Thesis for the Degree of Licentiate of Engineering

Optimized Elliptic Curve Cryptography and Efficient Elliptic Curve Parameter Generation

Greger Cronquist

Department of Mathematics
Chalmers University of Technology and Göteborg University
SE-412 96 Göteborg, Sweden
Göteborg, March 2002

Optimized Elliptic Curve Cryptography and
Efficient Elliptic Curve Parameter Generation
GREGER CRONQUIST

ISSN 0347-2809/NO 2002-13
Department of Mathematics
Chalmers University of Technology and Göteborg University
SE-412 96 Göteborg
Sweden
Telephone +46 (0)31-772 1000

This is a thesis of the ECMI (European Consortium for Mathematics
in Industry) post-graduate program in Industrial Mathematics at
Chalmers University of Technology.

The work was supported by Ericsson Mobile Platform and Ericsson Radio Systems.

Matematiskt centrum
Göteborg, Sweden 2002

Abstract

This thesis is concerned with security problems related to constrained devices such as mobile phones. Devices of this type put high demands on security solutions to be cost efficient in terms of computing power, network bandwidth and memory requirements. In this thesis, we study elliptic curve cryptography and related computational problems in connection with suitable security solutions for wireless terminals. We show that elliptic curve cryptography meets high security and low cost demands for this type of devices.

In the first part of the thesis, we study efficiency problems for arithmetical operations related to elliptic curve cryptography. In the second part, we investigate two problems in elliptic curve cryptography: Efficient implementation of elliptic curve arithmetic and elliptic curve parameter selection. By making an efficient implementation of elliptic curve arithmetic on an ARM micro processor, we show that elliptic curve cryptography is a very competitive alternative to traditional cryptosystems for mobile platforms. We also show how to select parameters for elliptic curves in order to ensure the security of the cryptosystem. An efficient parameter selection algorithm enables on-the-fly curve genereation and still higher security.

Keywords: Elliptic curves, cryptography, fast arithmetic, point counting

AMS 2000 subject classification: 11T71, 14G50, 14H52, 68P25, 94A60

About ECMI

This licentiate thesis concludes a five semester ECMI programme in applied mathematics. This programme includes a block of core courses covering several areas of applied mathematics and computing science and a block of specialization courses within a selected field. The final part of the programme is to work with a mathematical problem that emanates from the industry. The aims of the European Consortium for Mathematics in Industry are:

- To promote the use of mathematical models in industry.
- To educate *industrial mathematicians* to meet the growing demands for such experts.
- To operate on a European Scale.

Till Erika

Contents

Chapter 1

Introduction

In this thesis, we will discuss elliptic curve cryptography and its implementations. This is an important topic with many applications in modern communication where bandwidth, storage resources and computational power is constrained and strong security desired. The specific application we have in mind are mobile phones using the Wireless Application Protocol (WAP). Mobile phones suffer all the limitations previously mentioned, and an efficient cryptographic protocol and its implementation is crucial for usability.

Our presentation is outlined as follows:

- In the first chapter, we begin by giving a brief introduction to cryptography and elliptic curves. We also discuss the computer environments we are likely to encounter. Furthermore, we give some specifics regarding elliptic curve cryptography protocols.
- In the second chapter, we treat arithmetic over extension fields of characteristic two, and how it can be implemented efficiently.
- In chapter three, we turn to the elliptic curve group and address the issue of implementing arithmetic efficiently in this group.
- Lastly, in chapter four, we do an in-depth study of how to count the points on an elliptic curve, which is an important problem when constructing a secure elliptic curve cryptosystem.

1.1 Introduction to Cryptography

The area of cryptography is a broad one, hence we will only consider a small part of it. It is common to group the goals of cryptography into the four categories; Confidentiality, Data integrity, Authentication and Non-repudiation. Of these, we will concentrate on confidentiality—means to obstruct unauthorized parties from reading information. For an excellent introduction to cryptography, refer to [13].

When two entities wish to communicate secretly, they can adopt many different strategies. For instance, they may avoid their adversaries and meet in private, they may send their messages in a manner that evades detection or they may send messages that, hopefully, only they themselves can read. We will only consider the latter of these methods.

1.1.1 Mathematical background

For our purposes, it will be convenient to consider information to be made up by elements from some set $\mathcal{A}$, called the alphabet. Elements of $\mathcal{A}$ can be combined into a message, which in turn is an element of the message space $\mathcal{M}$, consisting of possible, or allowed strings of elements from $\mathcal{A}$. When enciphering a message, we use a function $\mathcal{M} \to \mathcal{C}$, where $\mathcal{C}$ is the ciphertext space, whose elements are strings of some alphabet, which may or may not be the same as $\mathcal{A}$. Thus, we can view the procedure of transmitting enciphered information as

$$\begin{aligned} \text{Sender:} \qquad & f_e(m) = c, \quad m \in \mathcal{M},\ c \in \mathcal{C}, \\ \text{Receiver:} \qquad & g_d(c) = m, \quad m \in \mathcal{M},\ c \in \mathcal{C}, \end{aligned}$$

where the indices $e, d \in \mathcal{K}$ are keys. For each $e, d \in \mathcal{K}$, there exist precisely one pair $f_e : \mathcal{M} \to \mathcal{C}$ and one $g_d : \mathcal{C} \to \mathcal{M}$ such that the relationship above holds. If finding e knowing d, or vice versa, is easy, we say that we have a symmetric-key cipher. If finding e knowing d, or vice versa, is difficult, we say that we have a public-key cipher, or an assymetric-key cipher.

Example (Symmetric-key cipher). Let $\mathcal{A} = \{$A, B, C, …, X, Y, Z, Å, Ä, Ö$\}$ be the Swedish alphabet, and let $\mathcal{M}$ and $\mathcal{C}$ be the set of all strings of letters. Moreover, let $\mathcal{K} = \mathbb{Z}$, and f_e, g_d be permutations that shift the alphabet e (d) steps to the right (left). E.g. with $e = 3$, f_e(A) =D, f_e(B) =E, $\ldots$, f_e(Z) =Ö, f_e(Å) =A, $\ldots$. Then there is a simple relationship between e and d, namely $d = -e$, so this cryptosystem is a symmetric-key cipher. This cipher is known as the Caesar cipher, all though the emperor himself only used one key, $e = 3$ ($d = -3$).

Our problem is to find a suitable function f so that only designated receivers can reconstruct the plaintext message m from c. To this end, we introduce the following.

Definition (Trapdoor one-way function). A trapdoor one-way function $f : A \to B$ is a function such that for each $a \in A$ it is easy to compute $f(a) \in B$, but unless given some extra information, it is almost infeasible to compute $a \in A$ so that $f(a) = b$ given $b \in B$. However, given the extra, trapdoor, information, it should be easy.

As an aid in our search for trapdoor one-way functions we will briefly study three important and difficult problems. The first is a very famous problem that, however, is not related to the elliptic curve group, our main topic which will be introduced later. The other two problems can, and in order to see the similarities with the elliptic curve group, we use additive notation. Especially, we use the notation $[k]g = g + g + \cdots + g$ (k times).

The integer factorization problem

We start out with the integer factorization problem. This is the problem of finding the prime factorization of a non-zero positive integer n, that is to write $n = p_1^{m_1} p_2^{m_2} \cdots p_k^{m_k}$, where $p_1, \ldots, p_k$ are different primes and $m_1, \ldots m_k$ are non-zero positive integers. There exist a plethora of algorithms to solve this, among which the (general) number field sieve appears to be the fastest for big numbers, see e.g. [2].

The discrete logarithm problem

Let G be a finite, cyclic group, and g its generator. The discrete logarithm problem is then, given G, g and an element $h \in G$, to find an integer n such that $[n]g = h$.

This appears to be an easy problem, consider for instance $G = \mathbb{Z}_p$ (with addition), where p is a prime. Then it is a simple matter to find n such that $na = b$, where $a, b \in \mathbb{Z}_p$. So, for the problem to be a difficult one, we must choose the cyclic group G so that the isomorphism $\varphi : G \to \mathbb{Z}_p$ is hard to compute.

The Diffie-Hellman problem

As in the discrete logarithm problem, let G be a finite cyclic group, and g its generator. The Diffie-Hellman problem is then, given two elements $[m]g$ and $[n]g$ in G, to find $[mn]g$.

Computationally, the Diffie-Hellman problem is believed to be as hard to solve as the discrete logarithm problem. It is however evident that if it is easy to solve the discrete logarithm problem, then it is no harder to solve the Diffie-Hellman problem. Nevertheless, it is interesting in another regard—it provides us with a nice basis for a simple cryptographic protocol. More on this in the next section.

1.1.2 Some public-key cryptosystems

We will now give some examples of cryptosystems that utilize the problems given in the previous section. To get started, we begin with a simple example.

Example (Diffie-Hellman key agreement). The Diffie-Hellman key agreement is a scheme for two people to share a common secret message.

1. *Setup.* The sender A and the receiver B agree on a finite cyclic group G, and a generator g. Separately and secretly, they each decide on a number, a and b respectively. The entity $(G, g, [a]g)$ is A's public key, and similarly for B. Here, A's private key is a, B's private key is b.

2. *Transmission.* A computes $[a]g$ and sends this to B, while B computes $[b]g$ and sends this to A.

3. Their common key, that is their common secret, is $[ab]g$.

Remember that in order for a third party to find the secret message, the Diffie-Hellman problem has to be solved. Now, it might not appear to be very useful to share a secret message that is not known beforehand, but this message might be used, in a specified manner, to construct a key for a symmetric-key cipher. In this case, it doesn't matter what the secret key is, as long as it is secret.

We go on with an actual encryption scheme, the first that was shown to be provably secure, which is to say that it has been proven to be as difficult to break as its underlying mathematical problem (unlike for instance the more well-known RSA cipher).

Example (Rabin public-key encryption). In this scheme, we have two companions, A and B, where B wants to send A a secret message m.

1. *Setup.* A chooses two large random primes, $p \equiv 3 \mod 4$ and $q \equiv 3 \mod 4$, of about similar, large sizes, and computes $n = pq$. A's public key is n, while her private key is (p, q).

2. *Encryption.* To send m to A, B computes and sends $c = m^2 \mod n$, where m is represented among $\{0, 1, \ldots, n-1\}$.

3. *Decryption.* To recover m from c, A uses (p, q) to find the four square roots m_1, m_2, m_3 and m_4 of c modulo n. They are $\pm x, \pm y \mod n$, if $r = c^{(p+1)/4} \mod p$, $s = c^{(q+1)/4} \mod q$, $x = (aps + bqr) \mod n$, $y = (aps - bqr) \mod n$, where $ap + bq = 1$. Then $m = m_i$ for one of the four m_i's.

Breaking the Rabin cryptosystem (passively, that is without interfering with the transmissions) is as difficult as factoring n, see for instance [13].

Next, we give an example where the Diffie-Hellman problem enters a cryptosystem.

Example (ElGamal public-key encryption). As in the previous example, B wishes to send A a secret message m. Unlike the previous example, we are not confined to a specific group (as long as the Diffie-Hellman problem is difficult enough to solve in the group).

1. *Setup.* A finite cyclic group G of order p, where p is a prime, and a generator g is agreed upon (in public). A picks a random integer a, and computes $[a]g$. Her public key is $(G, g, [a]g)$, and her private key is a.

2. *Encryption.* B now picks a random integer k, computes $\gamma = [k]g$, $\delta = m + [k]([a]g)$ and sends (γ, δ) to A.

3. *Decryption.* Upon reception, A computes $m = \delta + [-a]\gamma$.

It is plain to see that this scheme relies on the Diffie-Hellman problem (and the discrete logarithm problem in extension).

Finally, we address the issue of signatures. If A sends a message to B, she can append a signature which allows B to verify that A was indeed the sender. First, we assume that a digest of the message has been computed, using for instance the Secure Hash Algorithm [15]. A message digest is a representative of the message computed in such a way that it should be very difficult for two different messages to have the same digests.

Example (ElGamal signature scheme). Here we assume that we have a finite group G of order p, where p is a prime, and a hash function $h : \{0,1\}^* \to \mathbb{Z}_p^*$, where $\{0,1\}^*$ denotes the set of all finite strings consisting of 0 and 1. We also assume that we have a function $f : G \to \{0,1\}^*$. In the scheme, A wants to send the message m to B who in turn wants to verify that A actually sent the message.

1. *Setup.* A finite cyclic group G of order p, where p is prime, with generator g is selected. The sender A selects a private key a and a public key $[a]g$, with $1 \le a \le p-1$.

2. *Signature generation.* A selects a random integer k such that $\gcd(k, n) = 1$ and then computes $r = [k]g$ and $s = k^{-1}[h(m) - ah(f(r))] \mod p$. She then sends the signature (r, s) (along with the message m).

3. *Signature verification.* B computes $v_1 = [h(f(r))]([a]g) + [s]r$ and $v_2 = [h(m)]g$. The signature is valid if and only if $v_1 = v_2$.

As a note for the elliptic curve group, a good $f : G \to \{0,1\}^*$ is the projection $(x, y) \mapsto x$. In this case, with some modifications, the above algorithm is called ECDSA[1] [7] §7,10.

1.2 Introduction to Elliptic Curves

An important topic for us is that of elliptic curves. An elliptic curve E over a field K is the set of points $(x, y) \in \overline{K}^2$ that satisfies an equation

$$y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6, \tag{1.1}$$

where $a_i \in \overline{K}$, in conjunction with a 'point at infinity', which will be explained later. Here, $\overline{K}$ denotes the algebraic closure of K. We will denote this set of points $E(\overline{K})$, and when we want to emphasize that the coefficients of E are in K, we write E/K. The coefficients a_i have their indices due to historical reasons, and are standard throughout the literature.

By making the substitutions $x = X/Z$, $y = Y/Z$ and clearing the denominators, (1.1) becomes

$$Y^2Z + a_1XYZ + a_3YZ^2 = X^3 + a_2X^2Z + a_4XZ^2 + a_6Z^3. \tag{1.2}$$

We say that the equation has been rewritten into projective coordinates, as opposed to the original affine coordinates. Equation (1.2) is called the Weierstrass equation in the projective form (and (1.1) is the Weierstrass equation in the affine form). We denote its solutions by $[X : Y : Z]$, where two solutions are considered equal if they are proportional. The affine point (x, y) corresponds to the projective triplet $[X : Y : 1]$, which is equivalent to $[X/Y : 1 : 1/Y]$ when $Y \neq 0$. If $K = \mathbb{R}$, we can let $Y \to \infty$ and get the solution $[0 : 1 : 0]$, so this triplet may be understood as the direction of the y-axis. We denote this triplet $\mathcal{O}$, the point at infinity. This point does not exist in the (x, y)-coordinate system, but can be imagined to lie 'somewhere far away along the y-axis'.

