddCurve& curve) {
   return OddProj(NTL::ZZ_pE::zero(),
           curve.c,
           NTL::to_ZZ_pE(1),
           curve);
}

/**
 * @p proj_add(const OddProj& p1, const OddProj& p2)
 * @brief Projective addition for points on an odd curve.
 */
inline OddProj proj_add(const OddProj& p1, const OddProj& p2) {
   // From Bernstein & Lange, "Faster Addition and Doubling on Elliptic
   // Curves"
   auto a = p1.z * p2.z;
   auto b = NTL::sqr(a);
   auto c = p1.x * p2.x;
   auto d = p1.y * p2.y;
   auto e = p1.curve.d * c * d;
   auto f = b - e;
   auto g = b + e;
   return OddProj(a * f * ((p1.x + p1.y) * (p2.x + p2.y) - c - d),
                  a * g * (d - c),
                  p1.curve.c * f * g,
                  p1.curve);
}

/**
 * @p proj_neg(const OddProj& p)
 * @brief Negation of an projective point on an odd curve.
 */
inline OddProj proj_neg(const OddProj& p) {
   return OddProj(-p.x, p.y, p.z, p.curve);
}

/**
 * @p proj_double(const OddProj& p)
 * @brief Projective point doubling on odd curve.
 */
inline OddProj proj_double(const OddProj& p) {
   /// From Bernstein & Lange, "Faster Addition and Doubling on Elliptic
   /// Curves"
   auto b = NTL::sqr(p.x + p.y);
   auto c = NTL::sqr(p.x);
   auto d = NTL::sqr(p.y);
   auto e = c + d;
   auto h = NTL::sqr(p.curve.c * p.z);
   auto j = e - 2 * h;
   return OddProj(p.curve.c * (b - e) * j,
                  p.curve.c * e * (c - d),
                  e * j,
                  p.curve);
}


/**
 * @var BinaryProj
 * @brief Projective points on an binary curve
 *
 * This typedef builds the four functions necessary to flesh out our
 * Projective template for binary affine points.
 */
using BinaryProj = Projective<NTL::GF2E, BinaryCurve>;

/**
 * @p proj_id(const BinaryCurve& curve)
```

```
 * @brief Projective neutral element on binary curve.
 */
inline BinaryProj proj_id(const BinaryCurve& curve) {
   return BinaryProj(NTL::GF2E::zero(),
                     NTL::GF2E::zero(),
                     NTL::to_GF2E(1),
                     curve);
}

/**
 * @p proj_add(const BinaryProj& p1, const BinaryProj& p2)
 * @brief Projective addition for points on a binary curve.
 */
inline BinaryProj proj_add(const BinaryProj& p1, const BinaryProj& p2) {
   /// from Bernstein, Lange, and Farashahi, "Binary Edwards Curves"
   auto w1 = p1.x + p1.y;
   auto w2 = p2.x + p2.y;
   auto a = p1.x * (p1.x + p1.z);
   auto b = p1.y * (p1.y + p1.z);
   auto c = p1.z * p2.z;
   auto d = w2 * p2.z;
   auto e = p1.curve.c * NTL::sqr(c);
   auto h = (p1.curve.c * p2.z + p1.curve.d * w2) * w1 * c;
   auto i = p1.curve.c * c * p1.z;
   auto u = e + a * d;
   auto v = e + b * d;
   auto s = u * v;
   return BinaryProj(
          s * p1.y + (h + p2.x * (i + a * (p2.y + p2.z))) * v * p1.z,
          s * p1.x + (h + p2.y * (i + b * (p2.x + p2.z))) * u * p1.z,
          s * p1.z,
          p1.curve);
}

/**
 * @p proj_neg(const BinaryProj& p)
 * @brief Negation of an projective point on a binary curve.
 */
inline BinaryProj proj_neg(const BinaryProj& p) {
   return BinaryProj(p.y, p.x, p.z, p.curve);
}

/**
 * @p proj_double(const BinaryProj& p)
 * @brief Projective point doubling on binary curve.
 */
inline BinaryProj proj_double(const BinaryProj& p) {
   /// from Bernstein, Lange, and Farashahi, "Binary Edwards Curves"
   auto a = NTL::sqr(p.x);
   auto b = NTL::sqr(a);
   auto c = NTL::sqr(p.y);
   auto d = NTL::sqr(c);
   auto e = NTL::sqr(p.z);
   auto f = p.curve.c * NTL::sqr(e);
```

```
    auto g = (p.curve.d / p.curve.c) * (b + d);
    auto h = a * e;
    auto i = c * e;
    auto j = h + i;
    auto k = g + p.curve.d * j;
    return BinaryProj(k + h + d, k + i + b, f + j + g, p.curve);
}

/**
 * @p birMapProj(const NTL::GF2E& u, const NTL::GF2E& v,
 * const BinaryCurve& curve)
 * @brief Birational Map from Weierstrass curve to Binary Edwards curve.
 */
inline BinaryProj birMapProj(const NTL::GF2E& u, const NTL::GF2E& v,
       const NTL::GF2E& a2, const BinaryCurve& curve) {
    NTL::GF2E x, y, z;
    mol_bm_proj(x, y, z, u, v, NTL::GF2E::degree(), curve.c, curve.d, a2);
    return BinaryProj(x, y, z, curve);
}


