MASTER'S DEGREE THESIS

Master of Science in Advanced Mathematics and Mathematical Engineering

Title: Optimizations in the Computation of Pairings, and Cryptographic Applications

Author: David Sánchez Charles

Advisors: Javier Herranz Sotoca, Jorge Luis Villar Santos

Department: Matemàtica Aplicada IV

Academic year: 2012-2013

Facultat de Matemàtiques i Estadística

UNIVERSITAT POLITÈCNICA DE CATALUNYA

Universitat Politècnica de Catalunya
Facultat de Matemàtiques i Estadística

Tesi de màster

Optimizations in the Computation of Pairings, and Cryptographic Applications

David Sánchez Charles

Advisor: Javier Herranz Sotoca, Jorge Luis Villar Santos

Departament de Matemàtica Aplicada IV

A todos los amigos, profesores y familiares que me han apoyado, y aguantado, durante estos últimos años. Muchísimas gracias a la Fundación 'la Caixa', por confiar en mi como becario de la promoción 2012-2013.

Abstract

Key words: curvas elípticas, criptografía de clave pública, apareamientos sobre curvas elípticas, algoritmo de Miller

MSC2000: 14H52, 11T71, 68P25

En la criptografía de clave pública actual el uso de curvas elípticas es muy habitual: ofrecen una seguridad similar a los ya utilizados cuerpos finitos, pero con unas claves mucho más cortas. Además, la introducción de los apareamientos sobre curvas elípticas ha hecho posible el desarrollo de soluciones en escenarios en que la criptografía previa no era capaz de encontrar soluciones eficientes. Algunos ejemplos son el cifrado basado en la identidad del receptor, o el cifrado basado en atributos. Para que estos esquemas criptográficos puedan ser utilizados de manera eficiente en situaciones prácticas, es muy interesante encontrar optimizaciones en el cálculo de los apareamientos bilineales necesarios.

En este trabajo final de máster describimos y completamos el trabajo hecho por Costello *et al.* [**11**], sobre una optimización en el cálculo de apareamientos basada en precálculos. La optimización explota el hecho que un argumento del apareamiento se mantiene fijo para muchas evaluaciones diferentes (por ejemplo, en esquemas criptográficos donde una misma clave secreta es usada en muchas ejecuciones del protocolo de descifrado). Damos una cota superior a la complejidad en ambos algoritmos de precálculo y evaluación, en el peor caso posible.

También hemos implementado, por primera vez, esta optimización en el lenguaje de programación C usando la librería PBC [**27**] de criptografía basada en apareamientos. De esta manera hemos sido capaces de comparar los tiempos dados por la optimización de Costello con un apareamiento ya implementado previamente (parcialmente optimizado). También hemos dado una estimación de cuántos apareamientos se debe calcular para amortizar el tiempo gastado en precálculos. Los datos obtenidos muestran que esta optimización es factible en usos realistas de la criptografía basada en curvas elípticas.

Abstract

Keywords: elliptic curve, public-key cryptography, pairings, Miller's algorithm

MSC2000: 200014H52, 11T71, 68P25

In modern public-key cryptography the use of elliptic curves is very common: they offer a similar security level as finite field schemes, but with shorter keys. Moreover, the introduction of pairings over elliptic curves has made possible the development of solutions in scenarios where previous cryptography had failed in finding efficient solutions. Some examples are identity based encryption and attribute based encryption. In order to use such cryptographic schemes efficiently in practical situations, it is very interesting to find optimizations in the computation of the required pairings.

In this master thesis we describe and complete the work done by Costello *et al.* [**11**], on an optimization in the computation of bilinear pairings based on precomputation. The optimization exploits the fact that one or the arguments of the pairing remains fixed for many different evaluations (for instance, in cryptographic schemes where the same secret key is used in several executions of the decryption protocol). We give an upper bound of the complexity of both the precomputation and the evaluating stages, in the worst-case scenario.

We have also implemented, for the first time, this optimization in C using the pairing library PBC [**27**]. In this way we have been able to compare the execution times of this optimization with an already implemented (and partially optimized) pairing. We also give an estimate of how many pairing evaluations are needed in order to amortize the cost of the off-line precomputations. Surprisingly not so many evaluations are needed, and therefore this optimization is very useful in current implementations of cryptography over elliptic curves.

Notation

K	Any finite field
K^*	Multiplicative group of K
$\overline{K}$	The algebraic closure of a field K
$\mathbb{F}_q$	A finite field of q elements
$V(f)$	The set of zeros of a polynomial $f \in K[X, Y, Z]$
$E(K)$	An elliptic curve defined over a field K
k	The embedding degree of an elliptic curve.
μ_n	The subgroup of n-th roots of unity. The underlying field is given by context.