More demands on $E(\overline{K})$ are in order, as the curve must be smooth. In other words, we require that there must be no point in $E(\overline{K})$ where both partial derivatives with respect to x and y are zero. That is

$$a_1y = 3x^2 + 2a_2y + a_4, \qquad \text{and} \qquad 2y + a_1x + a_3 = 0$$

must not hold for any $(x, y) \in E(\overline{K})$, see also Figure 1.1. In characteristic 2, this means that a_1 and a_3 cannot both be equal to zero. Non-singular curves with $a_1 = 0$ are called *supersingular*, whereas curves with $a_3 = 0$ are called *non-supersingular*. For cryptographic reasons, we will almost exclusively be interested in non-supersingular elliptic curves. The equation (1.2), that describes an elliptic curve can be rewritten without changing the form (1.1) using the substitution

$$x = u^2x' + r, \qquad y = u^3y' + u^2sx' + t, \tag{1.3}$$

where $u \in K^*$, $r, s, t \in K$. A substitution of this form is called twisting. In our favorite case, fields of characteristic 2, if $a_1 \neq 0$, the substitution

$$x = a_1^2x' + a_3/a_1, \qquad y = a_1^3y' + (a_1a_4 + a_3^2)/a_1^3$$

[1] In ECDSA, the signature is $(s, f(r))$, and to verify the signature, we compute $q = s^{-1} \mod n$ and $q_1 = q \cdot h(m) \mod n$, $h_2 = q \cdot f(r) \mod n$. Then we compute $r' = [h_1]g + [h_2]([a]g)$. The signature is valid if $f(r') = f(r)$.

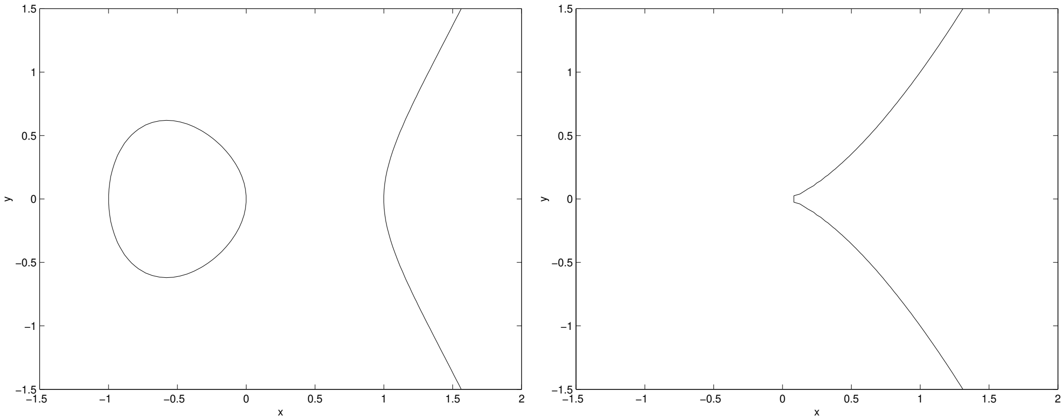

Figure 1.1: To the left is the non-singular elliptic curve $y^2 = x^3 - x$ over $\mathbb{R}$, and to the right is the singular curve $y^2 = x^3$, also over $\mathbb{R}$.

gives us an equation

$$y^2 + xy = x^3 + a_2x^2 + a_6. \tag{1.4}$$

Similarly, if $a_1 = 0$ (and $a_3 \neq 0$), the substitution $x = x' + a_2,\ y = y'$ gives an equation $y^2 + a_3y = x^3 + a_4x + a_6$.

For an elliptic curve E, we define the discriminant $\Delta(E)$ and the so-called j-invariant $j(E)$. In the case, when $K = \mathbb{F}_{2^n}$, and (1.4) is an equation for the elliptic curve, the discriminant $\Delta(E) = a_6$ and $j(E) = 1/a_6$. The j-invariant is invariant in the sense that two curves with the same j-invariant can be transformed into each other using the (1.3). It should be noted that all non-supersingular curves have a non-zero j-invariant, see [21], §III for more details.

1.2.1 The group law

The property that perhaps is the most important reason why elliptic curves are used in cryptography is that they can be supplied with a group structure turning the set of their points into a group. To see how it can be defined, let E be an elliptic curve over $\mathbb{R}$ for the remainder of this section unless otherwise stated.

Recall that the equation for an elliptic curve is a cubic in the x-coordinate. This means that a line in the plane intersects an elliptic curve at most three times. In fact, if the line is not a tangent to the curve nor vertical, it will intersect the curve exactly three times. This is the basis for the group law. Let $P = (x_P, y_P)$ and $Q = (x_Q, y_Q)$ be points on E.

1. If $P = \mathcal{O}$, the point at infinity, let $P + Q = \mathcal{O} + Q = Q$.

2. If $(x_P, y_P) = (x_Q, -x_Q)$, let $P + Q = \mathcal{O}$.

3. If $x_P \neq x_Q$, then let $-(P + Q)$ be the third point of intersection with E on the line through P and Q, see also the left graph in Figure 1.2.

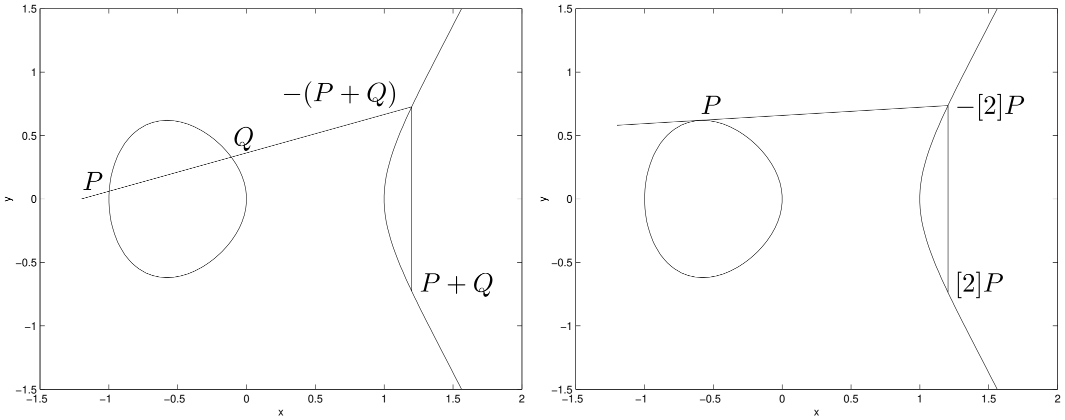

Figure 1.2: The group law on the elliptic curve $y^2 = x^3 - x$ over $\mathbb{R}$. To the left the addition law $P+Q$ is illustrated, to the right the doubling law $P+P = [2]P$.

4. If $P = Q$, then take the tangent line to the curve at P, and let $-(P+Q) = -[2]P = -[2]Q$ be the other point of intersection of this line with E, see also the right graph in Figure 1.2.

To derive formulae for these rules, let E have the equation

$$y^2 = x^3 + ax + b$$

(every elliptic curve over a field of characteristic not equal to 2 or 3 can be written in this form), and let $R = P + Q = (x_R, y_R)$. The line through P and Q ($P \neq Q$, case 3 above) is of the form $y = \alpha x + \beta$, where $\alpha = (y_P - y_Q)/(x_P - x_Q)$ and $\beta = y_P - \alpha x_P$. Now, a point on the line $(x, \alpha x + \beta)$ intersects E if $(\alpha x + \beta)^2 = x^3 + ax + b$, or $x^3 - (\alpha x + \beta)^2 + ax + b = 0$. We already now that $(x_P, \alpha x_P + \beta)$ and $(x_Q, \alpha x_Q + \beta)$ solve this cubic, which means that $\alpha^2 = x_P + x_Q + x_R$, or $x_R = \alpha^2 - x_P - x_Q$, so

$$\begin{aligned} x_R &= \alpha^2 - x_P - x_Q = \left(\frac{y_P - y_Q}{x_P - x_Q}\right)^2 - x_P - x_Q, \\ y_R &= -(\alpha x_R + \beta) = \left(\frac{y_P - y_Q}{x_P - x_Q}\right)(x_P - x_R) - y_P. \end{aligned}$$

In the fourth case, if $P = Q$, we take α as dy/dx of E at P. Taking the implicit derivative gives $\alpha = dy/dx = (3x_P^2 + a)/2y_P$, so

$$\begin{aligned} x_R &= \alpha^2 - 2x_P = \left(\frac{3x_P + a}{2y_P}\right)^2 - 2x_P, \\ y_R &= -(\alpha x_P + \beta) = \left(\frac{3x_P + a}{2y_P}\right)(x_P - x_R) - y_P. \end{aligned}$$

We note, however, that the two formulae are only valid if $\operatorname{char} K \neq 2, 3$. If $P = Q$ we write $R = [2]P$, and call this computation a point doubling. Repeated additions and doublings give us $[k]P$, what we call scalar multiplication.

In characteristic two, the formulae for non-supersingular curves become

$$
\begin{aligned}
x_R &= \lambda^2 + \lambda + x_P + x_Q + a_2, \\
y_R &= \lambda[x_R + x_P] + x_R + y_P), \\
\lambda &= \begin{cases} \frac{y_P + y_Q}{x_P + x_Q}, & \text{if } P \neq R, \\ x_P + y_P/x_P, & \text{if } P = R. \end{cases}
\end{aligned}
$$

To find this, first repeat the process above for the general curve equation (1.1), and then let $a_1 = 1$, $a_3 = a_4 = 0$.

1.2.2 The number of points

In this section, we will introduce the notion of point counting, and give some basic properties. Further information is found in section 4.1. An excellent pre-1999 account for point counting can be found in [19]. A description of more recent advances can be found in Chapter 4.

By the number of K-rational points on an elliptic curve E, we mean the number of $(x, y) \in K^2$ that satisfy the elliptic curve equation. We will denote this number N, and sometimes $\#E(K)$.

If $E/\mathbb{F}_q$ is an elliptic curve, it is also defined over the extension $\mathbb{F}_{q^r}$, and if the number of points to E over $\mathbb{F}_q$ is N, then we denote by $N_r = \#E(\mathbb{F}_{q^r})$ the number of points of E over $\mathbb{F}_{q^r}$, where $N_1 = N$. Now, define the zeta-function

$$Z(E/\mathbb{F}_q; T) = e^{\sum N_r T^r / r},$$

where the sum is over all positive $r = 1, 2, \ldots$. This function turns out to have a surprisingly simple form:

Theorem (The Weil Conjecture for Elliptic Curves). *Let E be an elliptic curve defined over the field $\mathbb{F}_q$. Then there is an $a \in \mathbb{Z}$ such that*

$$Z(E/\mathbb{F}_q; T) = \frac{1 - aT + qT^2}{(1-T)(1-qT)}.$$

Moreover, $N = q + 1 - a$, and $a^2 \leq 4q$.

See [21], §V.2, for a proof and further information. We note that from this formula it is easy to obtain N_r if we know N (by derivating r times with respect to T). However, curves $E/\mathbb{F}_q$, where q is small, are considered to be cryptographically weak if q is small, and will be of no interest to us.

An important and immediate consequence of this result is the following bound for the number of points on an elliptic curve:

Theorem (Hasse's Theorem). *Let E be an elliptic curve defined over $\mathbb{F}_q$, and let N be the number of points in $E(\mathbb{F}_q)$. Then,*

$$|q + 1 - N| \leq 2\sqrt{q}$$

Counting all the points on an elliptic curve is, in principle, easy. Given $x \in K$, there can be either 0, 1 or 2 solutions to the curve equation, which is then a simple quadratic equation in y. For simplicity, let $K = \mathbb{F}_p$, where $p > 3$ so that the elliptic curve has the equation $y^2 = x^3 + ax + b$. Recall the Legendre symbol

$$\left(\frac{f}{p}\right) = \begin{cases} 1, & \text{if there is a } g \text{ such that } g^2 = f \mod p, \text{ and } p \not| f, \\ 0, & \text{if } p|f, \\ -1, & \text{if are no } g \text{ such that } g^2 = f \mod p. \end{cases}$$

This means that, given x, the number of solutions to the elliptic curve equation is

$$1 + \left(\frac{x^3 + ax + b}{p}\right),$$

or, the number of points on E over $\mathbb{F}_p$ is

$$1 + \sum_{x \in \mathbb{F}_p} \left[1 + \left(\frac{x^3 + ax + b}{p}\right)\right] = 1 + p + \sum_{x \in \mathbb{F}_p} \left(\frac{x^3 + ax + b}{p}\right).$$

However, this method is only practical for very small fields, as its running time is $O(p^{1+\varepsilon})$. Here, $\varepsilon > 0$ depends on the implementation of the arithmetics in $\mathbb{F}_p$. A naïve implementation gives $\varepsilon = 1$.

Without further knowledge, it is possible to devise an algorithm for point counting that is faster than the abovementioned at the expense of storage requirements. The algorithm described below, which is due to D. Shanks, is called the baby-step-giant-method.

Let E be an elliptic curve defined over some finite field $\mathbb{F}_q$, and let $P \neq \mathcal{O}$ be a randomly chosen point on E. Furthermore, let $s = \lceil \sqrt[4]{q} \rceil$[2], and compute the baby steps $P, [2]P, \ldots, [s]P$. Note that since given P, it is easy to compute $-P$, we actually know $2s + 1$ points, including the point at infinity. Now, compute $Q = [2s + 1]P$, $R = [q + 1]P$, and the giant steps $R \pm Q, R \pm [2]Q, \ldots, R \pm [t]Q$, where $t = \lceil 2\sqrt{q}/(2s+1) \rceil$. Since $(2s+1)(2t+1) > 4\sqrt{q}$, Hasse's theorem now tells us that for some $i \in \{0, \pm 1, \pm 2, \ldots, \pm t\}$ and $j \in \{0, \pm 1, \pm 2, \ldots, \pm s\}$, the equation $R + [i]Q = [j]P$ has a solution. Take now $m = q + 1 + (2s + 1)i - j$. Then $m \in (q + 1 - 2\sqrt{q}, q + 1 + 2\sqrt{q})$, and $[m]P = \mathcal{O}$. If there are two distinct integers m and m' in the interval $(q + 1 - 2\sqrt{q}, q + 1 + 2\sqrt{q})$ with $[m]P = \mathcal{O}$ and $[m']P = \mathcal{O}$, the algorithm fails and a new point P has to be selected and we must run through the algorithm again. The second time we can, however, make use of the information that we know that $m - m'$ is a divisor of $\#E(\mathbb{F}_q)$ to speed up the computations. The running time of this algorithm is $O(q^{1/4+\varepsilon})$.

This concludes our introduction to elliptic curves.

1.3 Efficiency and the Environment

Before we consider specific algorithms for implementing elliptic curve cryptography, we must be aware of the computer environment in which all computations take place.

[2]We denote by $\lceil x \rceil$ the smallest integer larger than or equal to x. Likewise, we denote by $\lfloor x \rfloor$ the largest integer smaller than or equal to x.

1.3.1 Memory structure

In a modern computer, there is a hierarchy of storage memory. For instance, a PC usually has a hard disk, primary memory, an on chip cache and two additional caches in the central processing unit (CPU). The standard rule is that the closer memory is to the processor, the faster it is.

The fastest memory is that inside the processor. First, and foremost, a processor has a number of registers, which is where the actual computations take place. At the next higher level, a modern processor has two cache memories, one for instructions to be executed, and one for recently used, or to be used, data. The second level cache resides outside of the processor, but usually runs at a higher speed than the ordinary memory, though it commonly is not as large.

Normally, if an instruction reads data from memory, that and adjacent data is read to the cache, usually a multiple of 2 bytes. Since several bytes of data is read, a first memory fetch is usually quite slow. Subsequent reads from that same memory region can, however, be processed very fast. Therefore it is important to keep data dense, aligned to the specific multiple of 2 that the system architecture uses, and avoid random accesses instead of sequential. The same applies for instructions; especially important are frequently used instructions, commonly residing inside loops.