/**
 * @var TwistedProj
 * @brief Projective points on an twisted curve
 *
 * This typedef builds the four functions necessary to flesh out our
 * Projective* template for twisted affine points.
 */
using TwistedProj = Projective<NTL::ZZ_pE, TwistedCurve>;

/**
 * @p proj_id(const TwistedCurve& curve)
 * @brief Projective neutral element on twisted curve.
 */
inline TwistedProj proj_id(const TwistedCurve& curve) {
    return TwistedProj(NTL::ZZ_pE::zero(),
                       NTL::to_ZZ_pE(1),
                       NTL::to_ZZ_pE(1),
                       curve);
}

/**
 * @p proj_add(const TwistedProj& p1, const TwistedProj& p2)
 * @brief Projective addition for points on a twisted curve.
 */
inline TwistedProj proj_add(const TwistedProj& p1, const TwistedProj& p2) {
    /// From Bernstein, Birkner, Joye, Lange, Peters, "Twisted Edwards
    /// Curves"
    auto a = p1.z * p2.z;
    auto b = NTL::sqr(a);
    auto c = p1.x * p2.x;
    auto d = p1.y * p2.y;
    auto e = p1.curve.d * c * d;
    auto f = b - e;
```

```
        auto g = b + e;
        return TwistedProj(a * f * ((p1.x + p1.y) * (p2.x + p2.y) - c - d),
                           a * g * (d - p1.curve.c * c),
                           f * g,
                           p1.curve);
    }

    /**
     * @p proj_neg(const TwistedProj& p)
     * @brief Negation of an projective point on a twisted curve.
     */
    inline TwistedProj proj_neg(const TwistedProj& p) {
        return TwistedProj(-p.x, p.y, p.z, p.curve);
    }

    /**
     * @p proj_double(const TwistedProj& p)
     * @brief Projective point doubling on twisted curve.
     */
    inline TwistedProj proj_double(const TwistedProj& p) {
        /// From Bernstein, Birkner, Joye, Lange, Peters, "Twisted Edwards
        /// Curves"
        auto b = NTL::sqr(p.x + p.y);
        auto c = NTL::sqr(p.x);
        auto d = NTL::sqr(p.y);
        auto e = p.curve.c * c;
        auto f = e + d;
        auto h = NTL::sqr(p.z);
        auto j = f - 2 * h;
        return TwistedProj((b - c - d) * j, f * (e - d), f * j, p.curve);
    }
}



#endif // _POINTS_H
```

Listing 7.9: utilities.h

```
/**
 * @file utilities.h
 * @brief Utilties for Edwards Curves and points on them
 * @author Graham Enos
 *
 * This file contains various utilities for e2c2.
 */


#ifndef _UTILITIES_H
#define _UTILITIES_H


#include <algorithm>     // For reverse
#include <sstream>       // For stringstream set_parameter hackery
```

```
#include <stdexcept>     // Exceptions


/// Namespace for our library
namespace e2c2 {

    /**
     * @p set_parameter(T& param, const std::string& value,
     *      const bool& hex_and_rev=false)
     * @brief Generic hackery to set a parameter to a string
     *
     * @tparam T      type of parameter
     *
     * This function sets the parameter "param" to the value given in the
     * string "value." Basically this is to smooth over some of the rough edges
     * of NTL.
     */
    template <class T>
    void set_parameter(T& param, const std::string& value,
            const bool& hex_and_rev=false) {
        std::stringstream ss;
        std::ostream& out = ss;
        std::istream& in = ss;
        std::string v(value.begin(), value.end());
        if (hex_and_rev) {
            reverse(v.begin(), v.end());
            out << "0x";
        }
        out << v;
        in >> param;
    }


    /**
     * @p InvalidParametersException
     * @brief Custom exception to be thrown when building a curve with invalid
     * parameters
     */
    class InvalidParametersException : public std::invalid_argument {
    public:
        InvalidParametersException() :
            std::invalid_argument("INVALID PARAMETERS") {}
    };


    /**
     * @p NotImplementedException
     * @brief Custom exception to be thrown when we reach the limits of current
     * implementation
     */
    class NotImplementedException : public std::runtime_error {
    public:
        NotImplementedException() :
            std::runtime_error("NOT YET IMPLEMENTED") {}
    };


    /**
     * @p DifferentCurvesException
     * @brief Custom exception to be thrown when attempting to operate on
     * points from different curves
     */
    class DifferentCurvesException : public std::invalid_argument {
    public:
        DifferentCurvesException() :
            std::invalid_argument("THESE POINTS BELONG TO DIFFERENT CURVES") {}
    };
}
#endif // _UTILITIES_H
```

Listing 7.10: mol.h

```
/**
 * @file mol.h
 * @brief Birational Map and utilities from MOL Paper
 * @author Graham Enos
 *
 * This file contains the interface to the C++ implementation of the birational
 * map and assorted utility functions given in Moloney, O'Mahony, & Laurent's
 * paper, available here: http://eprint.iacr.org/2010/208
 */