Contents

Introduction

Public key cryptography is based on the existence of a pair of maps E and D such that $D \circ E$ is the identity map, and the information provided from E is not enough to compute the function D. If such a pair of maps exists, a message m can be encrypted as $E(m)$, and only the owner of the pair (E, D) is able to retrieve the message $D(E(M)) = m$. Historically, some examples of such pairs were defined thanks to the properties of integers modulo n and finite fields [**1**]. But thanks to Miller and Koblitz [**16**] [**17**], in 1985, a new mathematical object started to be interesting in cryptography: the group of points of an elliptic curve. The complicated geometric definition of addition in this group allowed cryptographers to hide an integer n in a multiple of a point P as nP. It was assumed that retrieving n from P and nP (e.g. the elliptic discrete logarithm problem) was an unfeasible operation for points with a large order, and an upper bound of the complexity of the elliptic discrete logarithm problem was given by Menezes, Okamoto and Vanstones [**18**] in 1991. In that paper a very useful tool was introduced: the existence of a bilinear map, called pairing, which sends a pair of points in an elliptic curve to a (well-studied) finite field.

Pairings over elliptic curves offered a solution to cryptographic problems that remained open, and also lead to the introduction of new cryptographic primitives. We will review three examples of cryptographic schemes where pairings play an important role: identity based encryption, where the receiver's identity is used as the key to encrypt data for him; the BLS signature scheme, used to generate very short signatures; and an attribute based encryption scheme in which data can be encrypted so that only users (maybe many) with attributes satisfying the decryption policy chosen by the sender can decrypt the ciphertext correctly.

Even though pairings over elliptic curves produce more secure, and useful, schemes, they are not massively used in practice because of the very expensive cost of the computation of a single pairing. A lot of work has been done in order to make pairings appealing to computer scientists. At a very early stage a lot of existing optimizations for the modular exponentiation algorithm were adapted to the evaluation of pairings [**2**], since the known algorithm to compute pairings is very similar to the algorithm for fast exponentiation. But this was not enough, and the search of new optimizations led to a study of good elliptic curves to be used in cryptography. Some of them offered a fast point operation and a reduction in the total number of operations [**13**] [**14**]. This last bunch of optimizations should convince people that

pairings are not a so expensive operations, and pairing-based cryptosystems should be used in real-life situations.

When pairings are used in some particular scenarios one may find further optimizations. For example, when a lot of pairing evaluations are done with one of the arguments of the pairing fixed, then off-line precomputations can be done in order to reduce later the cost of each pairing evaluation. One of these precomputation optimization was explained by Costello *et al.* in [**11**] and [**12**]. Later in [**15**] Scott gave some examples of public key cryptographic schemes where this optimization could be useful, and he also hinted a possible adaptation of the optimization given by Costello *et al.* to the computation of multipairings. None of these works included a explicit implementation of the algorithms of this optimization, and they did not give the execution time of the off-line precomputation phase.

In this master thesis we do a theoretical study of Costello *et al.* optimization, by completing some aspects of the off-line precomputations. We first give a theoretical estimation of how costly is the precomputation phase, and then we implement the whole optimization to show that it can be perfectly used to improve the global efficiency of many practical cryptographic scenarios. The master thesis is divided in five chapters.

In Chapter 1, we review the basics on finite fields, elliptic curves and pairings. Some useful basic optimizations of both finite fields and pairings are described. We do not consider optimizations on elliptic curves, like the use of different coordinates to speed-up the point operation, even though all pairing optimizations could benefit from them.

In Chapter 2 we explain three known cryptographic schemes with a heavy use of pairings. Two of them benefit directly from the Costello *et al.* optimization, but the third one needs some additional work to adapt the optimization to the scenario of multipairings.

In Chapter 3 we describe the Costello *et al.* optimization and we do a theoretical study of it. We extend the work of Costello *et al.* in [**11**] in two ways: firstly by not considering only a specific structure of the parameter (even though the elliptic curves can be chosen so that their order has few bits different from 0, not all of them are zero), and secondly by considering its application to multipairings as sketched in [**15**].

In Chapter 4 we give numerical results of our implementation of this optimization (the source code can be found in Appendix A). We chose an existing library, PBC Library, to implement this optimization and we compare it to an already implemented, and partially optimized, algorithm for the evaluation of a pairing. The most surprising result of these experiments is the (low) number of pairings needed to amortize the cost of the off-line precomputation phase of the new optimization: in all our test cases a maximum of 6 evaluations are enough.

Finally, in Chapter 5 we summarize the work done in this master thesis and we list some possible lines for related future work.