1.3.2 The central processing unit

Where memory structure does not differ much between platforms except for sizes, the workings of the CPU do. In personal computers nowadays there are two varieties of processor families, reduced instruction set computers (RISC) and complex instruction set computers (CISC). As their names indicate, the two families differ in how many instructions are available and how complex these instructions are. Typically, RISC processors, such as the ones found in many modern mobile phones, have two types of instructions—moving data between the memory and the registers of the CPU, and doing calculations on data in the registers. In comparison, in a CISC processor, such as the ones found in many modern PC's, instructions can often manipulate data regardless of its location.

1.4 Elliptic Curve Cryptosystems

In Section 1.1.2, we gave four examples of public-key cryptosystems three of which are valid for any cyclic group. These three cryptosystems are in fact the most commonly used for the elliptic curve group, although of most importance are the Diffie-Hellman key agreement and the ElGamal signature scheme. The reason for this is, that compared to symmetric-key ciphers, public-key ciphers are computationally more demanding for the same security level. However, as symmetric-key ciphers require that both parties know the same secret key, public-key ciphers are generally used to transmit the secret key securely, after which the actual message transmission uses the symmetric-key cipher.

Why is it then, that we should use the elliptic curve group instead of an ordinary cyclic group such as $\mathbb{Z}_p^*$ which is easier to implement and use? The reason is, of course, security. The best known algorithm for solving the discrete logarithm problem in $\mathbb{Z}_p^*$ is, as indicated earlier, the number field sieve, which has a running time proportional to

$$L(p) = e^{(1.992+o(1))(\ln p)^{1/3}(\ln \ln p)^{2/3}},$$

where $o(1) \to 0$ as $p \to \infty$. This means that the number field sieve is sub-exponential. To solve the discrete logarithm in the elliptic curve group, however, no sub-exponential algorithm is known. The best general method to date is Pollard's ρ-method which has running time $O(0.88\sqrt{n})$ group operations, where n is the largest prime divisor of the number of points on the elliptic curve. It should be noted that there are certain classes of curves for which there are sub-exponential algorithms, curves which should therefore be avoided, see Section 4.2.

In practical terms, these differences in running time, give implication in how large we need the groups to be, i.e. the key sizes. If we want our information to be secure for, say 20 years, until 2022, we should select an elliptic curve defined over $\mathbb{F}_q$ where $\log_2 q \approx 164$ [9], which means that our public key can be represented using a number which is about 165 bits (we need only store the x-coordinate and one extra bit which tells which of the two y-coordinates we use).

For a discrete logarithm cryptosystem over $\mathbb{Z}_p^*$, we'll need about 2000 bits to achieve the same security. This is a considerable difference in terms of storage space, the arithmetic and transmission costs, and is the main reason why elliptic curve cryptography is interesting today.

Chapter 2

Efficient Arithmetics in $\mathbb{F}_{2^n}$

In this chapter, we will discuss how to perform the basic operations in $\mathbb{F}_{2^n}$ efficiently. As mentioned previously, what is most efficient will depend heavily on the computer platform, but in this chapter no assumptions will be made regarding which computer platform is used. We will start with a discussion of different ways to represent the elements of $\mathbb{F}_{2^n}$ since the algorithms for addition/subtraction, multiplication, inversion and exponentiation will differ due to the representation. Only thereafter will the specific algorithms be put under scrutiny.

2.1 Field Representation

We first turn our attention to the question 'How do we represent an element $a \in \mathbb{F}_{2^n}$ in a computer so that we can perform efficient computations with it?'

2.1.1 Polynomial bases

Using a polynomial basis is the classical way of handling an extension field such as $\mathbb{F}_{2^n}$. In this case, we write $\mathbb{F}_{2^n}$as $\mathbb{F}_2[X]/(p(x))$, where $p(x)$ is an irreducible polynomial of degree n, so that each $a \in \mathbb{F}_{2^n}$ is a polynomial modulo $p(x)$ with binary coefficients, i.e. $a \in \mathbb{F}_{2^n}$ is written as

$$a = a_0 + a_1x + \cdots + a_{n-1}x^{n-1}, \qquad a_i \in \{0, 1\}$$

which using the basis $\{1, x, x^2, \ldots, x^{n-1}\}$ can be represented as the vector $a = (a_0, a_1, a_2, \ldots, a_{n-1})$. In a computer environment, this means that we let each a_i be one bit, with, preferably a_0 the least significant. The coefficients are then grouped into groups of, usually, 32 or 64 depending on the computer architecture.

The polynomial representation lends itself to computations rather straightforward manner. We now how to add, multiply and divide two polynomials from high school, but maybe not how to do it quickly. From experience, we know that multiplication and division will be the most cumbersome operations. Addition is simple, since all coefficients are modulo 2, and can be implemented with the logical exclusive-or operation. When multiplying and dividing, since we calculate modulo $p(x)$, care has to be taken to choose the reduction polynomial $p(x)$ so that these operations can be performed quickly. In practice this means that the weight of $p(x)$ should be small, preferably a trinomial.

2.1.2 Normal bases

In a normal basis, we start with an irreducible polynomial $p(x)$, and one of its roots $\theta \in \mathbb{F}_{2^n}$. The basis is of the form

$$\{\theta, \theta^2, \theta^{2^2}, \theta^{2^3}, \ldots, \theta^{2^{n-1}}\},$$

where all the elements of the basis satisfy $p(\theta^{2^i}) = 0$. We see that squaring becomes very simple—just shift your vector $a = (a_0, a_1, \ldots, a_{n-1})$, where again $a_i \in \{0, 1\}$, one step to the left, whereas multiplication and inversion is more difficult. There are two special cases of normal bases which are worth mentioning, both are called Optimal Normal Bases (ONB).

Optimal Normal Bases

For certain n, we can construct a normal basis that yields faster multiplication and inversion. This happens when for all $0 \leq i_1 \neq i_2 \leq n-1$ there exist j_1, j_2 such that $\theta^{2^{i_1}+2^{i_2}} = \theta^{2^{j_1}} + \theta^{2^{j_2}}$. An ONB can be constructed when

Type I $n+1$ is a prime p and 2 is a primitive root modulo p.

Type II $2n+1$ is a prime p and either

1. 2 is primitive modulo p, or
2. $p \equiv 3 \pmod 4$, and the multiplicative order of 2 modulo p is n.

Type I ONB's are attractive since then θ is a root of $p(x) = x^n + x^{n-1} + \cdots + x + 1$, and it is possible to treat elements as polynomials modulo $(x+1)f(x) = x^{n+1} + 1$. A change of bases between the polynomial basis and the normal basis can be performed using $1 = \theta + \theta^2 + \cdots + \theta^n$.

The type II representation can also be given similar characteristics. In this case, we can write $\theta = \gamma + \gamma^{-1}$, where γ is a pth root of unity in $\mathbb{F}_{2^n}$. It is now possible to calculate using polynomials if we take the polynomials to be

$$a(x) = \sum_{i=1}^{2n} a_i x^i,$$

where $a_i = a_{p-i}$ (a so called palindromic polynomial), and multiplication is carried out modulo $x^p - 1$. A change of bases to the ONB is then done through $a(\gamma) = \sum_{j=0}^{n-1} a_{2^j} \theta^{2^j}$. See also [6].

2.1.3 Subfield bases

A special case of $\mathbb{F}_{2^n}$ is when $n = n_1 n_2$ where $n_1, n_2 \geq 1$, and n_1 is rather small, say 16 or below. It is then possible to view $\mathbb{F}_{2^n}$ as $\mathbb{F}_{q^{n_2}}$, where $q = 2^{n_1}$, and the elements of $\mathbb{F}_{2^n}$ are polynomials with coefficients in $\mathbb{F}_{n_1}$. The operations in the subfield can be tabulated and computations severely sped up. However, we will not concern ourselves with these, as elliptic curve cryptosystems built on such fields have been shown to be weak, see Section 4.2.

2.2 Multiplication

We now move on to the problem of computing $c = ab$, when $a, b \in \mathbb{F}_{2^n}$. Obviously, our tactics will differ depending on the representation, and we will confine ourselves to only study polynomial bases, as experience has shown these to be most efficient.

2.2.1 Schoolbook multiplication

Given $a(x) = \sum_{i=0}^{n-1} a_i x^i,\ b(x) = \sum_{i=0}^{n-1} b_i x^i$, we can calculate $c(x) = a(x)b(x)$ through the double sum

$$c(x) = \sum_{i=0}^{2n-2} \left(\sum_{k+l=i} a_k b_l \right) x^i,$$

that is $c_i = \sum_{k=0}^{i} a_k b_{i-k}$, where only indices satisfying $0 \le i, i-k < n$ are included. This procedure is summarized in Algorithm 1. Note that we don't do the required reduction modulo $p(x)$.

Algorithm 1 Schoolbook multiplication

Require: $a(x) = \sum_0^{n-1} a_i x^i$, $b(x) = \sum_0^{n-1} b_i x^i$
Ensure: $c(x) = a(x)b(x)$
  for $i = 0$ to $2n-2$ **do**
    for $k = 0$ to i **do**
      if $0 \le i-k, k < n$ **then**
        $c_i \leftarrow c_i + a_k b_{i-k}$
      end if
    end for
  end for

2.2.2 Comb methods

One idea that turns out to be useful when multiplying is to use previous results whenever possible. First, we write a as a vector, $a = (a_{n-1}, a_{n-2}, \ldots, a_1, a_0)$, and write its division into W-tuplets as $A = (A[k], A[k-1], \ldots, A[0])$, where $A[i] = (a_{W(i+1)-1}, \ldots, a_{Wi})$ is called a word. Here we chose W in order to maximize speed, which is dependent on the computer architecture. Usually we take W to be equal to 32. We also define $A\{j\} = (A[k], \ldots, A[j+1], A[j])$ (thus $A = A\{0\}$).

First, we notice that we can write the polynomial product as

$$a(x)b(x) = \sum_{i=0}^{n-1} b(x) a_i x^i = x^{n-1} a_{n-1} b(x) + \cdots + x a_1 b(x) + a_0 b(x)$$

The idea behind the comb methods is based on the observation that if we have computed $b(x) \cdot x^k$ for some k, then it is very easy to compute $b(x) \cdot x^{Wj+k}$ by simply appending 0-words at the end of the vector representing $b(x) \cdot x^k$. This is equivalent with writing the product as

$$a(x)b(x) = \sum_{k=0}^{W-1} \left(\sum_{j=0}^{\lfloor n/W \rfloor} x^{Wj+k} a_{Wj+k} b(x) \right),$$

where the inner sum can be calculated quickly, as before mentioned. An example might be in order: let $a = 010|110$, $b = 001|101$ (so $c = 011|111|110$) ('|' indicates $W = 3$ separation). Now

$$\begin{aligned} c =& (0 \cdot x^0 + 0 \cdot x^{0+3}) + (b(x) \cdot x + b(x) \cdot x^{1+3}) + (b(x) \cdot x^2 + 0 \cdot x^{2+3}) = \\ =& 000|011|010 \\ &+011|010|000 \\ &+000|110|100 = 011|111|110. \end{aligned}$$

The comb methods can be implemented in several ways. Algorithm 2 is the one outlined in the example above, whereas Algorithm 3 is a slight modification.

Algorithm 2 The right-to-left comb method

Require: $a(x) = \sum_0^{n-1} a_i x^i$, $b(x) = \sum_0^{n-1} b_i x^i$
Ensure: $c(x) = a(x)b(x)$
  $C \leftarrow 0$
  for $k = 0$ to $W - 1$ **do**
    for $j = 0$ to $\lfloor n/W \rfloor$ **do**
      if the kth bit of $A[j]$ is 1 **then**
        $C\{j\} \leftarrow C\{j\} + B$
      end if
    end for
    if $k \neq 31$ **then**
      $B \leftarrow B \cdot x$
    end if
  end for

We notice that Algorithm 2 is faster than Algorithm 3, since in the latter, C is shifted instead of B, where C is twice as large as B. But, we do include Algorithm 3 due to the fact that it may be sped up using some overhead storage. If $w|W$, we can pre-compute $u(x)b(x)$ for all polynomials $u(x)$ of degree less than w, put them in a table, and shorten the outer loop. This is done in Algorithm 4.

2.2.3 Karatsuba multiplication and relatives

The multiplication methods described above all have in common that they require $O(n^2)$ multiplications (of coefficients or by x). That is not a necessary trait, as we will now show. When multiplying two first degree polynomials, the ordinary way, we get

$$a(x)b(x) = (a_0 + a_1 x)(b_0 + b_1 x) = a_0 b_0 + (a_0 b_1 + a_1 b_0)x + a_1 b_1 x^2,$$

Algorithm 3 The left-to-right comb method

Require: $a(x) = \sum_0^{n-1} a_i x^i$, $b(x) = \sum_0^{n-1} b_i x^i$
Ensure: $c(x) = a(x)b(x)$
  $C \leftarrow 0$
  for $k = W - 1$ downto 0 **do**
    for $j = 0$ to $\lfloor n/W \rfloor$ **do**
      if the kth bit of $A[j]$ is 1 **then**
        $C\{j\} \leftarrow C\{j\} + B$
      end if
    end for
    if $k \neq 0$ **then**
      $C \leftarrow C \cdot x$
    end if
  end for

Algorithm 4 The left-to-right comb method with windows

Require: $a(x) = \sum_0^{n-1} a_i x^i$, $b(x) = \sum_0^{n-1} b_i x^i$
Ensure: $c(x) = a(x)b(x)$
  Pre-compute $B_u = u(x)b(x)$ for all polynomials $u(x)$ of degree less than or equal to $w - 1$
  $C \leftarrow 0$
  for $k = W/w - 1$ downto 0 **do**
    for $j = 0$ to $\lfloor n/W \rfloor$ **do**
      Let $u = (u_3, u_2, u_1, u_0)$ where u_i is bit $(wk + i)$ of $A[j]$.
      $C\{j\} \leftarrow C\{j\} + B_u$.
    end for
    if $k \neq 0$ **then**
      $C \leftarrow C \cdot x^w$
    end if
  end for

which requires 4 multiplications and 1 addition. If, instead, we do the simple observation (first noted by Karatsuba in the late 60's, see e.g. [8]) that $a_0b_1 + a_1b_0 = (a_0 + a_1)(b_0 + b_1) - a_0b_0 - a_1b_1$ the expression for the product becomes

$$a(x)b(x) = (a_0 + a_1x)(b_0 + b_1x) = a_0b_0 + [(a_0 + a_1)(b_0 + b_1) - a_0b_0 - a_1b_1]x + a_1b_1x^2,$$

which only requires 3 multiplications (on the behalf of 4 additions). This might not look surprisingly astounding, but if we apply this scheme recursively, we find that multiplication will only require $O(n^{4/3})$ short multiplications. To see how this can be used, define the multiplication of $A,\ B$ as

$$\begin{aligned}
A \cdot_K B :=& [A[0] + A[1]x^W + \cdots + A[t/2-1]x^{W(t/2-1)} + (A[t/2] + \cdots + A[t-1]x^{W(t/2-1)})x^{Wt/2}] \times \\
& [B[0] + B[1]x + \cdots + B[t/2-1]x^{W(t/2-1)} + (B[t/2] + \cdots + B[t-1]x^{W(t/2-1)})x^{Wt/2}] = \\
=& (A_l + A_h x^{Wt/2}) \cdot (B_l + B_h x^{Wt/2}) = \\
=& A_l \cdot_K B_l - [(A_l - A_h) \cdot_K (B_l - B_h) - A_l \cdot_K B_l - A_h \cdot_K B_h]x^{Wt/2} + \\
& A_h \cdot_K B_h x^{Wt},
\end{aligned}$$

(where $\cdot_K$ denotes a Karatsuba multiplication step), and $A_{l,h},\ B_{l,h}$ are the lower and upper halves of A and B respectively. Thus, we assume that $A,\ B$ are t-vectors and $A_{l,h},\ B_{l,h}$ are $t/2$-vectors. How this works out in practice is illustrated in Algorithm 5.