#ifndef _MOL_H
#define _MOL_H

#include <iostream>      // Readable output
#include <NTL/ZZ.h>      // Arbitrarily large integers
#include <NTL/ZZ_pE.h>   // Field elements from @f$ \mathbf{F}_{p^n} @f$
#include <NTL/GF2E.h>    // Field elements from @f$ \mathbf{F}_{2^n} @f$
#include "curves.h"      // Edwards curves (and variations)
#include "utilities.h"   // Utilities header for e2c2 project


/// Namespace for our library
namespace e2c2 {

    /// Per MOL paper, \f$\sqrt{\alpha} = \alpha^{2^{m-1}}\f$
    inline const NTL::GF2E gf2m_sqrt(const NTL::GF2E& alpha, const long& m) {
        auto s = alpha;
        for (auto i = 0L; i < m - 1; ++i) {
            sqr(s, s);
        }
        return s;
    }


    /// Half-Trace function
    inline const NTL::GF2E half_trace(const NTL::GF2E& alpha, const long m) {
```

```
    auto ht = NTL::GF2E::zero();
    auto e = NTL::to_ZZ(1);
    for (auto i = 0L; i <= (m - 1) / 2; ++i) {
        ht += power(alpha, e);
        e <<= 2;
    }
    return ht;
}


/// MOL's Algorithm 1 to compute d1
inline const NTL::GF2E mol_alg_1(const long& n, const NTL::GF2E& a2,
                        const NTL::GF2E& a6) {
    auto t = trace(a2), r = trace(a6);
    auto a6_2 = gf2m_sqrt(a6, n);
    auto a6_4 = gf2m_sqrt(a6_2, n);
    auto x = NTL::GF2E::zero(), d1 = NTL::GF2E::zero();
    set_parameter(x, "[0 1]");
    auto w = x + trace(x);
    if (t == 0 && r == 1) {
        set(d1);
    } else {
        if (t == 1 && r == 0) {
            d1 = a6_4;
        } else {
            if (t == 1 && r == 1 && a6 != 1) {
                if (trace(inv(a6 + 1)) == 1) {
                    d1 = a6_2 + a6_4;
                } else {
                    d1 = a6_4 + 1;
                }
            } else {
                if (t == 1 && a6 == 1) {
                    if (trace(inv(w)) == 1) {
                        d1 = w;
                    } else {
                        if (trace(inv(w + 1)) == 1) {
                            d1 = inv(w + 1);
                        } else {
                            d1 = inv(w + 1) + 1;
                        }
                    }
                } else {
                    if (t == 0 && r == 0) {
                        if (trace(inv(a6 + 1)) == 0) {
                            d1 = a6_4 + 1;
                        } else {
                            auto i = 1;
                            auto s = a6_2;
                            while (trace(NTL::power(a6,
                                            NTL::power_long(2, i) + 1))
                                    == 0) {
                                s *= s;
                                ++i;
                            }
                            d1 = inv(s + 1);
                        }
                    }
                }
            }
        }
    }
    return d1;
}


/// Construct a Binary Edwards Curve from Weierstrass Parameters, degree of
/// field extension, and supplied cardinality of curve
inline BinaryCurve from_weierstrass(const long n, const NTL::ZZ& m,
        const NTL::GF2E& a2, const NTL::GF2E& a6) {
    auto c = mol_alg_1(n, a2, a6);
    auto d = NTL::sqr(c) + c + gf2m_sqrt(a6, n) / NTL::sqr(c);
    return BinaryCurve(c, d, m);
}


/// MOL's birational map from Weierstrass curve to Affine Binary Edwards
inline void mol_bm_aff(NTL::GF2E& x, NTL::GF2E& y, const NTL::GF2E& u,
            const NTL::GF2E& v, const long m, const NTL::GF2E& d1,
            const NTL::GF2E& d2, const NTL::GF2E& a2) {
    auto b = half_trace(sqr(d1) + d2 + a2, m), tmp = sqr(d1) + d1 + d2;
    auto z = sqr(u) + d1 * u + sqr(d1) * tmp;
    x = d1 * (b * u + v + (sqr(d1) + d1) * tmp);
    y = (x + d1 * u);
    x /= z;
    y /= z;
}


/// MOL's birational map from Weierstrass curve to Projective Binary
/// Edwards
inline void mol_bm_proj(NTL::GF2E& x, NTL::GF2E& y, NTL::GF2E& z,
        const NTL::GF2E& u, const NTL::GF2E& v, const long m,
        const NTL::GF2E& d1, const NTL::GF2E& d2, const NTL::GF2E& a2) {
    auto b = half_trace(sqr(d1) + d2 + a2, m), tmp = sqr(d1) + d1 + d2;
    x = d1 * (b * u + v + (sqr(d1) + d1) * tmp);
    y = (x + d1 * u);
    z = sqr(u) + d1 * u + sqr(d1) * tmp;
}
}
#endif // _MOL_H
```

Listing 7.11: curves_test.cc

```
/**
 * @file curves_test.cc
 * @brief A quick test of curve functionality
 * @author Graham Enos
 *
 * This file gives a demonstration of current curve functionality in e2c2.
 */

#include <cstdlib>       // User input
#include <iostream>      // Readable output
#include "e2c2.h"

using namespace std;
using namespace NTL;
using namespace e2c2;


/**
 * @p odd_test()
 * @brief Test of odd curve functionality
 */
void odd_test() {
    ZZ_p::init(power2_ZZ(255) - 19); /// Sets F_p
    ZZ_pE::init(ZZ_pX(1, 1)); /// Sets F_(p^n) = F_(p^1)