Chapter 1
Mathematical Preliminaries

1.1. Finite fields

It is well-known that if p is prime, then $\mathbb{Z}/p\mathbb{Z}$, denoted by $\mathbb{F}_p$, is a finite field. This mathematical object will be the basic structure of our study. Sometimes we will call it **small field** or **base field**.

In our base field we have three operations available: addition, multiplication, and the inversion of an element. Among these three operations, the most computationally expensive one is inversion: in the worst case, it requires about $\log_{10} x$ multiplications to compute the inverse of x [**3**]. Therefore, it is possible to obtain a better algorithm if the number of divisions is reduced, even if that increases the number of multiplications and additions, as one can see in Example 1.1.1. In Table 1.1 we compare the execution times of the three operations over finite fields of different sizes.

bits in p	$x + y$	$x \cdot y$	x^{-1}
32	32	89	370
58	31	88	553
124	22	60	1489
196	27	82	2480
437	30	179	5271
677	34	386	8489
1357	43	1294	18610
5295	173	17175	118335

TABLE 1.1. Time, in μs, to compute 1000 operations in $\mathbb{F}_p$

EXAMPLE 1.1.1 (Montgomery's Trick). Suppose we have n elements $x_1, x_2, \ldots, x_n$ and we want to compute their inverses $x_1^{-1}, x_2^{-1}, \ldots, x_n^{-1}$. Montgomery's Trick shows how we can compute all this with only 1 division.

We start by computing

$$a_i = \prod_{j=1}^{i} x_i \qquad\qquad i = 1, \ldots, n.$$

Then, we compute the inverse of a_n, and

$$\begin{aligned} x_n^{-1} &= a_n^{-1} \cdot a_{n-1} & a_{n-1}^{-1} &= a_n^{-1} \cdot x_n, & \\ x_i^{-1} &= a_i^{-1} \cdot a_{i-1} & a_{i-1}^{-1} &= a_i^{-1} \cdot x_i & \text{for } i = n-1, \ldots 2, \\ x_1^{-1} &= a_1^{-1}. & & & \end{aligned}$$

In this way, we replace $n-1$ inversions with $3(n-1)$ multiplications. As one can see in Table 1.1, one division is always more expensive than 3 multiplications. So, this way of computing simultaneously n inverses is faster than doing it independently.

In [**4**] authors used this technique to improve an implementation of Masked AES — a way of hiding how many operations, and which ones, are done in an AES encryption so that no information is provided to a possible attacker. Later in this chapter we will use Montgomery's trick to reduce the number of divisions in multiple evaluations of Miller's algorithm.

1.1.1. Extensions of finite fields. The finite fields of the form $\mathbb{F}_p$ are not the whole set of finite fields: given an integer n, and a prime p, we can find a finite field with p^n elements. To construct this bigger field we need an irreducible polynomial f of degree n with coefficients over $\mathbb{F}_p$, and then

$$\mathbb{F}_{p^n} := \mathbb{F}_p[X]/(f(X)).$$

We will denote this field as the **full field**.

Since $\dim_{\mathbb{F}_p} \mathbb{F}_{p^n} = n$, the set $\{1, x, \ldots, x^{n-1}\}$ is a basis and we can interpret an element of $\mathbb{F}_{p^n}$ as an n-tuple of elements — or a polynomial of degree less than n — of the base field $\mathbb{F}_p$. It seems natural to compare the cost of operations in $\mathbb{F}_{p^n}$ and in $\mathbb{F}_p$. A full addition in the full field needs n additions in the base field. The best known method to compute the greatest common divisor of two polynomials needs about $3n^2$ base operations [**5**], some of them are divisions. Therefore, we need about $3 \cdot n^2$ operations to compute an inversion in the full field. This could be greatly improved for some specific fields — see Examples 1.1.4, 1.1.5 and 1.1.6. A multiplication in $\mathbb{F}_{p^n}$ may need more than n^2 multiplications in the base field, but this can also be improved for some field as we can see in Examples 1.1.2, 1.1.3 and 1.1.6. In Table 1.2 we can find a comparison of these operations in different fields.

Example 1.1.2 (Multiplication in $\mathbb{F}_{p^2}$). We can use Karatsuba's algorithm for the multiplication of polynomials [**6**] to reduce the number of base multiplications needed. We will explain this optimization for

$$\mathbb{F}_{p^2} = \mathbb{F}_p[X]/(X^2 + \alpha)$$

where $\alpha \in \mathbb{F}_p$, but it can be adapted to any polynomial.