Algorithm 5 Multiply two multi-digit numbers using Karatsuba's algorithm

Require: $A = \sum_0^{k-1} a_i x^i$, $B = \sum_0^{k-1} b_i x^i$, $k = 2^n$
Ensure: AB
  $A_l \leftarrow \sum_0^{k/2} a_i x^i$, $A_h \leftarrow \sum_{k/2+1}^{n} a_i x^i$
  $B_l \leftarrow \sum_0^{k/2} b_i x^i$, $B_h \leftarrow \sum_{k/2+1}^{m} b_i x^i$
  $C \leftarrow$ Karatsuba(A_l, B_l)
  $D \leftarrow$ Karatsuba$(A_l + A_h, B_l + B_h)$
  $E \leftarrow$ Karatsuba(A_h, B_h)
  $AB \leftarrow C + (D - C - E)x^{k/2} + Ex^k$

In exactly the same spirit as Karatsuba, it is possible to hand-tailor formulae which divide A and B into 3 or 4 parts (and so on), which might be useful if a software is to be optimized for a certain n (e.g. numbers when n is between 160 and 190 can be stored in 6 32-bit words). But, it is also possible to use other methods to achieve a greater asymptotical speed-up such as the FFT-method. For our purposes, however, the overhead cost of such methods will render them not worthwhile to pursue unless n is large enough, which depends on the implementation.

2.2.4 Modular reduction

In all the algorithms presented above, we discretely avoided to reduce the results modulo $p(x)$, something that is absolutely necessary. Since we can rewrite $p(x)$ as $p(x) = x^n + r(x)$, we see that when reducing x^i in $c(x) = a(x)b(x)$, we can, when $n \le i \le 2n-2$, substitute $x^i = x^{i-n}r(x)$. This observation gives us Algorithm 6.

Algorithm 6 Trivial modular reduction

Require: $c(x) = \sum_0^{2n-2} c_i x^i$, $p(x) = x^n + r(x)$
Ensure: $c(x) \pmod{p(x)}$
  for i from $2n-2$ to n **do**
    if $c_i = 1$ **then**
      $c_i \leftarrow 0$
      $c(x) \leftarrow c(x) + x^{n-i} r(x)$
    end if
  end for

We note three things about this algorithm: Firstly, the lower the degree of $r(x)$ (or if the terms are close to each other), the better. Secondly, the lower the weight of $r(x)$ the better and lastly, it might be worthwhile to precompute $x^j r(x)$ for $0 \leq j \leq 31$.

It is possible to speed this algorithm up, by working with groups of W coefficients at a time, where W is typically 32. To see how this can be done, consider the substitution $x^a \to x^{a-l}$ for W a's, with $l = Wm + n$, where $0 \leq n < W$:

$$\begin{aligned} x^{Wk+W-1} &\to x^{W-1} x^{W(k-m)-n}, \\ x^{Wk+W-2} &\to x^{W-2} x^{W(k-m)-n}, \\ &\vdots \\ x^{Wk} &\to x^0 x^{W(k-m)-n}. \end{aligned}$$

Since $W - i - n = 0$ for some i, this can be rewritten as

$$\begin{aligned} x^{Wk+W-1} &\to x^{W-1} x^{W(k-m)-n}, \\ x^{Wk+W-2} &\to x^{W-2} x^{W(k-m)-n}, \\ &\vdots \\ x^{Wk+W-i} &\to x^{W-i} x^{W(k-m-1)+(W-n)}, \\ x^{Wk+W-i-1} &\to x^{W-i-1} x^{W(k-m-1)+(W-n)}, \\ &\vdots \\ x^{Wk} &\to x^0 x^{W(k-m-1)+(W-n)}. \end{aligned} \tag{2.1}$$

This means that in a computer, where usually $W = 32$, we can implement this substitution using a bit-shift of n to the right or $W - n$ to the left. The reason for the division at i is that a right shift larger than W becomes zero in the computer.

For a full modular reduction, write the reduction polynomial as $p(x) = x^{p_1} + \cdots + x^{p_k}$, let $l_i = p_1 - p_{i+1}$ for $i = 1, \ldots, k-1$ and write $l_i = Wm_i + n_i$ with $0 \leq n_i < W$ and do the substitution (2.1) $k - 1$ times. For details, see Algorithm 7.

2.3 Squaring

The problem of squaring is to calculate $a^2 = a \cdot a$. In characteristic two, squaring is particularly simple as $(x+y)^2 = x^2 + y^2$, giving $a^2(x) = \sum a_i x^{2i}$. Thus we have to insert a 0 between

Algorithm 7 Windowed modular reduction

Require: $c(x) = \sum_0^{2n-1} c_i x^i$, $p(x) = x^{p_1} + \cdots + x^{p_l}$, a window size $W = 2^d$, usually 32.
Ensure: $c(x) \pmod{p(x)}$
  Precomputation. Let $Wj + n_{j,i} = p_1 - p_{i+1}$, $0 \leq n_{j,i} < W$ and $o_{j+1,i} = 32 - n_{j,i}$. Let $d_j = \#\{i : n_{j,i}$ have been defined$\}$ and $J = \{j : d_j > 0\}$. Further, let $M_2 = 2^k - 1$, where k is the smallest k s.t. $2^k - 1 \geq [p_1\ AND\ (W-1)]$ and $M_1 = NOT\ M_2$ be bit-masks. As before, denote by $C[i] = (c_{Wi+W-1}, \ldots, c_{Wi+1}, c_{Wi})$.
  for i from $\lceil[(2n-1)/W\rceil$ to $\lceil n_1 \rceil$ **do**
    $A \leftarrow C[i]$
    for $j \in J$ **do**
      $C[i-j] \leftarrow C[i-j] + \sum_k (A >> n_{j,k}) + \sum_k (A << o_{j,k})$
    end for
  end for
  $A \leftarrow C[\lfloor n_1/W \rfloor]\ AND\ M_1$
  for $j \in J \cap \{j : \lfloor n_1/W \rfloor >= j\}$ **do**
    $C[i-j] \leftarrow C[i-j] + \sum_k (A >> n_{j,k}) + \sum_k (A << o_{j,k})$
  end for
  $C[\lfloor n_1 \rfloor] \leftarrow C[\lfloor n_1 \rfloor]\ AND\ M_2$

every bit in the representation of $a(x)$. If we allow ourselves some storage for a table, we can do this even faster, as in Algorithm 8.

Algorithm 8 Squaring ($W = 32$)

Require: $a(x) = \sum_0^{n-1} a_i x^i$
Ensure: $a^2(x)$
  Precompute say $b(x) = (0, x_3, 0, x_2, 0, x_1, 0, x_0)$ for every nibble $(x_4 x_3 x_2 x_1 x_0)$.
  for i from 0 to $\lfloor n/32 \rceil$ **do**
    Let $A[i] = (u_7, u_6, u_5, \ldots, u_0)$ where u_i is a nibble.
    $C[2i] \leftarrow (b(u_3), b(u_2), b(u_1), b(u_0))$, $C[2i+1] \leftarrow (b(u_7), b(u_6), b(u_5), b(u_4))$
  end for

2.4 Inversion

Next, we turn to the problem of inverting the elements of $\mathbb{F}_{2^n}$. Here too, we confine ourselves to the polynomial basis representation of $\mathbb{F}_{2^n}$.

2.4.1 The Extended Euclidean Algorithm

The traditional way of inverting a polynomial $f(x) \pmod{p(x)}$ is to use the Extended Euclidean Algorithm for polynomials, which gives us polynomials $g,\ h$ such that $f(x)g(x) + h(x)p(x) = 1$, where $g(x) = f(x)^{-1} \pmod{p(x)}$. This algorithm is on display in Algorithm 9.

Algorithm 9 Extended Euclidean Algorithm for inversion

Require: $a(x) = \sum_0^{n-1} a_i x^i$, $p(x)$
Ensure: $a(x)^{-1} \pmod{p(x)}$
  $b \leftarrow 1$, $c \leftarrow 0$, $v \leftarrow p$. $u \leftarrow a$
  while $\deg(u) \neq 0$ **do**
    $j \leftarrow \deg(u) - \deg(v)$
    if $j < 0$ **then**
      $u \leftrightarrow v$, $b \leftrightarrow c$, $j \leftrightarrow -j$
    end if
    $u \leftarrow u + x^j v$, $b \leftarrow b + x^j c$
  end while

2.4.2 The Almost-Inverse Algorithm

An alternative to the algorithm above is the Almost-Inverse Algorithm (AIA) [20]. Instead of returning $g,\ h$ such that $f(x)g(x) + h(x)p(x) = 1$, AIA returns $f,\ g$ such that $f(x)g(x) + h(x)p(x) = x^k \pmod{p(x)}$.

Algorithm 10 Almost-Inverse Algorithm

Require: $a(x) = \sum_0^{n-1} a_i x^i$, $p(x)$
Ensure: $b(x) = x^k a(x)^{-1} \pmod{p(x)}$
1: $b \leftarrow 1$, $c \leftarrow 0$, $v \leftarrow p$, $u \leftarrow a$
2: **while** x divides u **do**
3:   $u \leftarrow u/x$, $c \leftarrow cx$, $k \leftarrow k+1$
4: **end while**
5: **if** $u = 1$ **then**
6:   **return** (b, k)
7: **end if**
8: **if** $\deg(u) < \deg(v)$ **then**
9:   $u \leftrightarrow v$, $b \leftrightarrow c$
10: **end if**
11: $u \leftarrow u + v$, $b \leftarrow b + c$
12: **goto** step 2.

On the average, AIA is expected to need fewer iterations, as is evident from the while-loop. After the algorithm has been performed, we will need to reduce the result, which can be done as follows: Define s to be the smallest $i \geq 1$ such that $p_i = 1$, where $p(x) = \sum p_i x^i$, and let b be such that $b(x)a(x) = x^k$, and b' be the polynomial formed by the s rightmost bits of b. Then let $b'' = (b'f + b)/x^s$ (which is a non-rational function) and continue with $b \leftarrow b''$. We notice that if $s \geq W$, this process is faster and such reduction polynomials are more suited for AIA than others.

2.4.3 The Modified Almost-Inverse Algorithm

The AIA can be modified to the extent that the final reduction step is removed by integrating the reduction into the first while-loop. This is done in Algorithm 11.

Algorithm 11 Modified Almost-Inverse Algorithm

Require: $a(x) = \sum_0^{n-1} a_i x^i$, $p(x)$
Ensure: $a(x)^{-1} \pmod{p(x)}$
$b \leftarrow 1$, $c \leftarrow 0$, $v \leftarrow p$, $u \leftarrow a$
while x divides u **do**
  $u \leftarrow u/x$
  if x divides b **then**
    $b \leftarrow b/x$
  else
    $b \leftarrow (b+p)/x$
  end if
end while
if $u = 1$ **then**
  return (b)
end if
if $\deg(u) < \deg(v)$ **then**
  $u \leftrightarrow v$, $b \leftrightarrow c$
end if
$u \leftarrow u + v$, $b \leftarrow b + c$
goto step 2.

In comparison with AIA, MAIA is faster if s is small, but might be slower when s is large, so it depends on the reduction polynomial (and the implementation).

2.5 Summary

We give a few timings of an implementation made in assembler and C for the ARM 7TDMI processor in Table 2.1.

Routine	$\mathbb{F}_{2^{163}}$
Addition	12 μs
Multiplication (and reduction)	830 μs
Squaring (and reduction)	250 μs
Inversion (EEA)	9180 μs
Inversion (MAIA)	10710 μs
Windowed reduction	190 μs
Simple reduction	750 μs

Table 2.1: Timings of arithmetic routines for an 10MHz ARM 7TDMI processor (emulated) on $\mathbb{F}_{2^{163}}$.

We see from the table that inversion is more than 10 times slower than multiplication. In the implementation, all basic routines except for reduction were implemented in assembler, and no special optimizations were made with regard to the specific field $\mathbb{F}_{2^{163}}$. Also, execution speed on the ARM processor suffers when data is fetched and stored to memory, so squaring for instance would benefit with a larger look-up table than what was used (16 values stored), but it would require more storage space, which is not abundant.

Chapter 3

Efficient Elliptic Curve Arithmetics

In this chapter, we will discuss how to do calculations on elliptic curve groups, defined over a finite field of characteristic two, as efficiently as possible. We will also try to, when appropriate, link this discussion to that of the previous chapter, as they are dependent on each other. In this chapter we will always consider elliptic curves defined over $\mathbb{F}_{2^n}$.

3.1 Coordinate Representations

In this first section, we will look into various ways of representing the points of elliptic curves. As a rule, the (ordinary) affine coordinates will be denoted x and y, and rules will be given that transform the other variants back to these. In general, the different forms of projective coordinates (which use a third Z-coordinate) can be obtained by simply setting the extra coordinate values to 1.

3.1.1 Affine coordinates

We recall that in affine coordinates, the elliptic curve equation in characteristic two (with $j \neq 0$) may be given as

$$E: \quad y^2 + xy = x^3 + Bx^2 + A. \tag{3.1}$$

We also recall the group law algorithms in affine coordinates. Let $P_i = (x_i, y_i) \in E$. Then

$$\begin{aligned} -P_1 &= (x_1, x_1 + y_1), \\ P_3 = P_1 + P_2 = (x_3, y_3) &= (\lambda^2 + \lambda + x_1 + x_2 + B, \lambda[x_1 + x_3] + x_3 + y_1), \\ \lambda &= \begin{cases} \frac{y_2+y_1}{x_2+x_1}, & \text{if } P_1 \neq P_2, \\ x_1 + y_1/x_1, & \text{if } P_1 = P_2. \end{cases} \end{aligned}$$

Further, we can try to calculate the number of finite field operations that is needed to perform the computations above. When adding two distinct points, we thus need 9 additions, 2 multiplications, 1 squaring and 1 inversion to add two points, whereas a point doubling needs 8 additions, 2 multiplications and 1 inversion in $\mathbb{F}_{2^n}$. From now on, we will denote the number of operations needed for point doubling as $t_2(\text{affine}) \leq 9A + 2M + 1S + 1I$ and the number of operations needed for point addition as $t_+(\text{affine}) \leq 8A + 2M + 1I$ where A, M, S, I stands for addition, multiplication, squaring and inversion respectively.

Projective coordinates

As mentioned before, an other natural way of describing the points on the curve is to use the projective coordinates (X, Y, Z), where the point at infinity is represented as $(0, 1, 0)$. To transfer to projective coordinates, set $x = X/Z$ and $y = Y/Z$ to get the curve equation

$$E: \quad Y^2Z + XYZ = X^3 + BX^2Z + AZ^3. \tag{3.2}$$

We note that this will enable us to do curve calculations without any field inversions.