    /// Bernstein's "Curve25519"
    auto c = to_ZZ_pE(1), d = to_ZZ_pE(121665) / to_ZZ_pE(121666);
    auto m = 8 * (power2_ZZ(252) +
                  to_ZZ("27742317777372353535851937790883648493"));
    OddCurve o(c, d, m);
    cout << o << endl;
}


/**
 * @p binary_test()
 * @brief Test of binary curve functionality
 */
void binary_test() {
    /// Our irred. polynomial is x^163 + x^7 + x^6 + x^3 + 1, per FIPS 186-3
    GF2E::init(GF2X(163, 1) + GF2X(7, 1) + GF2X(6, 1) + GF2X(3, 1) +
               GF2X(0, 1));
    GF2X::HexOutput = true; /// more compact output
    auto n = 163;
    auto m = to_ZZ("5846006549323611672814742442876390689256843201587");
    auto a2 = to_GF2E(1), a6 = GF2E::zero();
    /// a6 = b in Fips 186-3 language
    set_parameter(a6, "20a601907b8c953ca1481eb10512f78744a3205fd", true);
    auto b_163 = from_weierstrass(n, m, a2, a6);

    cout << b_163 << endl;
    cout << "Here are the (c, d) parameters again: " << endl;
    cout << "(" << b_163.c << ", " << b_163.d << ")" << endl;

    /// Purposefully raising an InvalidParametersException
    cout << endl << endl << "Hold on tight; I'm going to try to"
        << " make a binary curve with c = d = 0..." << endl;
    BinaryCurve b(GF2E::zero(), GF2E::zero(), ZZ::zero());
}


/**
 * @p twisted_test()
 * @brief Test of twisted curve functionality
 */
void twisted_test() {
    ZZ_p::init(power2_ZZ(255) - 19); /// Sets F_p
    ZZ_pE::init(ZZ_pX(1, 1)); /// Sets F_(p^n) = F_(p^1)

    /// Bernstein's "Curve25519," in twisted form
    auto c = to_ZZ_pE(121666), d = to_ZZ_pE(121665);
    auto m = 8 * (power2_ZZ(252) +
                 to_ZZ("27742317777372353535851937790883648493"));
    TwistedCurve t(c, d, m);
    cout << t << endl;
}


/**
 * @p main(int argc, char *argv[])
 * @brief Runs appropriate curve test
 */
int main(int argc, char *argv[]) {
    try {
        if (argc == 1)
            binary_test();
        else {
            switch(atoi(argv[1])) {
            case 0:
                odd_test();
                break;
            case 1:
                binary_test();
                break;
            case 2:
                twisted_test();
                break;
            default:
                cout << "Please select a type of curve." << endl;
            }
        }
    } catch (InvalidParametersException& e) {
        cout << e.what() << endl;
    }

    return 0;
}
```

Listing 7.12: points_test.cc

```
/**
 * @file points_test.cc
```

```cpp
 * @brief A quick test of point functionality
 * @author Graham Enos
 *
 * This file gives a demonstration of current point functionality in e2c2.
 */

#include <cstdlib>       // User input
#include <iostream>      // Readable output
#include "e2c2.h"

using namespace std;
using namespace NTL;
using namespace e2c2;


/**
 * @p odd_test()
 * @brief Test of odd point functionality
 */
void odd_test() {
    ZZ_p::init(power2_ZZ(255) - 19); /// Sets F_p
    ZZ_pE::init(ZZ_pX(1, 1)); /// Sets F_(p^n) = F_(p^1)

    /// Bernstein's "Curve25519"
    auto c = to_ZZ_pE(1), d = to_ZZ_pE(121665) / to_ZZ_pE(121666);
    auto m = 8 * (power2_ZZ(252) +
                 to_ZZ("27742317777372353535851937790883648493"));
    OddCurve o(c, d, m);
    cout << o << endl;

    OddAff id(o);
    cout << "id = " << id << endl;
    cout << "17 * id = " << 17 * id << endl;

    auto x = to_ZZ_pE(1), y = ZZ_pE::zero(), z = to_ZZ_pE(1);
    OddProj point1(x, y, z, o), point2(point1);
    for (auto i = 0; i < 4; ++i) {
        cout << "ProjectivePoint 2 = " << point2 << endl;
        cout << "ProjPoint2 + ProjPoint1 = " << point2 + point1 << endl;
        point2 += point1;
    }

    cout << (point1 + OddProj(OddAff(x, y, o))) << endl;

    cout << -point1 << endl;

    cout << "3 * point2 = " << 3 * point2
        << endl;

}


/**
 * @p binary_test()
 * @brief Test of binary point functionality
 */
void binary_test() {
    /// Our irred. polynomial is x^163 + x^7 + x^6 + x^3 + 1, per FIPS 186-3
    GF2E::init(GF2X(163, 1) + GF2X(7, 1) + GF2X(6, 1) + GF2X(3, 1) +
               GF2X(0, 1));
    GF2X::HexOutput = true; /// more compact output
    auto n = 163;
    auto m = to_ZZ("5846006549323611672814742442876390689256843201587");
    auto a2 = to_GF2E(1), a6 = GF2E::zero();
    /// a6 = b in Fips 186-3 language
    set_parameter(a6, "20a601907b8c953ca1481eb10512f78744a3205fd", true);
    auto b_163 = from_weierstrass(n, m, a2, a6);
    cout << b_163 << endl;