We can represent an element in $\mathbb{F}_{p^2}$ as a polynomial of degree 1. To multiply two elements $a = a_0 + a_1 X$ and $b = b_0 + b_1 X$ we first compute:

$$D_0 = a_0 b_0 \qquad D_1 = a_1 b_1 \qquad D_{0,1} = (a_0 + a_1) \cdot (b_0 + b_1)$$

bits in p	n	$x+y$	$x \cdot y$	x^{-1}
190	6	203	5268	16774
196	6	221	6055	18410
102	12	281	11945	26568
252	6	214	5961	19443
127	12	283	12510	28162
250	10	140	22185	88931
437	6	295	10357	31393
677	6	278	18210	50974
2046	10	322	263008	1285844
2456	12	1031	548559	795344
5295	6	1333	672896	1411283

TABLE 1.2. Time, in μs, to compute 1000 operations in $\mathbb{F}_{p^n}$

And then

$$\begin{aligned} a \cdot b &= D_1 X^2 + (D_{0,1} - D_0 - D_1)X + D_0 \\ &= (D_{0,1} - D_0 - D_1)X + D_0 - \alpha D_1 \end{aligned}$$

We only used 4 multiplications and 5 additions, instead of 5 multiplications and 2 additions.

EXAMPLE 1.1.3 (Multiplication in $\mathbb{F}_{p^3}$). Suppose we have the field $\mathbb{F}_{p^3}$ constructed as follows

$$\mathbb{F}_{p^3} = \mathbb{F}_p[X]/(X^3 + \beta)$$

for some $\beta \in \mathbb{F}_p$. We can represent an element in $\mathbb{F}_{p^3}$ as a polynomial of degree 2. Then, to multiply two elements $a = a_0 + a_1 X + a_2 X^2$ and $b = b_0 + b_1 X + b_2 X^2$ one can use Karatsuba's algorithm in the case of polynomials of degree 2

$$D_0 = a_0 b_0 \qquad D_1 = a_1 b_1 \qquad D_2 = a_2 b_2$$

$$D_{0,1} = (a_0 + a_1)(b_0 + b_1) \quad D_{0,2} = (a_0 + a_2)(b_0 + b_2) \quad D_{1,2} = (a_1 + a_2)(b_1 + b_2)$$

Then

$$\begin{aligned} a \cdot b =& D_2 X^4 + (D_{1,2} - D_1 - D_2)X^3 + (D_{0,2} - D_2 - D_0 + D_1)X^2 \\ &+ (D_{0,1} - D_1 - D_0)X + D_0 \\ =& (D_{0,2} - D_2 - D_0 + D_1)X^2 + (D_{0,1} - D_1 - D_0 - \beta D_2)X \\ &+ D_0 - \beta(D_{1,2} - D_1 - D_2) \end{aligned}$$

We only used 8 multiplications and 16 additions, instead of 10 multiplications and 5 additions.

EXAMPLE 1.1.4 (Inversion in $\mathbb{F}_{p^2}$). There are better division algorithms if we know something more about the field. For example, if

$$\mathbb{F}_{p^2} = \mathbb{F}_p[X]/(X^2 + \alpha)$$

for some $\alpha \in \mathbb{F}_p$, then the inverse of $a + bX$ is

$$\frac{a}{a^2 - \alpha b^2} + \frac{-b}{a^2 - \alpha b^2} X$$

whose computation needs only five base multiplications, one base division and one addition.

EXAMPLE 1.1.5 (Inversion in $\mathbb{F}_{p^3}$). As in the previous example, we can improve the inversion of an element $a + bX + cX^2$ if

$$\mathbb{F}_{p^3} = \mathbb{F}_p[X]/(X^3 + \beta)$$

by first computing

$$\triangle = a^3 - \beta((b^2 - 3ac)b - \beta c^3),$$

and then the inverse is

$$\triangle^{-1} \left[a^2 + bc\beta - (\beta c^2 + ab)X + (b^2 - ac)X^2\right].$$

We only needed 18 multiplications, 1 base inversion and 7 additions.

1.1.2. Towers of finite fields. Sometimes it is useful to construct an intermediate field between the base and the full fields,

$$\mathbb{F}_p \subset \mathbb{F}_{p^d} \subset \mathbb{F}_{p^n}, \quad \text{with } d|n.$$

To construct the full field from the new $\mathbb{F}_{p^d}$ one may use the same idea as before, replacing the role of $\mathbb{F}_p$ with the new field.

One of the biggest utilities of towers of finite fields is to improve the computation of multiplications and inversions; the next example shows how to use the formulas given in Examples 1.1.2 and 1.1.3 to compute efficiently a multiplication in $\mathbb{F}_{p^6}$.