To find the addition and doubling formulae in these coordinates, we make the substitutions $x_i \leftarrow X_i/Z_i$ and $y_i \leftarrow Y_i/Z_i$ in the formulae for the affine coordinates. Then we try to simplify the formulae inserting temporary variables wherever possible to reduce the number of operations. For addition, we get

$$\begin{aligned} X_3 &= \lambda_2\lambda_3, \\ Y_3 &= X_1Z_2\lambda_1\lambda_2^2 + \lambda_1\lambda_3 + \lambda_2\lambda_3 + Y_1Z_2\lambda_2^2, \\ Z_3 &= Z_1Z_2\lambda_2^3, \end{aligned}$$

where

$$\begin{aligned} \lambda_1 &= \begin{cases} Y_1Z_2 + Y_2Z_1, & \text{if } P_1 \neq P_2, \\ X_1^2 + Y_1Z_1, & \text{if } P_1 = P_2, \end{cases} \\ \lambda_2 &= \begin{cases} X_1Z_2 + X_2Z_1, & \text{if } P_1 \neq P_2, \\ X_1Z_1, & \text{if } P_1 = P_2, \end{cases} \\ \lambda_3 &= Z_1Z_2\lambda_1^2 + Z_1Z_2\lambda_1\lambda_2 + \lambda_2^3 + BZ_1Z_2\lambda_2^2. \end{aligned}$$

This means that $t_+(\text{projective}) \leq 8A + 16M + 2S$ and $t_2(\text{projective}) \leq 7A + 14M + 3S$

Jacobian projective coordinates

Now, we start looking into 'unnatural' choices of coordinate substitutions that are merely constructs to enhance the speed of computation. The, historically, first and foremost of these were the Jacobian projective coordinates (X, Y, Z) which are found through the substitutions $x = X/Z^2$, $y = Y/Z^3$. In this case, the elliptic curve equation transforms into

$$E: \quad Y^2 + XYZ = X^3 + BX^2Z^2 + AZ^6. \tag{3.3}$$

We get the addition formula

$$\begin{array}{llllll} U_0 & = X_1Z_2^2, & S_1 & = Y_2Z_1^3, & Z_3 & = LZ_2, \\ S_0 & = Y_1Z_2^3, & R & = S_0 + S_1, & T & = R + Z_3, \\ U_1 & = X_2Z_1^2, & L & = Z_1W, & X_3 & = BZ_2^2 + TR + W^3, \\ W & = U_0 + U_1, & V & = RX_2 + LY_2, & Y_3 & = TX_3 + CL^2 \end{array}$$

and the doubling formula, where $C = A^{2^{n-2}}$

$$\begin{aligned} Z_2 &= X_1Z_1^2 \\ X_2 &= (X_1 + CZ_1^2)^4 \\ U &= Z_2 + X_1^2 + Y_1Z_1 \\ Y_2 &= X_1^4Z_2 + UX_2 \end{aligned}$$

These formulae yield $t_+(\text{jacobian}) \leq 7A + 15M + 5S$ and $t_2(\text{jacobian}) \leq 4A + 5M + 5S$. See also [7]. We conclude that these coordinates give faster formulae for both addition and multiplication than ordinary projective coordinates but, unlike [7] proclaims, there exists a faster algorithm as we will soon see.

Chudnovsky Jacobian coordinates

Another construct, due to Chudnovsky, reaches speed on the behalf of storage memory requirements. Here, we represent the points as the quintuplet (X, Y, Z, Z^2, Z^3), where, as in the ordinary Jacobian coordinates, $x = X/Z^2$, $y = Y/Z^3$. The speed-up versus Jacobian coordinates is, of course, due to the fact that we won't have to calculate the square or the cube of Z.

We don't list the formulae for addition and doubling here, as they are identical to those in the previous section, only we never need calculate Z_i^2, Z_i^3 for $i = 1, 2$. The gains are modest—we save $1M + 1S$ when adding but loose that same time when doubling, leading us to the conclusion that the Chudnovsky representation is appropriate when more additions than doublings are performed.

López-Dahab projective coordinates

The last set of coordinate representation we look into are due to López and Dahab [10]. They propose that we represent points as the triplets (X, Y, Z), where $x = X/Z,\ y = Y/Z^2$. In this case, the elliptic curve equation transforms into

$$E: \quad Y^2 + XYZ = X^3 + BX^2Z^2 + AZ^4. \tag{3.4}$$

The first fact used here is that, using the affine elliptic curve equation (Equation (3.1)), the affine doubling formula can be rewritten as

$$\begin{aligned}
x_2 &= \lambda^2 + \lambda + B = \left(\frac{x_1^2 + y_1}{x_1}\right)^2 + \frac{x_1^2 + y_1}{x_1} + B = \frac{x_1^4 + y_1^2 + x_1^3 + x_1 y_1}{x_1^2} + B = \\
&= \frac{x_1^4 + Bx_1^2 + A}{x_1^2} + B = x_1^2 + \frac{A}{x_1^2}, \\
y_2 &= x_1^2 + (x_1 + \frac{y_1}{x_1})x_2 + x_2.
\end{aligned}$$

(This reformulation means that two inversions are necessary, so it is rather slow.) Rewriting these formulae using the substitution into the López-Dahab coordinates yields the doubling formula

$$\begin{aligned}
Z_3 &= X_1^2 Z_1^2 \\
X_3 &= X_1^4 + AZ_1^4 \\
Y_3 &= AZ_1^4 Z_2 + X_2(BZ_2 + Y_1^2 + AZ_1^4),
\end{aligned}$$

where $t_2(\mathrm{ld}) \leq 4A + 5M + 5S$. The addition formula becomes

$$
\begin{aligned}
A_0 &= Y_2 \cdot Z_1^2, & D &= B_0 + B_1, & H &= C \cdot F, \\
A_1 &= Y_1 \cdot Z_2^2, & E &= Z_1 \cdot Z_2, & X_3 &= C^2 + H + G, \\
B_0 &= X_2 \cdot Z_1, & F &= D \cdot E, & I &= D^2 \cdot B_0 \cdot E + X_3, \\
B_1 &= X_1 \cdot Z_2, & Z_3 &= F^2, & J &= D^2 \cdot A_0 + X_3, \\
C &= A_0 + A_1, & G &= D^2 \cdot (F + B \cdot E^2), & Y_3 &= H \cdot I + Z_3 \cdot J,
\end{aligned}
$$

where $t_+(\mathrm{ld}) \leq 8A+14M+6S$. At the present time, López-Dahab coordinates give the fastest algorithm for addition and doubling on elliptic curves defined over fields of characteristic two.

3.1.2 Mixed coordinate systems

In certain cases, it is more efficient to represent P, Q and R in the summation $R = P + Q$ using different coordinate systems. It is especially interesting to study the case where R and P (or Q) are in projective coordinates and Q (or P) is in affine coordinates, as this is equivalent with writing $Q = (X_2, Y_2, 1)$, so we can just set $Z_2 = 1$ in the different projective addition formulae.

If Table 3.1 we list a summary of the findings for the different coordinate systems, and in Table 3.2 are the results for the mixed coordinate systems. We conclude that, unless inver-

Coordinates	Adding	Doubling
(x, y)	$I + 2M$	$I + 2M + S$
$(x/Z, y/Z)$	$16M + 2S$	$14M + 3S$
$(x/Z^2, y/Z^3)$	$15M + 5S$	$5M + 5S$
$(x/Z, y/Z^2)$	$14M + 6S$	$5M + 5S$

Table 3.1: Field operations for addition and doubling in different coordinate systems.

Coordinates	Adding
$(x/Z, y/Z)$	$14M + 2S$
$(x/Z^2, y/Z^3)$	$11M + 4S$
$(x/Z, y/Z^2)$	$10M + 4S$

Table 3.2: Field operations for addition in mixed coordinate systems. Of the summands, one is given in affine coordinates and the other, as well as the result, is given in the indicated coordinates.

sion is fast enough, we should use López-Dahab coordinates. The crossover point depends on the application. We will mostly be interested in scalar multiplication, and depending on the chosen algorithm, the crossover will be different, ranging from about $t(I)/t(M) > 4.5$ to $t(I)/t(M) > 5.5$.

3.2 Montgomery Methods

The alternative form of the doubling formula presented in the discussion of the López-Dahab representation above, one where the x-coordinate only depends on x, entices us to think that there might be a way to compute elliptic curve operations solely using just one coordinate. To achieve just that same thing for the addition formula as for the doubling formula, let $P_1 = (x_1, y_1)$, $P_2 = (x_2, y_2)$ and $P_3 = (x_3, y_3) = P_1 + P_2$ and consider

$$x_3 = \lambda^2 + \lambda + x_1 + x_2 + B = \left(\frac{y_1 + y_2}{x_1 + x_2}\right)^2 + \frac{y_1 + y_2}{x_1 + x_2} + x_1 + x_2 + B =$$
$$= \frac{y_1^2 + y_2^2 + x_1y_1 + x_2y_2 + x_1y_2 + x_2y_1 + x_1^3 + x_1^2x_2 + x_1x_2^2 + x_2^2 + Bx_1^2 + Bx_2^2}{(x_1 + x_2)^2} =$$

(using the elliptic curve equation)

$$= \frac{x_1^2x_2 + x_1x_2^2 + x_1y_2 + x_2y_1}{(x_1 + x_2)^2}. \tag{3.5}$$

To go further, we use the point $P = (x, y) = P_2 - P_1 = P_2 + (-P_1)$ as P_3 in the formula above, where $-P_1 = (x_1, x_1 + y_1)$. We get

$$x = \frac{x_1^2x_2 + x_1x_2^2 + x_1y_2 + x_2(x_1 + y_1)}{(x_1 + x_2)^2}. \tag{3.6}$$

Calculating $x + x_3$ gives us

$$x_3 = x + \frac{x_1x_2}{(x_1 + x_2)^2} = x + \left(\frac{x_1}{x_1 + x_2}\right)^2 + \frac{x_1}{x_1 + x_2}, \tag{3.7}$$

so we can compute $P_1 + P_2$ using P_1, P_2 and $P_2 - P_1$ with only x-coordinates. To retrieve the y-coordinate easily, we use (3.5) with P as P_2 and P_2 as P_3, to get

$$x_2(x_1 + x) = x_1y + xy_1 + x_1x^2 + xx_1^2.$$

Solving this for y_1 results in

$$y_1 = (x_1 + x)[(x_1 + x)(x_2 + x) + x^2 + y]/x + y. \tag{3.8}$$

So, we can calculate $x(P_1 + P_2)$ using only the $x(P_1),\ x(P_2)$. Similar formulae can be found for projective coordinates ridding us of inversions. Standard projective coordinates seem to be the best, whereby the formulae become

$$\begin{cases} X_3 &= X_1^4 + AZ_1^4, \\ Z_3 &= Z_1^2X_1^2, \end{cases} \qquad \text{when } P_1 = P_2,$$
$$\begin{cases} Z_3 &= (X_1Z_2 + X_2Z_1)^2, \\ X_3 &= xZ_3 + (X_1Z_2)(X_2Z_1), \end{cases} \qquad \text{when } P_1 \neq P_2,$$

where $x = x(P)$ as before. We conclude that, $t_2(\text{mg}) \leq 2M + 4S + A$ and $t_+(\text{mg}) \leq 4M + S + 2A$, but for conversion we also need $I + 3M + S + 5A$.

We'll get back to Montgomery methods in the next section.

3.3 Scalar Multiplication

We now turn our attention to the very important problem of calculating $[k]P$ where $k \in \mathbb{Z}$, a problem also known as the addition chain problem. To find the optimal solution to the addition chain problem is in general a difficult problem [8]. So, we are going to look for simpler, yet satisfyingly fast, methods.

We will frequently use the binary representation of k, $k = k_0 + 2k_1 + 2^2k_2 + \cdots + 2^{l-1}k_{l-1}$, where $k_i \in \{0,1\}$. We will commonly use vector notation, so $k = (k_{l-1}, \ldots, k_1, k_0)$.

3.3.1 The binary method

The perhaps simplest method is the well known binary, or double-and-add, method. This algorithm is outlined in Algorithm 12.

Algorithm 12 The Binary Method for Scalar Multiplication

Require: $k = (k_{l-1}, \ldots, k_1, k_0)$, $P \in E$
Ensure: $[k]P$
  $Q = \mathcal{O}$
  for i from $l-1$ to 0 **do**
    $Q \leftarrow 2Q$
    if $k_i = 1$ **then**
      $Q \leftarrow Q + P$
    end if
  end for
  return Q

On average, for a random $k \in \mathbb{Z}$ half of the k_i's are 1, which means that the binary method uses roughly n doublings and $n/2$ additions. However, we can do better.

3.3.2 Non-adjacent forms

As we saw earlier, subtraction is almost as a difficult operation as addition in the elliptic curve group (one more field addition is needed). Thus it might be advantageous to use both additions and subtractions when scalar multiplying. Ponder, for instance, $[15]P = P + [2]P + [4]P + [8]P = [16]P - P$, where the first (binary) method needs three doublings and three additions, whereas the second needs only four doublings and one addition.

A solution to this problem, is to use the non-adjacent form (NAF) of k. The NAF's are representations such that no consecutive coefficients k_i are non-zero and the number of non-zero coefficients are the fewest of any signed digit representations. Fortunately, computing NAF's are simple, and if we have enough memory available, we may compute NAF's with $|k_i| \leq 2^{m-1}$, which we will call $\mathrm{NAF}_m(k)$. An algorithm to compute $\mathrm{NAF}_m(k)$ is presented in Algorithm 13, where $k \operatorname{mods} 2^m$ is the integer l satisfying $l \equiv k \pmod{2^m}$ and $-2^{m-1} \leq l < w^{m-1}$.

Using the NAF to rewrite the binary method is then rather straightforward, as can be seen in Algorithm 14. Note here that if no precomputation is wanted, set $m = 2$ to get the ordinary NAF-method. As approximately $1/(m+1)$ of the coefficients of $\mathrm{NAF}(k)$ are nonzero

Algorithm 13 Computing $\mathrm{NAF}_m(k)$

Require: k, m
Ensure: $\mathrm{NAF}_m(k)$

```
i ← 0
while k ≥ 1 do
  if k is odd then
    k_i ← k mods 2^m
    k ← k − k_i
  end if
  k ← k/2
  i ← i + 1
end while
```

[23], we expect Algorithm 14 to use approximately $n/(m+1)$ additions and n doublings. For precomputation, we need 1 doubling and $2^{m-2} - 1$ additions.