    BinaryAff id(b_163);
    cout << "id = " << id << endl;
    cout << "17 * id = " << 17 * id << endl;

    GF2E x = to_GF2E(1);
    GF2E y = to_GF2E(1);
    GF2E z = to_GF2E(1);
    BinaryProj point1(x, y, z, b_163);
    BinaryProj point2(point1);
    for (auto i = 0; i < 4; ++i) {
        cout << "ProjectivePoint 2 = " << point2 << endl;
        cout << "ProjPoint2 + ProjPoint1 = " << point2 + point1 << endl;
        point2 += point1;
    }

    cout << (point1 + BinaryProj(BinaryAff(x, y, b_163))) << endl;

    cout << -point1 << endl;

    cout << "3 * point2 = " << 3 * point2
        << endl;
    /// Purposefully raising an InvalidParametersException
    cout << endl << endl << "Hold on tight; I'm going to try to"
          << " make a binary point with (x : y : z) = (1 : 0 : 1)..." << endl;
    BinaryProj(to_GF2E(1), GF2E::zero(), to_GF2E(1), b_163);
}


/**
 * @p twisted_test()
 * @brief Test of twisted point functionality
 */
void twisted_test() {
    ZZ_p::init(power2_ZZ(255) - 19); /// Sets F_p
    ZZ_pE::init(ZZ_pX(1, 1)); /// Sets F_(p^n) = F_(p^1)

    /// Bernstein's "Curve25519," in twisted form
    auto c = to_ZZ_pE(121666), d = to_ZZ_pE(121665);
    auto m = 8 * (power2_ZZ(252) +
                to_ZZ("277423177773723535358519377908836484493"));
    TwistedCurve t(c, d, m);
    cout << t << endl;

    TwistedAff id(t);
    cout << "id = " << id << endl;
    cout << "17 * id = " << 17 * id << endl;

    ZZ_pE x = ZZ_pE::zero();
    ZZ_pE y = to_ZZ_pE(-1);
    ZZ_pE z = to_ZZ_pE(1);
    TwistedProj point1(x, y, z, t);
    TwistedProj point2(point1);
    for (auto i = 0; i < 4; ++i) {
        cout << "ProjectivePoint 2 = " << point2 << endl;
        cout << "ProjPoint2 + ProjPoint1 = " << point2 + point1 << endl;
        point2 += point1;
    }

    cout << (point1 + TwistedProj(TwistedAff(x, y, t))) << endl;

    cout << -point1 << endl;

    cout << "3 * point2 = " << 3 * point2
        << endl;
    /// Purposefully raising an InvalidParametersException
    cout << endl << endl << "Hold on tight; I'm going to try to"
          << " make a binary point with (x : y : z) = (1 : 0 : 1)..." << endl;
    TwistedProj(to_ZZ_pE(1), ZZ_pE::zero(), to_ZZ_pE(1), t);
}


/**
 * @p main(int argc, char *argv[])
 * @brief Runs appropriate point test
 */
int main(int argc, char *argv[]) {
    try {
        if (argc == 1)
            binary_test();
        else {
            switch(atoi(argv[1])) {
            case 0:
                odd_test();
                break;
            case 1:
                binary_test();
                break;
            case 2:
                twisted_test();
                break;
            default:
                cout << "Please select a type of curve." << endl;
            }
        }
    } catch (InvalidParametersException& e) {
        cout << e.what() << endl;
    }

    return 0;
}
```

Listing 7.13: key_demo.cc

```
/**
 * @file key_demo.cc
 * @brief Demo of DH Key Exchange with projective points
 * @author Graham Enos
 *
 * This is a sample file that makes use of the classes and methods in
 * b_projective.cc and b_edwards.cc to conduct a DH key exchange demonstration.
 */

#include <iostream>
#include <sstream>
#include "e2c2.h"

using namespace std;
using namespace NTL;
using namespace e2c2;

/**
 * @brief A quick key exchange demo
 *
 * This function shows how to use our projective points on a Binary Edwards
 * Curve to conduct a Diffie-Hellman key exchange.
 */
int main(int argc, char *argv[]) {
    /// Our irred. polynomial is x^163 + x^7 + x^6 + x^3 + 1, per FIPS 186-3
    GF2E::init(GF2X(163, 1) + GF2X(7, 1) + GF2X(6, 1) + GF2X(3, 1) +
               GF2X(0, 1));
    GF2X::HexOutput = true; /// more compact output
    auto n = 163;
    GF2E a2 = to_GF2E(1), a6;
    /// a6 = b in Fips 186-3 language
    set_parameter(a6,
                  string("20a601907b8c953ca1481eb10512f78744a3205fd"),
                  true);
    auto m = to_ZZ("5846006549323611672814742442876390689256843201587");
    auto c = mol_alg_1(n, a2, a6);
    auto d = NTL::sqr(c) + c + gf2m_sqrt(a6, n) / NTL::sqr(c);
    auto b_163 = from_weierstrass(n, m, a2, a6);