EXAMPLE 1.1.6 (Multiplication and inverse in $\mathbb{F}_{p^6}$). Suppose we have the field $\mathbb{F}_{p^6}$ constructed as follows

$$\mathbb{F}_{p^6} = \mathbb{F}_{p^3}[X]/(X^2 + \alpha)$$

for some $\alpha \in \mathbb{F}_{p^3}$. By using the formulas in Example 1.1.5, in order to multiply two elements of $\mathbb{F}_{p^6}$ we need to do 4 multiplications in $\mathbb{F}_{p^3}$ and 5 additions in $\mathbb{F}_{p^3}$ — or 15 additions in $\mathbb{F}_p$.

Now, we can use the formulas in Example 1.1.2 to transform these 4 multiplications in $\mathbb{F}_{p^3}$ into $4 \cdot 8 = 32$ multiplications in $\mathbb{F}_p$ and $4 \cdot 10$ additions in $\mathbb{F}_p$. Therefore, we can multiply in $\mathbb{F}_{p^6}$ with 32 base multiplications and 55 base additions, instead of 41 multiplications and 30 additions.

This technique can also be applied to compute the inverse of an element in $\mathbb{F}_{p^6}$: we will use 5 multiplications and one inverse in $\mathbb{F}_{p^3}$. Therefore we will need $5 \cdot 8 + 18 = 58$ base multiplications and 1 division, instead of the $3 \cdot 6^2 = 108$ base operations needed by the algorithm in [**5**].

1.2. Elliptic curves over finite fields

In this introduction we are going to define elliptic curves as the projective smooth curves of degree 3, so that they have a nice geometric property that allows us to define a group law over their points. But, in fact, there is a more algebraic definition of elliptic curves: they are all curves of genus 1. This algebraic property allows us to define a group law over their *divisors* which translates into the geometric law that we are going to explain.

DEFINITION 1.2.1. A projective curve in the projective plane $\mathbb{P}^2(K)$ is the set of zeros of a given homogeneous polynomial $f \in K[X, Y, Z]$:

$$V(f) = \{[x : y : z] \in \mathbb{P}^2 \,/\, f(x, y, z) = 0\}.$$

If f is a homogeneous polynomial of degree d, then the curve $V(f)$ has degree d.

The curve is smooth if $(\frac{\partial f}{\partial X}, \frac{\partial}{\partial Y}, \frac{\partial}{\partial Z})(P)$ is not the zero vector for every point $P \in V(f)$ — and it defines the tangent line at P.

Every curve of degree 3 is defined by a polynomial of the form

$$aX^3 + bY^3 + cZ^3 + dXY^2 + eXZ^2 + fXYZ + gX^2Y + hX^2Z + iY^2Z + jYZ^2$$

but, if we want to make it smooth, we can do some projective transformations (in [**7**] one can see the details for the real case, even though it only uses that the underlying field does not have characteristic 2) to obtain a simpler equation:

DEFINITION 1.2.2. An elliptic curve $E(K)$ is a projective curve in $\mathbb{P}^2(K)$ that can be transformed into a curve defined by a polynomial

$$ZY^2 - h(X, Z)$$

where h is a homogeneous polynomial of degree 3 and $h(X, 1)$ has no repeated factors.

REMARK. There is only one point in $E(K)$ with $Z = 0$, which is $[0 : 1 : 0]$. This is the point at infinity and will be denoted by P_∞.

REMARK. Since the algebraic curve $V(ZY^2 - h(X, Z))$ is smooth, we can compute the tangent line at every point of an elliptic curve.

REMARK. In affine coordinates one usually says that an elliptic curve is the set of zeros of the polynomial

$$y^2 - h(x)$$

with $\deg h = 3$ and the resultant of h is different from 0 - so it has no repeated factors.

Bezout's Theorem on curves over algebraically closed fields says that the intersection of a curve of degree 3 and a line contains always 3 points (counting multiplicities). This is no longer true when we are working over general fields, but if we add an extra condition over the line we obtain a similar result:

COROLLARY 1.2.3. *Let $E(K)$ be an elliptic curve over a field K and L a line which intersects $E(K)$ in at least 2 points (counting multiplicities). Then L intersects $E(K)$ at a third point.*

This geometric property allows us to define a group law over the set of points of an elliptic curve

DEFINITION 1.2.4. Let P and Q be two points of $E(K)$, not necessarily different. Then the line PQ - the tangent of the curve at P - intersects at $E(K)$ in a third point R. Then we define $P + Q$ as the third point of intersection of the line RP_∞.