Algorithm 14 The Window NAF_m-method for scalar multiplication

Require: k, m, $\mathrm{NAF}_m(k) = (k_{l-1}, \ldots, k_0)$, $P \in E(\mathbb{F}_{2^n})$
Ensure: $[k]P$

```
Precompute P_i = [i]P for i ∈ {1, 3, 5, ..., 2^(m−1) − 1}
Q ← O
for i from l − 1 downto 0 do
  Q ← [2]Q
  if k_i ≠ 0 then
    if k_i > 0 then
      Q ← Q + P_{k_i}
    else
      Q ← Q − P_{k_i}
    end if
  end if
end for
```

3.3.3 Fixed-base comb

If we are allowed a modest amount of precomputation and space, we can split the binary representation of k in $[k]P$ in w parts, each d bits as $k = (k_{l-1}, \ldots, k_{(w-1)d}, \ldots, k_{(w-2)d}, \ldots, k_d, k_{d-1}, \ldots, k_0)$, where we pad with zeros if $l \not\equiv 0 \mod d$. Let $(a_{w-1}, \ldots, a_0)P = a_{w-1}[2^{(w-1)d}]P + \cdots a_2[2^{2d}]P + a_1[2^d]P + a_0 P$ be precomputed for all $(a_{w-1}, \ldots, a_1, a_0)$. Then

$$\begin{aligned}[k]P =& (k_{(w-1)d}, \ldots, k_{2d}, k_d, k_0)P + \cdots \\ &+ (k_{wd-2}, \ldots, k_{2d-2}, k_{d-2})P + (k_{wd-1}, \ldots, k_{2d-1}, k_{d-1})P,\end{aligned}$$

so we can reduce the number of point additions by $1/w$ while storing $2^w - 2$ points (P and $\mathcal{O}$ may be excluded). For details, see Algorithm 15. For precomputations, we need $w-1$ point doublings and $2^w - w - 1$ point additions, after which we need d doublings and d additions.

Algorithm 15 Fixed-base comb for scalar multiplication

Require: $k = (k_{m-1}, \ldots, k_0)$, $P \in E$ and a window size w
Ensure: $[k]P$
  Precompute $(a_{w-1}, \ldots, a_1, a_0)P$.
  $Q \leftarrow \mathcal{O}$, $d \leftarrow \lceil n/w \rceil$
  for i from $d-1$ downto 0 **do**
    $Q \leftarrow [2]Q$
    $Q \leftarrow Q + (k_{(w-1)d+i}, \ldots, k_{d+i}, k_i)P$
  end for

3.3.4 Montgomery scalar multiplication

In section 3.2 we described a function f such that $P_1 + P_2 = f(P_1, P_2, P_2 - P_1)$, where only x- (and z)-coordinates were needed except for retrieving the final y-coordinate. To turn this into an algorithm useful for scalar multiplication, we need to find a chain of numbers (m_i) where $m_i = m_j + m_l$ for $j, l < i$ and $m_j - m_l \in \{0, 1\}$ (as we only know the full coordinates of one point, namely P itself), $n < i$, and, of course, $m_i = k$ for some i. One way of finding such a chain is to use the binary expansion of $k = (k_{l-1}, \ldots, k_0)$ and let

$$m_{2i} = \begin{cases} 2m_{2i-2}, & \text{if } k_{l-i} = 0, \\ m_{2i-2} + m_{2i-1}, & \text{if } k_{l-i} = 1, \end{cases}$$
$$m_{2i+1} = \begin{cases} m_{2i-2} + m_{2i-1}, & \text{if } k_{l-i} = 0, \\ 2m_{2i-2}, & \text{if } {}_{l-i}k = 1, \end{cases}$$

where we let $m_0 = 1$, $m_1 = 2$.

3.4 Dual Point Scalar Multiplication

For instance, in the Elliptic Curve Digital Signature Algorithm (ECDSA) [7], we are required to calculate $[k]P + [l]Q$. We give a condensed presentation of two methods to do these calculations quickly.

3.4.1 Shamir's trick

In a fashion very similar to the binary method, Shamir's trick is to utilize the binary expansions of both k and l simultaneously. For further details, consult Algorithm 16.

Note that this algorithm can be extended to use larger windows in the spirit of Algorithm 14. In the algorithm presented here, we need approximately $(m-1)$ doublings and $(m-1)$ additions. Compare this with the binary method that needs $(m-1)$ doublings and approximately $m/2$ additions to see that it is an efficient method.

3.4.2 A Montgomery method

In the spirit of Shamir's trick, the Montgomery scalar multiplication scheme can also be utilized to render a fast dual point multiplication algorithm, see Algorithm 17.

Algorithm 16 Shamir's trick for dual point scalar multiplication

Require: $k = (k_{m-1}, \ldots, k_0), l = (l_{m-1}, \ldots, l_0)$ and $P, Q \in E$
Ensure: $[k]P + [l]Q$
  $R \leftarrow \mathcal{O}$
  Precompute $[k_i]P + [l_i]Q$ for $k_i, l_i = 0, 1$
  for i from $m - 1$ downto 0 **do**
    $R \leftarrow [2]R$
    $R \leftarrow R + ([k_i]P + [l_i]Q)$
  end for

Algorithm 17 Montgomery dual point scalar multiplication

  $f(k, P, l, Q, P + Q)$
Require: k, l, and $P, Q, P + Q \in E$
Ensure: $[k]P + [l]Q$
  if $k = l$ **then**
    return $[k](P + Q)$ (ordinary scalar multiplication)
  else if $k < l$ **then**
    return $f(l, Q, k, P, Q + P)$
  else if $k \leq 5l$ **then**
    return $f(k - l, P, l, P + Q, [2]P + Q)$
  else if k is even **then**
    return $f(k/2, [2]P, l, Q, [2]P + Q)$
  else if $k \equiv l \mod 2$ **then**
    return $f((k - l)/2, [2]P, l, P + Q, [3]P + Q)$
  else
    return $f(k, P, l/2, 2[Q], P + [2]Q)$
  end if

We should note that in the ECDSA, we know one of the two points beforehand, and it might therefore be advantageous to use a method which uses precomputations for one of the points and a method that doesn't for the other point.

3.5 Summary

We first give a short summary of the presented algorithms for addition and doubling using different coordinate systems in Table 3.3 (this is a repetition of Table 3.1 and 3.2).

Coordinates	Adding	Doubling	Mixed coordinates
(x, y)	$I + 2M$	$I + 2M + S$	
$(x/Z, y/Z)$	$16M + 2S$	$14M + 3S$	$14M + 2S$
$(x/Z^2, y/Z^3)$	$15M + 5S$	$8M + 3S$	$11M + 3S$
$(x/Z, y/Z^2)$	$14M + 6S$	$5M + 5S$	$10M + 4S$

Table 3.3: Field operations for addition and doubling in different coordinate systems

Next, in Table 3.4, we give a summary of the number of point doublings and additions that are needed in the different scalar multiplication algorithms presented. The specific case when E is an elliptic curve over $\mathbb{F}_{2^{163}}$ is presented in Table 3.5. In the last column of Table 3.5, mixed coordinates are used where precomputed points are represented using affine coordinates.

	Binary	Window NAF_w
Precomputations	0	$D + (2^{w-2} - 1)A$
Points stored	0	$w - 1$
EC ops	$(n-1)D + (n/2)A$	$(n/(w+1))A + (n-1)D$
	Fixed-base comb	Montgomery
Precomputations	$\lceil n/w \rceil (w-1)D + (2^w - w - 1)A$	0
Points stored	$2^w - 2$	0
EC ops	$(\lceil n/w \rceil - 1)D + (\lceil n/w \rceil - 1)A$	$6nM + 1I + 10M$

Table 3.4: Elliptic curve operations needed for scalar multiplication in the general case of an elliptic curve defined over $\mathbb{F}_{2^n}$. A denotes elliptic curve additions, D doublings and M, I multiplications and inversions in $\mathbb{F}_{2^n}$.

We conclude that if memory is not constrained and if the point is known, the fixed-base comb is the fastest method, otherwise Montgomery multiplication is fastest. We also conclude that we should use affine coordinates if the ratio $t(I)/t(M)$ is smaller than 5.5, since the fastest method is the fixed-base comb and $(600 - 160)/80 = 5.5$ (the crossover for Montgomery multiplication is much lower).

From the timings in Section 2.5, where $t(I)/t(M) > 11$, we see that we should use López-Dahab projective coordinates mixed with affine. With this choice, we get timings for $\mathbb{F}_{2^{163}}$ on the emulated 10 MHZ ARM 7TDMI according to Table 3.6. As Montgomery dual scalar multiplication seems to be in-between the two Shamir variants, we do not list it.

	Binary	Window NAF_4	NAF_5
Precomputations	0	$D + 3A$	$D + 7A$
Points stored	0	3	7
EC ops	$162D + 82A$	$33A + 162D$	$21A + 162D$
Affine	$244I + 488M$	$195I + 380M$	$183I + 366M$
L-D/affine	$1958M$	$1140M$	$1020M$
	Fixed-base comb, $w = 4$	Montgomery	
Precomputations	$123D + 11A$	0	
Points stored	14	0	
EC ops	$40D + 40A$		
Affine	$80I + 160M$		
L-D/affine	$600M$	$988M + 1I$	

Table 3.5: Approximate number of elliptic curve operations needed for scalar multiplication in the case of an elliptic curve defined over $\mathbb{F}_{2^{163}}$.

Method	Timing
Montgomery scalar multiplication	970 ms
Fixed-base comb, $w = 4$	550 ms
Shamir dual point multiplication	1850 ms
Shamir dual point, windowed (13 points stored)	1630 ms
Montgomery + Fixed base dual point	1520 ms

Table 3.6: Timings elliptic curve operations on an elliptic curve defined over $\mathbb{F}_{2^{163}}$ implemented on an emulated 10 MHZ ARM 7TDMI.

Chapter 4

Elliptic Curve Parameter Selection

4.1 Advanced Elliptic Curve Point Counting

In the introduction, we presented two simple algorithms to count the number of points on an elliptic curve, a simplistic, exhaustive method involving the Legendre symbol, and a somewhat more intricate one, the baby-step-giant-step approach due to Shanks. The former of these, we asserted, has running time $O(p^{1+\varepsilon})$, whereas the latter has running time $O(p^{1/4+\varepsilon})$ over $\mathbb{F}_p$. In terms of the number of bits in p, $\lceil \log_2 p \rceil$, these running times are both exponential, and rather unpractical for p larger than 20–30 digits.

In this section, we will find out that there are faster, sub-exponential, algorithms that are efficient and much more practical.

4.1.1 Schoof's algorithm, $\mathbb{F}_p$

In 1985, R. Schoof presented the first sub-exponential algorithm for counting points on elliptic curves, [18][1]. This algorithm, which serves as the basis for many alterations, among which we will mention only a few, is the only as-of-yet known practical way to count the number of points on elliptic curves over large prime-fields. Our presentation will follow Schoof [18, 19], but see also [6].

Let an affine equation for an elliptic curve E defined over $\mathbb{F}_p$, where $p \neq 2, 3$ is a prime, be given as

$$y^2 = x^3 + ax + b,$$

where $\Delta = 4a^3 + 27b^2 \neq 0$ (so the curve is non-singular). The idea of Schoof's algorithm is this: count the number of points of the curve modulo some small primes $l_1 = 2, l_2 = 3, \ldots$ such that

$$\prod_i l_i > 2\sqrt{p}$$

Since, as we saw in Chapter 1, $|\#E(\mathbb{F}_p) - p - 1| \leq 2\sqrt{q}$, we may then retrieve $\#E(\mathbb{F}_p)$ using the Chinese remainder theorem. The Prime Number Theorem tells us that we need at most

[1] As a rather humorous side-note, interest for point counting in those long forgotten days was rather shallow, hence the name of the article, 'Elliptic Curves Over Finite Fields and the Computation of Square Roots mod p'. Rumour has it, the lack of an important application of the result in the paper led to a refusal from the publisher and a subsequent addendum of square root-computations from the author.

$O(\log p)$ of size no larger than $O(\log p)$, so the algorithm is indeed sub-exponential. First, we need some definitions.

Let the p-Frobenius morphism $\phi_p : E \to E$ be defined as

$$\phi_p(x, y) = (x^p, y^p).$$

Then ϕ_p is an automorphism that satisfies an equation

$$\phi^2 - [t]\phi + [p] = 0, \qquad t \in \mathbb{Z},$$

where t is called the trace of the Frobenius morphism and $[k]$ denotes the multiplication-by-k-morphism. It is well known that $\#E(\mathbb{F}_p) = p + 1 - t$, so Hasse's theorem gives the bound, $|t| \leq 2\sqrt{p}$. For further information on these facts, see e.g. [21]. We want to solve this equation $\bmod\ l_i$, which means that we need the points belonging to the l_i-torsion subgroup of E:

$$E[l_i] = \{P \in E(\overline{\mathbb{F}}_p) : [l_i]P = \mathcal{O}\},$$

which is isomorphic to $\mathbb{Z}/l_i\mathbb{Z} \times \mathbb{Z}/l_i\mathbb{Z}$ if $l_i \neq p$.

To find the points in a specific l_i-torsion subgroup, we use the division polynomials ψ_{l_i} defined by ($[l_i]P \neq \mathcal{O}$)

$$[l_i]P = \left(\frac{\theta_{l_i}(x, y)}{\psi_{l_i}^2(x, y)}, \frac{\omega_{l_i}(x, y)}{\psi_{l_i}^3(x, y)} \right).$$

It is possible to derive a recursion formula, and after substituting y^2 by $x^3 + ax + b$ we can remove y altogether by letting $f_n(x) = \tilde{\psi}_n(x, y)$, if n is odd and $f_n = \tilde{\psi}_n(x, y)/y$ if n is even. Here $\tilde{\psi}_n$ is ψ_n with y^2 substituted. The degree of $\tilde{\psi}_n$ is $n^2 - 1$, which means that they grow slowly. These polynomials are useful since if $P = (x, y) \in E[l_i]$, then $\psi_{l_i}(x, y) = 0$. Our task is then to find t_i's that satisfy

$$\phi^2 - [t_i]\phi + [p_i] = 0, \qquad t_i \in \{0, 1, 2, \ldots, l_i - 1\}$$

or

$$(x^{p^2}, y^{p^2}) + [p_i](x, y) = [t_i](x^p, y^p) \qquad \text{in} \quad \mathbb{F}_p[x, y]/(\psi_{l_i}(x, y), y^2 - x^3 - ax - b),$$

where $p_i \equiv p \mod l_i$. Finding t_i is a matter of trial and error—we start by finding a $P \in E[l_i]$ and then check which $t_i \in \{0, \pm 1, \ldots, \pm(l_i - 1)/2\}$ satisfies the equation above. This concludes Schoof's original algorithm, which has, with naïve arithmetics, running time $O(\log^8 p)$.

Following Schoof several alterations to his algorithms have been made, most notably by Atkin and Elkies. See for example [19, 1].

4.1.2 Satoh's algorithm and relatives, $\mathbb{F}_{2^n}$

We will now turn our attention to counting points on curves defined over $\mathbb{F}_{2^n}$ using Satoh's algorithm and modifications thereof. The principle behind Satoh's algorithm is completely different from Schoof's. The principle is to lift the elliptic curve E defined over $\mathbb{F}_{2^n}$ to an elliptic curve $E^{\uparrow}$ defined over an unramified extension of the 2-adic numbers, which is a field of characteristic zero. We then find the trace of the Frobenius morphism in this field, and then transform it back to $\mathbb{F}_{2^n}$. The algorithm works for any $\mathbb{F}_{p^n}$ (with slight modifications), but the dependence on p is bad which is why we choose p low, especially $p = 2$.

Preliminaries

First we need some basic facts about morphisms on elliptic curves. These can also be found in any regular text book such as [21].