    /// Weierstrass params; need to be changed to Edwards Curve
    GF2E g_x, g_y, x, y, z;
    set_parameter(g_x, "3f0eba16286a2d57ea0991168d4994637e8343e36", true);
    set_parameter(g_y, "0d51fbc6c71a0094fa2cdd545b11c5c0c797324f1", true);
    mol_bm_proj(x, y, z, g_x, g_y, n, c, d, a2);
    BinaryProj P(x, y, z, b_163);
    BinaryProj id(b_163);

    /// Key Exchange
    cout << "Alice and Bob wish to communicate in private, " <<
        "so they need a shared secret key." << endl << endl << endl;
    /// Seeding pseudorandom generator
    SetSeed(argc <= 1 ? to_ZZ(1729) : to_ZZ(argv[1]));
    /// Alice's first steps
    auto a = RandomLen_ZZ(NumBits(m));
    cout << "Alice first picks a secret random number a = " << a <<
        " with the same number of bits as m (the size of our group)..." <<
        endl;
    auto aP = a * P;
    cout << "...then sends Bob the point pA = a * P (the generator) = "
        << aP << endl << endl;
    /// Bob's first steps
    auto b = RandomLen_ZZ(NumBits(m));
    cout << "Bob also picks a secret random number b = " << b << endl;
    auto bP = b * P;
    cout << "Then he sends Alice the point bP = b * P = " << bP << endl
        << endl;
    /// Alice reconstructs private key
    auto key_a = a * bP;
    cout << "Then Alice takes a and multiplies it by bP to get key_a = " <<
        key_a << endl;
    /// Bob does the same
    auto key_b = b * aP;
    cout << "Similarly, Bob calculates key_b = b * aP = " << key_b << endl;
    /// Quick check
    cout << "Are these in fact the same key? " << endl;
    if (key_a == key_b) {
        cout << "Yes!" << endl <<
            "So now they share a secret key, and can communicate securely."
             << endl << endl << endl;
    } else {
        cout << "NO...uh oh, my code is wrong somewhere..." << endl;
    }
    return 0;
}
```

REFERENCES

[1] Adams, W., and Loustaunau, P. *An introduction to Gröbner bases*, vol. 3. Amer Mathematical Society, 1994.

[2] Alnoon, H., and Al Awadi, S. Executing parallelized dictionary attacks on cpus and gpus. Tech. rep., ENSIMAG, 2009.

[3] Arene, C., Lange, T., Naehrig, M., and Ritzenthaler, C. Faster computation of the tate pairing. *Journal of number theory 131*, 5 (2011), 842–857.

[4] Baek, J., and Zheng, Y. Identity-based threshold decryption. *Public Key Cryptography–PKC 2004* (2004), 262–276.

[5] Baum, M. NIST selects winner of secure hash algorithm (SHA-3) competition. Tech. rep., National Institute of Stanards and Technology, 2012.

[6] Bernstein, D., Birkner, P., Joye, M., Lange, T., and Peters, C. Twisted edwards curves. *Progress in Cryptology–AFRICACRYPT 2008* (2008), 389–405.

[7] Bernstein, D., and Lange, T. Faster addition and doubling on elliptic curves. *Advances in cryptology–ASIACRYPT 2007* (2007), 29–50.

[8] Bernstein, D., Lange, T., and Rezaeian Farashahi, R. Binary edwards curves. *Cryptographic Hardware and Embedded Systems–CHES 2008* (2008), 244–265.

[9] Billet, O., and Joye, M. The jacobi model of an elliptic curve and side-channel analysis. *Applied Algebra, Algebraic Algorithms and Error-Correcting Codes* (2003), 604–604.

[10] Boneh, D., and Franklin, M. Identity-based encryption from the weil pairing. In *Advances in Cryptology–CRYPTO 2001* (2001), Springer, pp. 213–229.

[11] Brickell, E. Some ideal secret sharing schemes. In *Advances in Cryptology–EUROCRYPT 89* (1990), Springer, pp. 468–475.

[12] Brier, E., and Joye, M. Weierstraß elliptic curves and side-channel attacks. In *Public Key Cryptography* (2002), Springer, pp. 183–194.

[13] Brown, D., Antipa, A., Campagna, M., and Struik, R. Ecoh: the elliptic curve only hash. Submission to NIST, 2008.

[14] Brumley, B. B., and Tuveri, N. Remote timing attacks are still practical. Cryptology ePrint Archive, Report 2011/232, 2011.

[15] Chatterjee, A., and Sengupta, I. Fpga implementation of binary edwards curve usingternary representation. In *Proceedings of the 21st edition of the great lakes symposium on Great lakes symposium on VLSI* (2011), ACM, pp. 73–78.

[16] Chevallier-Mames, B., Ciet, M., and Joye, M. Low-cost solutions for preventing simple side-channel analysis: Side-channel atomicity. *Computers, IEEE Transactions on 53*, 6 (2004), 760–768.

[17] Chow, S., Yiu, S., Hui, L., and Chow, K. Efficient forward and provably secure id-based signcryption scheme with public verifiability and public ciphertext authenticity. *Information Security and Cryptology-ICISC 2003* (2004), 352–369.

[18] Cohen, H., Frey, G., and Avanzi, R. *Handbook of elliptic and hyperelliptic curve cryptography.* CRC press, 2006.