EXAMPLE 1.2.5 (Group law). Assume the elliptic curve $E(K)$, with $char(K) \neq 2, 3$, is defined by $y^2 = x^3 + bx + c$. Given two affine points $P = (x_P, y_P)$ and $Q = (x_Q, y_Q)$ we know that:

- The identity element is P_∞.
- $-P = -(x_P, y_P) = (x_P, -y_P)$.
- If $x_P \neq x_Q$ then the coordinates of $P + Q$ can be computed as follows. First compute the slope of the line PQ:
$$m = \frac{y_P - y_Q}{x_P - x_Q}$$
and then, the third point of intersection is (x_{P+Q}, y_{P+Q}):
$$x_{P+Q} = m^2 - x_P - x_Q$$
and
$$y_{P+Q} = -y_P - m(x_{P+Q} - x_P)$$
- The coordinates of $2P$ can be computed as follows. First we need the slope of the tangent line at P:
$$m = \frac{3x_P^2 + b}{2y_P}$$
an then, the third point of intersection is (x_{2P}, y_{2P}):
$$x_{2P} = m^2 - 2x_P$$
and
$$y_{2P} = -y_P - m(x_{2P} - x_P)$$

Similar to other commutative groups like $\mathbb{Z}_p$, the group of points of an elliptic curve offers an alternative to be used in cryptography since the discrete logarithm problem seems unfeasible in that group; that is, given a point $P \in E(\mathbb{F}_q)$ and a multiple $Q = aP = \sum_{i=1}^{a} P$ there is no general efficient method to retrieve the integer a.

But this group may have an efficient way to solve the generalized Decisional Diffie-Hellman problem: given points P, aP, Q, T decide whether $T = aQ$ or not. This problem can be efficiently solved by using bilinear maps defined over the elliptic curve. We are going to define now such a bilinear map — called *pairing* — and some known algorithms and optimizations to evaluate this pairing.

Before doing that, we need to know how applications over elliptic curves are defined. Not all applications will be useful since a point P of an elliptic curve has many representatives. We will ask for homogeneous polynomials of some degree d, since
$$g(\lambda X, \lambda Y, \lambda Z) = \lambda^d g(X, Y, Z)$$
is well defined for zeros of g.

DEFINITION 1.2.6. A polynomial function defined over the points of an elliptic curve $E(K) = V(f)$ is an homogeneous representative of the quotient

$$K[E] = K[X, Y, Z]/(f).$$

If g and h are two representatives of the same class, then $g - h \in (f)$ and therefore $g(P) = h(P)$ for every point P of the curve.

DEFINITION 1.2.7. The multiplicity of a polynomial function $g(x, y)$ at $(0, 0) \in E(K) = V(f(x, y))$ is

$$i_{(0,0)}(g, f) = \dim_{\overline{K}} \overline{K}[[x, y]]/(f, g),$$

where $\overline{K}[[x, y]]$ is the formal series ring on the variables X, Y and Z.

REMARK. This is a local property. Given a projective point P different from $[0 : 0 : 1]$ we need to do a projective transformation in order to send it to $[0 : 0 : 1]$ and then replace $Z = 1$ in both $f(X, Y, Z)$ and $g(X, Y, Z)$.

EXAMPLE 1.2.8. The multiplicity tells us how good is $V(g)$ as an approximation of $V(f) = E(K)$ at the point P.

For example,

$$\begin{aligned} i_{(0,0)}(y - x^3, y - x^3) &= \infty \\ i_{(0,0)}(y - x, y - x^3) &= 1 \\ i_{(0,0)}(y - x^2, y - x^3) &= 2 \end{aligned}$$

The last two multiplicities show us that $V(y - x)$ only goes through $(0, 0)$, but $y - x^2$ goes through $(0, 0)$ and is tangent to $V(y - x^3)$ in $(0, 0)$.

DEFINITION 1.2.9. The field of functions on an elliptic curve $E(K)$ is

$$K(E) = \text{Field of fractions } K[E].$$

where every function $\varphi \in K(E)$ can be written as $\varphi = g/h$ with g and h polynomial functions with the same homogeneous degree.

REMARK. The multiplicity at P can be extended to functions $\varphi = g/h \in K(E)$ by

$$i_P(g/h, f) = i_P(g, f) - i_P(h, f).$$

Now we can define principal divisors. This is an elegant tool to describe functions over elliptic curves and to check whether two functions are *equal* or not.

DEFINITION 1.2.10. A divisor of $E(K)$ is a formal sum of a finite number of points in $E(K)$; in other words, a divisor D is

$$D = \sum_{P \in E(K)} n_P(P)$$

where $n_P = 0$ except at a finite number of points, and no sum is done at all.