A morphism $f : E_1 \to E_2$, where E_1, E_2 are elliptic curves defined over some field K, is called an isogeny if $f(\mathcal{O}) = \mathcal{O}$. The morphism f is regular if it defines a homomorphism $g \mapsto g \circ f : K[E_2] \to K[E_1]$ from the K-algebra of regular functions on E_1 to the K-algebra of regular functions on E_2. This map is injective, so it defines a map on the fields of fraction $K(E_2) \to K(E_1)$, which realizes $K(E_1)$ as a finite extension of $K(E_2)$. We denote by $\deg f$ the degree of this extension. Furthermore, we say that f is separable if $K(E_1)$ is separable over $K(E_2)$, see for example [14].

If $f : E_1 \to E_2$ is an isogeny, then the unique isogeny $\hat{f} : E_2 \to E_1$ such that $f \circ \hat{f} = [\deg f]$ is called the its dual isogeny. Here, as previously, $[k]$ is the multiplication-by-k-morphism.

Also, if $f : E \to E$ is an isogeny, and $\hat{f}$ its dual isogeny, then $f + \hat{f} = [\operatorname{tr} f]$. Especially, if $f = \phi_p$, the p-Frobenius morphism, then $\deg f = p$, and $\operatorname{tr} f$ is the actual trace of Frobenius found in Hasse's theorem, see [21] §V. Furthermore, $\hat{\phi}_p$ is called the Verschiebung, and it is separable if E is non-supersingular. In our case $K = \mathbb{F}_{2^n}$, so the 2^n-Frobenius, can be divided into a chain of n 2-Frobenius morphisms $\phi_2^{(i)} : E_{i-1} \to E_i$ according to

$$E = E_0 \stackrel{\phi_2^{(1)}}{\to} E_1 \stackrel{\phi_2^{(2)}}{\to} E_2 \stackrel{\phi_2^{(3)}}{\to} \cdots \stackrel{\phi_2^{(n-1)}}{\to} E_{n-1} \stackrel{\phi_2^{(n)}}{\to} E_n = E, \tag{4.1}$$

since $\phi_{2^n} = \phi_2^{(1)} \circ \phi_2^{(2)} \circ \cdots \circ \phi_2^{(n)}$. The same, in reverse order, holds for the duals, $\hat{\phi}_2^{(i)}$.

We also need to know something about 2-adic numbers. For details, see [3]. A 2-adic (or p-adic in general, where p is a prime) integer is a sequence $x = (x_1, x_2, \ldots, x_d, \ldots)$ where $x_d \in \mathbb{Z}/2^d\mathbb{Z}$ such that $x_{d+1} \equiv x_d \mod 2^d$. A 2-adic integer can be approximated at a precision d by the sequence $x \approx (x_1, x_2, \ldots, x_d)$. In a computer, we realize such an approximated integer just as x_d, which can be represented as a binary number with d bits. We denote the 2-adic integers by $\mathbb{Z}_2$, and their quotient field, the 2-adic numbers, by $\mathbb{Q}_2$.

If $f(t)$ is a monic polynomial of degree n with coefficients in $\mathbb{Z}_2$, such that $f(t) \mod 2$ is irreducible in $\mathbb{Z}/2\mathbb{Z} = \mathbb{F}_2$, then we denote $R = \mathbb{Z}_2[t]/(f(t)) = \left\{a_0 + a_1 t + \cdots + a_{n-1}t^{n-1} \mod f(t) : a_i \in \mathbb{Z}_2\right\}$. If we approximate $a \in R$ at precision d, we approximate each coefficient a_i as indicated above. Here, R is an unramified discrete valuation ring of $\mathbb{Q}_2[x]/(f(x))$ over $\mathbb{Z}_2$.

The Satoh-Skjernaa algorithm

The idea of Satoh's algorithm is to lift the elliptic curve to an unramified extension of the p-adic numbers, and calculate the trace of the Frobenius morphism at a good enough precision over this field. Satoh originally presented this algorithm for characteristic $p > 3$ [17], and there are two different approaches to extend it to characteristic 2, of which we choose to present Skjernaa's [22, 3][2].

From now on, let E be a non-supersingular elliptic curve defined over $\mathbb{F}_{2^n}$ given by an equation

$$y^2 + xy = x^3 + a.$$

[2] In [3] "the other" extension is given, that also includes characteristic 3. The difference between the methods is that Skjernaa gives an explicit formula for the point in $\ker \phi_2^{(i)\uparrow}$, whereas Foquet, Gaudry and Harley use a Newton iteration to find it.

Note that every non-supersingular elliptic curve over $\mathbb{F}_{2^n}$ can be rewritten into this equation[3]. Suppose also, that $j(E) \in \mathbb{F}_{2^n} \setminus \mathbb{F}_4$, which is no restriction, since if $j(E) \in \mathbb{F}_4$ the elliptic curve is isomorphic to a curve defined over $\mathbb{F}_2$ or $\mathbb{F}_4$. The number of points of such curves are easy to handle using Weil's theorem[4].

The canonical lift We now construct the canonical lift $E^\uparrow$ of an elliptic curve E. According to [11] there exists only one canonical lift $E^\uparrow$ such that

- $\mathrm{End}(E) \cong \mathrm{End}(E^\uparrow)$
- $E^\uparrow \mod p$ is E

If these conditions hold, $\mathrm{tr}\,\phi_p = \mathrm{tr}\,\phi_p^\uparrow$. To find $E^\uparrow$ explicitly, we use the following theorem.

Theorem (Lubin-Serre-Tate in characteristic 2). *Let E be an elliptic curve over $\mathbb{F}_{2^n}$, with j-invariant $j(E) \in \mathbb{F}_{2^n} \setminus \mathbb{F}_4$. Then there is a unique $j(E^\uparrow) \in R$ such that*

$$\Phi_2(j(E^\uparrow), \Sigma(j(E^\uparrow))) = 0, \quad \textit{and} \quad j^\uparrow = j \mod 2,$$

and $j(E^\uparrow)$ is the j-invariant of $E^\uparrow$. Here Σ denotes the Frobenius substitution $x \mapsto x^2$ on R.

The theorem uses $\Phi_2(x,y)$, the 2nd modular polynomial,

$$\begin{aligned}\Phi_2(x,y) =& x^3 + y^3 - x^2y^2 + 2^4 3 \cdot 31(xy^2 + x^2y) - 2^4 3^4 5^3 (x^2 + y^2) \\ &+ 3^4 5^3 \cdot 4027xy + 2^8 3^7 5^6 (x+y) - 2^{12} 3^9 5^9.\end{aligned}$$

In general the p-th modular polynomial has several important properties: It is symmetric, two elliptic curves E and E' have a p-isogeny if and only if $\Phi_p(j(E), j(E')) = 0$ and if p is a prime, it satisfies the Kronecker relation

$$\Phi_p(x,y) \equiv (x^p - y)(y - x^p) \mod p. \tag{4.2}$$

In order to calculate $j^\uparrow$, Satoh uses the Lubin-Serre-Tate theorem to calculate the j-invariants of the curves $E^\uparrow = E_0^\uparrow, E_1^\uparrow, \ldots, E_{n-1}^\uparrow, E_n^\uparrow = E^\uparrow$ in the chain (4.1). That is, he solves the system of equations ($j_0 = j = j_n$)

$$\begin{aligned}\Phi_2(j(E_0^\uparrow), j(E_1^\uparrow)) &= 0, \\ \Phi_2(j(E_1^\uparrow), j(E_2^\uparrow)) &= 0, \\ \Phi_2(j(E_2^\uparrow), j(E_3^\uparrow)) &= 0, \\ &\vdots \\ \Phi_2(j(E_{n-1}^\uparrow), j(E_n^\uparrow)) &= 0,\end{aligned}$$

[3]In general $E: \ y^2 + xy = x^3 + a_2x^2 + a_6$. But if $\mathrm{tr}\, a_2 = 0$ ($\mathrm{tr}\, a = a + a^2 + a^3 + \cdots + a^{2^{n-1}}$), we can solve $s^2 + s + a_2 = 0$ and make the substitution $y = y' + sx', \ x = x'$. If $\mathrm{tr}\, a_2 \neq 0$, consider E', the quadratic twist of E: $a_2y^2 + xy = x^3 + a_2x^2 + a_6$, which has an equation $y^2 + xy = x^3 + a_2^2x^2 + a_2^3a_6$ (through $y = y'/a_2^2, \ x = x'/a_2$). Now $\#E(\mathbb{F}_{2^n}) + \#E'(\mathbb{F}_{2^n}) = 2^n + 2$.

[4]Taking the n:th derivative of the zeta-function, and some manipulations, gives the recursion formula $t_n = t_1t_{n-1} - 2t_{n-2}$, where $\#E(\mathbb{F}_{2^n}) = 2^n + 1 - t_n$ and $t_0 = 2$.

using a multivariate Newton iteration initialized with $(j_0, j_1, \ldots, j_n)$ where $j_i^2 \equiv j_{i+1} \mod 2$, $j_i \mod 2 \notin \mathbb{F}_4$ and $\Phi_2(j_i, j_{i+1}) \equiv 0 \mod 2$. The Newton iteration converges quadratically, and the Jacobian matrix $D(\Phi_2(j(E_0^\uparrow), j(E_1^\uparrow)), \ldots, \Phi_2(j(E_{n-1}^\uparrow), j(E_n^\uparrow)))$ (where the derivative is taken with respect to the first argument) is bidiagonal, so an implementation can be made rather efficiently. The Kronecker relation (4.2) is important for the invertibility of the Jacobian.

Calculating the trace of the Frobenius morphism If f is a morphism of an elliptic curve, and $\hat{f}$ is its dual, then $\operatorname{tr} f = \operatorname{tr} \hat{f}$. We can use that in combination with the following theorem to find the trace of the Frobenius morphism.

Theorem. *Let $E^\uparrow$ be an elliptic curve, and let $f^\uparrow$ be a morphism of $E^\uparrow$ with $\deg f^\uparrow = d$. Furthermore, let τ be the local parameter of $E^\uparrow$ at $\mathcal{O}$, and assume that the reduction of $f \mod p$ is separable. Then $\operatorname{tr} f = c_1 + d/c_1$, where $\tau \circ f = \sum_1^\infty c_n \tau^n$.*

See [17, 22] for a proof.

Now, recall that the Verschiebung, $\hat{\phi}_2$, is separable, so if we have the chain

$$E^\uparrow \xleftarrow{\hat{\phi}_2^{(1)\uparrow}} E_1^\uparrow \xleftarrow{\hat{\phi}_2^{(2)\uparrow}} E_2^\uparrow \xleftarrow{\hat{\phi}_2^{(3)\uparrow}} \cdots \xleftarrow{\hat{\phi}_2^{(n-1)\uparrow}} E_{n-1}^\uparrow \xleftarrow{\hat{\phi}_2^{(n)\uparrow}} E^\uparrow,$$

where thus $\hat{\phi}_2^{(i)\uparrow} : E_i^\uparrow \to E_{i-1}^\uparrow$. If $\tau_{i-1} \circ \hat{\phi}_2^{(i)\uparrow} = c_i \tau_i + O(\tau_i^2)$, we have $\tau \circ \hat{\phi}_{2^n}^\uparrow = \prod_1^n c_i \tau + O(\tau^2)$, so $\operatorname{tr} \phi_{2^n} = \operatorname{tr} \hat{\phi}_{2^n} = \prod_1^n c_i$. The problem has been reduced to finding c_i.

In order to find c_i, or actually c_i^2, we use a formula due to Vélu [24]. There exists a unique isomorphism that makes the following diagram commutative

$$\begin{tikzcd}
E_i^\uparrow \arrow[rr, "\hat{\phi}_2^{(i)\uparrow}"] \arrow[dr] & & E_{i-1}^\uparrow \\
& E_i^\uparrow / \ker \hat{\phi}_2^{(i)\uparrow} \arrow[ur, "\lambda"'] &
\end{tikzcd}$$

Vélu gave a formula for $E_i^\uparrow / \ker \hat{\phi}_2^{(i)\uparrow}$, so, given that $\lambda(x, y) = (u^2 x + r, u^3 y + u^2 s x + t)$ is an isomorphism, we can solve for u in terms of the j-invariant of the elliptic curve $E_i^\uparrow$ and the x-coordinate of the non-trivial point in $\ker \phi_2^{(i)\uparrow}$. The j-invariant we already have from the previous canonical lift, and the x-coordinate of the point in $\ker \phi_2^{(i)\uparrow}$ can be found using $\ker \phi_2^{(i)\uparrow} \subset E_i^\uparrow[2]$ and accordingly must be a zero of the second division polynomial. Following Skjernaa [22] and Vercauteren, Preneel and Vandewalle [25], we get

$$\begin{aligned}
z_i &= x_i/2 = \frac{(j(E_i))^2 + 195120 j(E_i) + 409 j(E_{i-1}) + 660960000)/2^{12}}{(j(E_i))^2 + j(E_i)(563760 - 512 j(E_{i-1})) + 372735 j(E_{i-1}) + 8981280000)/2^9}, \\
t_i &= (12 z_i^2 + z_i)(j(E_{i-1}) - 1728) - 46, \\
c_i^2 &= \frac{j(E_i) - (504 + 12096 z_i) t_i}{j(E_i) + 240 t_i}.
\end{aligned}$$

To summarize:

1. Canonical lift; lift the j-invariants using a multivariate Newton iteration.
2. Calculate c_i^2.
3. Calculate $\sqrt{\prod c_i^2} \equiv 1 \mod 4$.

The Vercauteren-Preenel-Vandewalle modification

In the original Satoh algorithm, the multivariate Newton iteration requires $O(n^3)$ space. The following observation by Vercauteren, Preneel and Vandewalle [25] reduces the space usage to $O(n^2)$.

Theorem (Vercauteren-Preneel-Vandewalle). *Let $G \in R[x,y]$ and assume $x_0, y_0 \in R$ such that*

$$g(x_0,y_0) \equiv 0 \mod p, \quad \frac{\partial g}{\partial x}(x_0,y_0) \not\equiv 0 \mod p \quad \textit{and} \quad \frac{\partial g}{\partial y}(x_0,y_0) \equiv 0 \mod p.$$

Then

1. *For every $y \equiv y_0 \mod p$ there exists a unique $x \in R$ such that $x \equiv x_0 \mod p$ and $g(x,y) = 0$.*
2. *Let $y' \in R$ where $y \equiv y' \mod p^M,\ M \geq 1$ and let $x' \in R$ be the unique element such that $x' \equiv x_0 \mod p$ and $g(x',y') = 0$. Then $x' = x \mod p^{M+1}$.*

In other words, when solving the univariate $\Phi_2(x_{i-1}, x_i)$ with $x_i \equiv j(E_i^{\uparrow}) \mod 2^d$ we get $x_{i-1} \equiv j(E_{i-1}^{\uparrow}) \mod 2^{d+1}$, and so gain in p-adic precision.

This means that instead of solving a system of equations, we can first find $j(E^{\uparrow}) = j(E_0^{\uparrow})$ with precision n from $j(E^{\uparrow})$ with precision 1 using n Newton iterations. Once we have $j(E^{\uparrow})$ at high enough precision, we can cycle through $E_{n-1}^{\uparrow},\ E_{n-2}^{\uparrow}, \ldots, E_1^{\uparrow}, E_0^{\uparrow}$, calculate $j(E_i^{\uparrow})$ and c_i^2. The algorithm then becomes:

1. For $i = n$ to 0 calculate $j(E_i^{\uparrow})$ with increasing p-adic precision.
2. For $i = n$ to 0 calculate $j(E_i^{\uparrow})$ and then c_i^2.
3. Calculate $c \equiv \sqrt{\prod c_i^2} \mod p^n$ with the correct sign.