[19] Cohen, H., Frey, G., Avanzi, R., Doche, C., Lange, T., Nguyen, K., and Vercauteren, F. *Handbook of elliptic and hyperelliptic curve cryptography.* Chapman & Hall/CRC, 2005.

[20] Coron, J.-S. Resistance against differential power analysis for elliptic curve cryptosystems. In *Cryptographic Hardware and Embedded Systems* (1999), Springer, pp. 725–725.

[21] Daemen, J., and Rijmen, V. *The design of Rijndael: AES–the advanced encryption standard.* Springer-Verlag New York Inc, 2002.

[22] Das, M., and Sarkar, P. Pairing computation on twisted edwards form elliptic curves. *Pairing-Based Cryptography–Pairing 2008* (2008), 192–210.

[23] Diao, O., and Fouotsa, E. Edwards model of elliptic curves defined over any fields. Tech. rep., Cryptology ePrint Archive: Report 2012/346. http://eprint.iacr.org/2012/346, 2012.

[24] Diffie, W., and Hellman, M. New directions in cryptography. *Information Theory, IEEE Transactions on 22*, 6 (1976), 644–654.

[25] Edwards, H. A normal form for elliptic curves. *Bulletin of the American Mathematical Society 44*, 3 (2007), 393–422.

[26] Farashahi, R., and Joye, M. Efficient arithmetic on hessian curves. *Public Key Cryptography–PKC 2010* (2010), 243–260.

[27] Florencio, D., and Herley, C. A large-scale study of web password habits. In *Proceedings of the 16th international conference on World Wide Web* (2007), ACM, pp. 657–666.

[28] for Standardization, I. O. Information technology–programming languages–c++. sc22/wg21. iso/iec 14882: 2011. Tech. rep., ISO, 2011.

[29] Galbraith, S., Harrison, K., and Soldera, D. Implementing the tate pairing. *Algorithmic Number Theory* (2002), 69–86.

[30] Gallagher, P., Foreword, D. D., and Director, C. F. Fips pub 186-3 federal information processing standards publication digital signature standard (dss), 2009.

[31] Ghodosi, H., Pieprzyk, J., and Safavi-Naini, R. Secret sharing in multi-level and compartmented groups. In *Information Security and Privacy* (1998), Springer, pp. 367–378.

[32] Gómez, J., Montoya, F., Benedicto, R., Jimenez, A., Gil, C., and Alcayde, A. Cryptanalysis of hash functions using advanced multiprocessing. *Distributed computing and artificial intelligence* (2010), 221–228.

[33] Goodin, D. Trade group exposes 100,000 passwords for Google, Apple engineers. http://arstechnica.com/security/2012/09/ieee-trade-group-exposes-100000-password-for-google-apple-engineers/, September 2012.

[34] Goubin, L. A refined power-analysis attack on elliptic curve cryptosystems. *Public Key Cryptography–PKC 2003* (2002), 199–211.

[35] Granlund, T. *GNU MP: The GNU Multiple Precision Arithmetic Library*, 1996.

[36] Grillet, P. A. *Abstract algebra*, vol. 242. Springer, 2007.

[37] Halcrow, M., and Ferguson, N. A second pre-image attack against elliptic curve only hash (ecoh). Tech. rep., Microsoft, 2009.

[38] Halcrow, M. A., and Ferguson, N. A second pre-image attack against elliptic curve only hash (ecoh). Cryptology ePrint Archive, Report 2009/168, 2009. http://eprint.iacr.org/.

[39] Hankerson, D., Menezes, A., and Vanstone, S. *Guide to elliptic curve cryptography*. Springer, 2004.

[40] Hartshorne, R. *Algebraic geometry*, vol. 52. Springer, 1977.

[41] Icart, T. How to hash into elliptic curves. *Advances in Cryptology–CRYPTO 2009* (2009), 303–316.

[42] Iftene, S. Compartmented secret sharing based on the chinese remainder theorem. Tech. rep., Cryptology ePrint Archive, 2005.

[43] Iftene, S. General secret sharing based on the chinese remainder theorem with applications in e-voting. *Electronic Notes in Theoretical Computer Science 186* (2007), 67–84.

[44] Izu, T., Möller, B., and Takagi, T. Improved elliptic curve multiplication methods resistant against side channel attacks. *Progress in CryptologyINDOCRYPT 2002* (2002), 296–313.

[45] Izu, T., and Takagi, T. Exceptional procedure attack on elliptic curve cryptosystems. *Public Key Cryptography—PKC 2003* (2002), 224–239.

[46] Joye, M. Elliptic curves and side-channel analysis. *ST Journal of System Research 4*, 1 (2003), 17–21.

[47] Joye, M., and Quisquater, J.-J. Hessian elliptic curves and side-channel attacks. In *Cryptographic Hardware and Embedded Systems–CHES 2001* (2001), Springer, pp. 402–410.

[48] Koblitz, N. Elliptic curve cryptosystems. *Mathematics of computation 48*, 177 (1987), 203–209.

[49] Koblitz, N. *Algebraic aspects of cryptography*, vol. 3. Springer, 2004.

[50] Kocabas, U. *Hardware Implementations Of ECC Over A Binary Edwards Curve*. PhD thesis, Katholieke Universiteit Leuven, 2009.