DEFINITION 1.2.11. Given a function $\varphi \in K(E)$, with $E = V(f)$, its divisor is

$$(\varphi) = \sum_{P \in E(K)} i_P(\varphi, f)(P).$$

All divisors D such that there exists a function $\varphi \in K(E)$ with $D = (\varphi)$ are called *principal* divisors.

DEFINITION 1.2.12. Two divisors D_1 and D_2 are equivalent if $D_1 - D_2$ is a principal divisor.

Principal divisors are totally characterized:

THEOREM 1.2.13. *Let D be a divisor of an elliptic curve, then D is principal if, and only if, $\sum_{P \in E(K)} n_P = 0$ and $\sum_{P \in E(K)} n_P \cdot P = P_\infty$.*

Furthermore, $(\varphi) = 0$ if, and only if, φ is constant.

PROOF. Corollary 3.3.5 in [**8**]. □

This result is what allows us to check if two functions are *equal* — except by, maybe, a multiplicative constant: if $(\varphi) = (\Psi)$ then $(\varphi/\Psi) = 0$ and φ/Ψ is a constant function.

1.3. Pairings on elliptic curves

DEFINITION 1.3.1. A pairing is a map $e : G_1 \times G_2 \to G_3$ with $G_1 = \langle P \rangle, G_2 = \langle Q \rangle$ cyclic groups and G_3 a group such that

- $e(P, Q) \neq 1$
- $e(aP, bQ) = e(P, Q)^{ab}$ for every choice of $a, b \in \mathbb{Z}$.

We are interested in pairings that can be computed efficiently.

Not every choice of the groups G_1 and G_2 allows us to find a suitable pairing. But elliptic curves over finite fields are an example of an algebraic structure capable of producing this kind of bilinear maps.

These two groups are defined as follows in the case of elliptic curves. G_1 can be defined as the multiples of a point $P \in E(\mathbb{F}_p)$, but with order n coprime to $p = char(K)$. Then, let k be the minimum possible positive integer such that $n | p^k - 1$. This value of k is called the embedding degree of the point in the curve. Then we can choose a point Q in $E(\mathbb{F}_{p^k})$ which is not a multiple of $P \in E(\mathbb{F}_p) \subset E(\mathbb{F}_{p^k})$. This can always be done because given an integer $n > 1$ coprime with p the subgroup of points of n-torsion is isomorphic to $\mathbb{Z}_n \times \mathbb{Z}_n$. Finally, the group G_3 is the set of n-th roots of 1 in $\overline{\mathbb{F}_p} \subset \mathbb{F}_{p^k}$.

REMARK. For some special elliptic curves, called *supersingular*, one may define this bilinear map as $e : G_1 \times G_1 \to \mu_n$, where $G_1 = \langle P \rangle \subset \mathbb{F}_{p^m}$. These elliptic curves allow us to have $G_1 = G_2$ but by using bigger base fields.

It was very recently when Joux proved [**26**] that one can compute *quickly* the discrete logarithm in μ_n for every choice of a supersingular elliptic curve over $\mathbb{F}_{2^{257}}$. And therefore, this kind of groups are not very useful in cryptography: if we get two

points P and aP, it is *easy* to retrieve the value of a by first computing $g = e(P, aP)$ and then computing the discrete logarithm of g, in μ_n.

That is why nowadays pairings are almost always defined in the asymmetric setting ($G_1 \neq G_2$), as we have done.

DEFINITION 1.3.2. Given two points P, Q of n-torsion, we define the Tate pairing of P and Q as follows: let f_P be a function such that $\mathrm{Div}(f_P) = n(P) - n(P_\infty)$. Let R be a point of n-torsion different from $P_\infty, P, -Q, P - Q$ and consider the divisor $D_Q = (Q + R) - (R)$. Then

$$e_T(P, Q) = f_P(D_Q)^{(q^k-1)/n}$$

where k is the embedding degree.

This definition show us that with two points of n-torsion we can define a map $e : \langle P\rangle \times \langle Q\rangle \to \mu_n$, where μ_n are the n-th roots of unity in $\mathbb{F}_{p^k} \subset \overline{\mathbb{F}_p}$. But this map does not always satisfy the properties we wanted: $e_T(P, P) = 1$ for every point P.

REMARK. The final exponentiation allows us to use any possible choice of the function f_P, which is not unique.

It is also true that the Tate pairing is well defined for every divisor D_Q equivalent to $(Q) - (P_\infty)$.

THEOREM 1.3.3. *There exist two points P and Q of n-torsion, with n coprime with $p = char(\mathbb{F}_p)$ such that the map*

$$e_T : \langle P\rangle \times \langle Q\rangle \to \mu_n$$

is a pairing, and it is a surjective map.