Implementation details

We now describe in some detail how to implement the Satoh-Vercauteren-Preneel-Vandewalle point counting algorithm. At the heart of this algorithm is the Newton iteration to solve

$$\Phi_2(x_i, x_{i+1}) = 0, \qquad x_i, x_{i+1} \in R = \left\{ \sum_{i=0}^{n} a_i t^i \mod f(t) : a_i \in \mathbb{Z}/2^d\mathbb{Z} \right\},$$

given $x_{i+1} \equiv j(E_{i+1}^{\uparrow}) \mod 2^d$ for some precision d. The Newton iteration is

$$x_i^{(k+1)} = x_i^{(k)} - \frac{\Phi_2(x_i^{(k)}, x_{i+1})}{\frac{\partial}{\partial x}\Phi_2(x_i^{(k)}, x_{i+1})},$$

where the derivative is taken with respect to the first argument of $\Phi_2(x, y)$. Since x_{i+1} is already known, we can write, and pre-calculate, $\Phi_2(x_i^{(k)}, x_{i+1})$ as the polynomial $\Phi_2(x_i^{(k)}, x_{i+1}) = (x_i^{(k)})^3 + A(x_i^{(k)})^2 + B(x_i^{(k)}) + C$. As the starting point, we use $x_i^0 = x_{i+1}^2 \mod 2$.

Newton's method converges quadratically, and at iteration k we will have $\Phi_2(x_i^{(k)}, x_{i+1}) \equiv 0 \mod 2^k$, which can be utilized to reduce computational complexity. If $a \equiv 0 \mod 2^k$, then $a \cdot b \equiv 0 \mod 2^k$, so instead of computing $a \cdot b \mod 2^l$, $l > k$, we can compute $((a/2^k) \cdot b \mod 2^{l-k}) \cdot 2^k \mod 2^l$. This is faster since division and multiplication by 2^k can be implemented using fast bit-shifts, and the reduction of the precision make operations much faster.

An algorithm for this Newton iteration can be found in Algorithms 18 and 19.

Algorithm 18 Lifting j-invariant using Newton iteration

Require: $x_{i+1} \in R$ with precision d s.t. $x_{i+1} \equiv j(E_{i+1}^{\uparrow}) \mod 2^{d-1}$.
Ensure: $x_i \in R$ with precision d s.t. $x_i \equiv j(E_i^{\uparrow}) \mod 2^d$
  $A \leftarrow x_{i+1}(x_{i+1} + 1488) - 16200 \mod 2^d$
  $B \leftarrow x_{i+1}(40773375 + 1488x_{i+1}) + 8748000000 \mod 2^d$
  $C \leftarrow x_{i+1}(8748000000 + x_{i+1}(x_{i+1} - 16200)) - 157464000000000 \mod 2^d$
  Return NewtonRec(d, A, B, C)

Algorithm 19 Lifting j-invariant using Newton iteration, recursive step

  Defines $j =$**NewtonRec**(d, A, B, C)
  if $d = 1$ **then**
    $j \leftarrow x_{i+1}^2 \mod 2$
  else
    $d' \leftarrow \lceil d/2 \rceil$
    $j =$ NewtonRec(d', A, B, C)
    $D \leftarrow j(B + j(A + j)) + C \mod 2^d$
    $N \leftarrow j(2A + 3j) + B \mod 2^{(d-d')}$
    $j \leftarrow j - ((D/2^{d'})/N \mod 2^{(d-d')}) \cdot 2^{d'} \mod 2^d$
  end if
  Return j

To complete the Newton iteration, we need the basic arithmetic operations in R, addition, subtraction, multiplication and inversion. The first three of these, are straightforward to implement using ordinary polynomial arithmetic with coefficients $\mod 2^d$. If the elliptic curve is defined over an extension field of degree in a cryptographically interesting range, 100–500, it appears to be fastest to implement the coefficient multiplication using the ordinary 'schoolbook' method, and the polynomial multiplication using Karatsuba's trick. For inversion of an element $a \in R$, we use a Newton iteration $x \leftarrow x(2 - ax)$ using $1/x \mod 2$ (an inversion in $\mathbb{F}_{2^n}$ using, say, the Extended Euclidean Algorithm) as our starting point.

Since all coefficients are calculated $\mod 2^d$, reduction can be implemented using a binary AND operation, though it should be noted that we rarely need to do this. Instead, we do the arithmetics at any higher precision that suits the computer best.

4.1.3 The Arithmetic Geometric Mean, $\mathbb{F}_{2^n}$

The latest, and not very well-documented, method for counting the points on an elliptic curve is the Arithmetic Geometric Mean-method (AGM). It is due to Mestre, Gaudry and Harley and works not with the j-invariants of the lifted curves $E^{\uparrow}$ but with the coefficients of the elliptic curve equation. Since information on this algorithm is scarce, our treatment of the method will be rudimentary at best. For definitions, please refer to Section 4.1.2.

We define the AGM $M : R^2 \to R^2$, where R is an unramified extension of the 2-adic numbers, by

$$M(a, b) = \left(\frac{a+b}{2}, \sqrt{ab}\right). \tag{4.3}$$

If $E^{\uparrow}_{ab}$ is an elliptic curve given by an equation

$$y^2 = x(x - a^2)(x - b^2),$$

and $E^{\uparrow}_{a'b'}$ is an elliptic curve given by an equation

$$y^2 = x(x - a'^2)(x - b'^2),$$

where $(a', b') = M(a, b)$, then $E^{\uparrow}_{ab}$ and $E^{\uparrow}_{a'b'}$ are 2-isogenous, i.e. linked by a 2-isogeny (c.f. the 2-Frobenius). In fact, with our previous notation, if $E^{\uparrow}_{ab} = E^{\uparrow}_{i}$, then $E^{\uparrow}_{i+1} = E^{\uparrow}_{a'b'}$ (we need to make an appropriate change of variables first). Also, if $E^{\uparrow}_{i}$ is known to precision d, then, after applying the AGM, $E^{\uparrow}_{i+1}$ is known to precision $d + 1$.

Furthermore, if $(a', b') = M(a, b)$ and $(a'', b'') = M(a', b')$, then it turns out that $\operatorname{tr} \phi^{\uparrow}_{2} = \pm a'/a'' \mod 2^d$ for a certain d. So, in order to calculate the trace of the Frobenius morphism, we can first calculate (a, b) with enough precision for some $E^{\uparrow}_{i}$ using the AGM and then cycle through all the $E^{\uparrow}_{i}$ once again and calculate the quotient as described above. For a more detailed description of this, see Algorithm 20.

Implementation details

For the most part, see the implementation details under Section 4.1.2. The only new thing we need is the square-root computation in R. For this we choose to do a Newton iteration on the function $f(x) = x^2 - a$. However, as $f'(x) \equiv 0 \mod 2$ and $x \equiv 1 \mod 8$ initially, we need to use the iteration

$$x_{k+1} = x_k - \frac{(x_k^2 - a)/2}{x_k}, \qquad x_0 = 1 \mod 4,$$

which converges quadratically and will give us $\sqrt{a} \equiv 1 \mod 4$.

4.2 Elliptic Curve Selection Criteria

We now focus on the problem of choosing elliptic curves that are suitable for cryptography. As stated in Section 1.4, the best method to solve the discrete logarithm problem in the elliptic curve group is Pollard's ρ-method. Because of this, we require the elliptic curve

Algorithm 20 The Arithmetic Geometric Mean-method for counting points on an elliptic curve

Require: $E/\mathbb{F}_{2^n} : y^2 + xy = x^3 + A$
Ensure: $\operatorname{tr} \phi_{2^n}$
  $a \leftarrow 1 + 8A$, $b \leftarrow 1$
  $m \leftarrow \lceil n/2 \rceil + 4$
  for $i = 4$ to m **do**
    $(a, b) \leftarrow \left(\frac{a+b}{2}, \sqrt{ab}\right) \mod 2^i$
  end for
  $c \leftarrow a$
  for $i = 1$ to n **do**
    $(a, b) \leftarrow \left(\frac{a+b}{2}, \sqrt{ab}\right) \mod 2^m$
  end for
  $t \leftarrow c/a \mod 2^{m-2}$
  if $t > 2\sqrt{2^n}$ **then**
    return $-t$
  else
    return t
  end if

group to have a large subgroup of prime order. There are, however, cases where there exist better algorithms than Pollard's ρ-method for solving the discrete logarithm problems, cases which should therefore be avoided. They are

- Supersingular elliptic curves [12]. For these curves, an index-calculus attack can be used which solves the discrete logarithm problem in subexponential time. In our case, curves defined over fields of characteristic two, this means testing whether $j = 0$ which is quickly done.

- Curves for which the trace of Frobenius is equal to 1, that is curves where $\#E(\mathbb{F}_p) = p$. This attack is not applicable to curves defined over fields of characteristic two. [16].

- So-called Koblitz curves [26]. These are curves defined over $\mathbb{F}_{2^n}$ on the form

$$y^2 + xy = x^3 + ax^2 + 1,$$

  where $a \in \{0, 1\}$. For these curves, the discrete logarithm computation can be sped up by a factor $\sqrt{n}$. Still they are common as it is often decided that the arithmetic speed-ups that are possible using these curves out-weigh the loss of security.

- Curves defined over $\mathbb{F}_{2^n}$, where n is composite [5]. The discrete logarithm problem on these curves may in certain cases be transformed to a subexponential problem on a hyperelliptic curve.

To summarize, this means that in order to construct an elliptic curve E over $\mathbb{F}_{2^n}$, given by an equation

$$y^2 + xy = x^3 + ax^2 + b,$$

we:

1. Make sure that n is a prime.
2. Randomly select a and b in the elliptic curve equation and make sure that
   (a) the curve is not supersingular, that is $j = 1/b \neq 0$,
   (b) the coefficients a and b are sufficiently large.
3. Count the number of points on $E(\mathbb{F}_{2^n})$ and check that this number has a large prime divisor, preferably of the same order as 2^n.

Counting the number of points is quickly done using Satoh's algorithm or the AGM. However, had we used Schoof's algorithm, we could have quickly checked whether $\#E(\mathbb{F}_{2^n})$ had small prime divisors, stop the counting and select a new curve. Using Satoh's algorithm, or the AGM, there is no such quick check and we need to run through algorithm in its entirety. One possible circumvention of this problem is to first run through Schoof's algorithm for some small primes $l_i < 19$ for example, and use Satoh's algorithm, or AGM, for the candidate curves that survive this test [4]. Since Satoh's algorithm is so much faster than Schoof's, this turns out to be a very efficient method for curve selection.

Bibliography

[1] I. Blake, G. Seroussi, and N. Smart. *Elliptic curves and cryptography*. London Mathematical Society, 1999.

[2] R. Crandall and C. Pomerance. *Prime Numbers*. Springer-Verlag, 2001.

[3] M. Fouqet, P. Gaudry, and R. Harley. On Satoh's algorithm and its implementation. *Journal of the Ramanujan Mathematical Society*, 15:281–318, 2000.

[4] M. Fouquet, P. Gaudry, and R. Harley. Finding secure curves with the Satoh-FGH algorithm and an early-abort strategy. In *Advances in Cryptology—Eurocrypt 2001*, number 2045 in Lecture Notes in Computer Science, pages 14–29. Springer-Verlag, 2001.

[5] S. Galbraith and N. Smart. A cryptographic application of Weil descent. In *Codes and Cryptography*, number 1746 in Lecture Notes in Computer Science, pages 191–200. Springer-Verlag, 1999.

[6] I.Blake, R. Roth, and G. Seroussi. Efficient arithmetic in $GF(2^n)$ through palindromic representation. *HPL-98-134*, 1998.

[7] IEEE. *IEEE standard specifications for public-key cryptography*. IEEE, 2000.

[8] D. E. Knuth. *The Art of Computer Programming—Seminumerical Algorithms*. Addison-Wesley, 1998.

[9] A. Lenstra and E. Verheul. Selecting cryptographic key sizes. In *Proceedings of PKC 2000*, number 1751 in Lecture Notes in Computer Science, pages 446–465. Springer-Verlag, 2000.

[10] J. López and R. Dahab. Improved algorithms for elliptic curve arithmetic in $GF(2^n)$. In *Selected Areas in Cryptography—SAC'98*, number 1556 in Lecture Notes in Computer Science. Springer-Verlag, 1998.

[11] J. Lubin, J. P. Serre, and J. Tate. Elliptic curves and formal groups. Scanned copies `http://www.ma.utexas.edu/users/voloch/lst.html`. Lecture notes prepared in connection with the seminars held at the Summer Institute on Algebraic Geometry, Whitney Estate, Woods Hole, Massachusetts, July 6–31 1964.

[12] A. Menezes, T. Okamoto, and S. Vanstone. Reducing elliptic curve logarithms to logarithms in a finite field. *IEEE Transactions on Information Theory*, 39:1639–1646, 1993.

[13] A. Menezes, P. van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 2001.

[14] J. S. Milne. Elliptic curves. http://www.jmilne.org/math/. Lecture notes for Math 679, University of Michigan, Winter 1996.

[15] National Institute of Standards and Technology. FIPS 180-1 secure hash standard. http://www.itl.nist.gov/fipspubs/index.htm.

[16] H.-G. Rück. On the discrete logarithm in the divisor class of curves. *Mathematics of Computation*, 68:805–806, 1999.

[17] T. Satoh. The canonical lift of an ordinary elliptic curve over a finite field and its point counting. *Journal of the Ramanujan Mathematical Society*, 15:483–494, 2000.

[18] R. Schoof. Elliptic curves over finite fields and the computation of square roots $\mod p$. *Mathematics of Computation*, 44(170):483–494, 1985.

[19] R. Schoof. Counting points on elliptic curves over finite fields. *Journal de Théorie des Nombres de Bordeaux*, 7:219–254, 1995.

[20] R. Schroeppel, H. Orman, S. O'Malley, and O. Spatscheck. Fast key exchange with elliptic curve systems. In *Advances in Cryptology—Crypto '95*, number 963 in Lecture Notes in Computer Science, pages 43–56. Springer-Verlag, 1995.

[21] J. H. Silverman. *The Arithmetic of Elliptic Curves*. Springer-Verlag, 1986.

[22] B. Skjernaa. Satoh's algorithm in characteristic 2. *preprint*, 2000. http://home.imf.au.dk/skjernaa/.

[23] J. Solinas. Efficient arithmetic on Koblitz curves. *Designs, Codes and Cryptography*, (19):195–249, 2000.

[24] J. Vélu. Isogénies entre courbes elliptiques. *C.R. Acad. Sc. Paris*, 273:238–241, 1971.

[25] F. Vercauteren, B. Preneel, and J. Vandewalle. A memory efficient version of Satoh's algorithm. In *Advances in Cryptology—Eurocrypt 2001*, number 2045 in Lecture Notes in Computer Science, pages 1–13. Springer-Verlag, 2001.

[26] M. J. Wiener and R. J. Zuccherato. Faster attacks on elliptic curve cryptography. In *Selected Areas of Cryptography*, number 1556 in Lecture Notes in Computer Science, pages 190–200. Springer-Verlag, 1999.