[51] Kocabas, U., Fan, J., and Verbauwhede, I. Implementation of binary edwards curves for very-constrained devices. In *Application-specific Systems Architectures and Processors (ASAP), 2010 21st IEEE International Conference on* (2010), IEEE, pp. 185–191.

[52] Kwon, S. Efficient tate pairing computation for elliptic curves over binary fields. In *Information Security and Privacy* (2005), Springer, pp. 134–145.

[53] Li, F., Xin, X., and Hu, Y. ID-based signcryption scheme with (t, n) shared unsigncryption. *International Journal of Network Security 3*, 2 (2006), 155–159.

[54] Li, L., Wu, H., and Zhang, F. Pairing computation on edwards curves with high-degree twists. Cryptology ePrint Archive, Report 2012/532, 2012.

[55] Lim, R. Parallelization of John the Ripper (JtR) using MPI. Tech. rep., Nebraska: University of Nebraska, 2004.

[56] Menezes, A., Okamoto, T., and Vanstone, S. Reducing elliptic curve logarithms to logarithms in a finite field. *Information Theory, IEEE Transactions on 39*, 5 (1993), 1639–1646.

[57] Menezes, A., Van Oorschot, P., and Vanstone, S. *Handbook of applied cryptography*. CRC, 1996.

[58] Meyers, S. D. *Effective C++: 55 specific ways to improve your programs and designs*. Addison-Wesley Professional, 2005.

[59] Miller, V. Use of elliptic curves in cryptography. In *Advances in Cryptology: CRYPTO85 Proceedings* (1986), Springer, pp. 417–426.

[60] Miller, V. S. The weil pairing, and its efficient calculation. *Journal of Cryptology 17*, 4 (2004), 235–261.

[61] Möller, B. Securing elliptic curve point multiplication against side-channel attacks. *Information Security* (2001), 324–334.

[62] Moloney, R., O'Mahony, A., and Laurent, P. Efficient implementation of elliptic curve point operations using binary edwards curves. Tech. rep., Cryptology ePrint Archive, Report 2010/208, 2010. http://eprint. iacr. org, 2010.

[63] NSA. The case for elliptic curve cryptography. `http://www.nsa.gov/business/programs/elliptic_curve.shtml`, January 2009.

[64] Okeya, K., and Sakurai, K. Power analysis breaks elliptic curve cryptosystems even secure against the timing attack. *Progress in Cryptology–INDOCRYPT 2000* (2000), 217–314.

[65] Percival, C. Stronger key derivation via sequential memory-hard functions. In *BSDCan 2009* (2009).

[66] Provos, N., and Mazieres, D. A future-adaptable password scheme. In *Proceedings of the Annual USENIX Technical Conference* (1999).

[67] Reddy, E. K. Elliptic curve cryptosystems and side-channel attacks. *Published in international Journals of Network Security* (2009), 151 – 158.

[68] Shamir, A. How to share a secret. *Communications of the ACM 22*, 11 (1979), 612–613.

[69] Shamir, A. Identity-based cryptosystems and signature schemes. In *Advances in cryptology* (1985), Springer, pp. 47–53.

[70] Shoup, V. NTL: A library for doing number theory. `http://www.shoup.net/ntl/index.html`, August 2009.

[71] Silverman, J. *The arithmetic of elliptic curves*, vol. 106. Springer Verlag, 2009.

[72] Simmons, G. How to (really) share a secret. In *Proceedings on Advances in cryptology* (1990), Springer-Verlag New York, Inc., pp. 390–448.

[73] Stallman, R. GNU compiler collection internals. Tech. rep., Free Software Foundation, Cambridge, MA, 2002.

[74] Stein, W., et al. *Sage Mathematics Software (Version 5.0)*. The Sage Development Team, 2012. `http://www.sagemath.org`.

[75] STINSON, D. *Cryptography: theory and practice*. Chapman & Hall/CRC, 2005.

[76] STROUSTRUP, B. *The C++ programming language*. Addison-Wesley Longman Publishing Co., Inc., 1997.

[77] TURAN, M. S., BARKER, E., BURR, W., AND CHEN, L. Recommendation for password-based key derivation. *NIST Special Publication 800* (2010), 132.

[78] WANG, B., TANG, C., AND YANG, Y. A new model of elliptic curves with effective and fast arithmetic. *Journal of Computational Information Systems 8*, 10 (2012), 4061–4067.

[79] WASHINGTON, L. *Elliptic curves: number theory and cryptography*, vol. 50. Chapman & Hall/CRC, 2008.

[80] WU, H., TANG, C., AND FENG, R. A new model of binary elliptic curves with fast arithmetic. Tech. rep., Cryptology ePrint Archive: Report 2010/608. http://eprint.iacr.org/2010/608, 2010.

[81] ZHENG, Y. Digital Signcryption or How to Achieve Cost(Signature & Encryption) << Cost(Signature) + Cost(Encryption). *Advances in Cryptology–CRYPTO 1997* (1997), 165–179.

[82] ZHENG, Y., AND IMAI, H. How to construct efficient signcryption schemes on elliptic curves. *Information Processing Letters 68*, 5 (1998), 227–233.

[83] ZONENBERG, A. Distributed hash cracker: A cross-platform gpu-accelerated password recovery system. *Rensselaer Polytechnic Institute* (2009), 27.