PROOF. This can be found in [**9**]. The idea is that the map $e_T(P, -)$ is constant if, and only if, $P = P_\infty$. Then for sure there exist two points P and Q such that $e_T(P, Q) \neq 1$. Now consider the image by e_T of all pairs of points of n-torsion. We know it is not trivial, and it is a subgroup of μ_n equal to μ_d, with $d \leq n$. Then $e_T(R, S)^d = 1$ for all points R and S, i.e. $e_T(dR, S) = 1$ for all choices of R and S. Since the map $e_T(dR, -)$ is constant, $dR = P_\infty$. This means that all points of n-torsion are d-torsion points. But this is only possible if $d = n$. □

DEFINITION 1.3.4. A multipairing of points P_i with Q_i, for $i = 1, \ldots, m$ is the multiplication of m parings $e(P_i, Q_i)$:

$$e(\{P_i\}_{i=1}^m, \{Q_i\}_{i=1}^m) = \prod_{i=1}^{m} e(P_i, Q_i).$$

1.3.1. Miller's algorithm. The definition of the Tate pairing does not tell us how we can find the map f_P such that $(f_P) = n(P) - n(P_\infty)$. But there is a very simple algorithm that computes it.

Instead of computing directly f_P we are going to compute a lot of intermediate steps f_i in which $(f_i) = i(P) - (iP) - (i - 1)(P_\infty)$. This family of functions have the following properties:

- $f_n = f_P$ since

$$\begin{aligned}(f_n) &= n(P) - (nP) - (n-1)(P_\infty) \\ &= n(P) - (P_\infty) - (n-1)(P_\infty) \\ &= (f_P)\end{aligned}$$

- f_0 is constant, because

$$(f_0) = 0(P) - (0P) + (P_\infty) = 0$$

  and a constant zero divisor means a constant function.
- There exists a relation

$$f_{i+j} = f_i \cdot f_j \cdot \frac{l_{iP,jP}}{v_{(i+j)P}}$$

  where $l_{P,Q}$ is the line that goes through points P and Q, and v_P is the vertical line through P. This relation holds because the divisors of both applications are the same.

These last two properties resemble two key properties of exponentiation: $e^0 = 1$ and $e^{i+j} = e^i \cdot e^j$. In fact, Miller's algorithm is an adaptation of a famous algorithm for fast exponentiation: we consider the representation of n in base 2 and starting with $f_0 = 1$ we update the value of the pairing using the information of the bits of n until we compute f_n. In Algorithm 1 we give a sketch of this algorithm.

In the rest of this master thesis we are going to call *doubling step* to the part of the algorithm that does not depend on the bit n_i. On the other hand, the *adding step* is the part of the algorithm that is only executed when $n_i = 1$. We are very interested in reducing the number of times the *adding step* is executed, and this could be done if a good n is chosen. But, even if we cannot choose a n with few ones in its bit representation, we can reduce the number of times the *adding step* is executed by replacing the list (n_i) with its NAF representation — no two ones are adjacent, but we add the possibility of a -1 bit. This is possible because the relation $f_{i+j} = f_i \cdot f_j \cdot \frac{l_{iP,jP}}{v_{(i+j)P}}$ is also true when i or j are negative integers.

Data: An elliptic curve $E(F_p)$ and two points $P \in E(F_p)$, $Q \in E(F_{p^k})$ of order n.
Result: The Tate pairing $e(P, Q)$.
Write n in base 2: $n = (n_t n_{t-1} \dots n_1)_2$.
Choose a point $R \neq P_\infty, P, -Q, P - Q$. Initialize $f = 1$, $T = P$
for $i = n, \dots, 1$ **do**
  Compute the tangent line to T, l_T, and the line v_T that goes vertically through T. Update the point $T = 2T$.
  Update the value of the pairing $f = f^2 \cdot \frac{l_T(Q+R)}{v_T(Q+R)} \cdot \frac{v_T(R)}{l_T(R)}$.
  if $n_i = 1$ **then**
    Compute the line through T and P, $l_{T,P}$, and the line v_{T+P} that goes vertically through T.
    Update the point $T = T + P$. Update the value of the pairing $f = f \cdot \frac{l_T(Q+R)}{v_T(Q+R)} \cdot \frac{v_T(Q)}{l_T(Q)}$.
  end
end
Return $f^{(q^k-1)/n}$.

Algorithm 1: Miller's algorithm for computing the Tate pairing.

There are some important optimizations that reduce the number of operations in Algorithm 1. In [**14**] authors explained that if the embedding degree k is even, then we can remove all divisions in the algorithm. They also proved that the fact of adding a random point R to the algorithm is not necessary at all.

Using these two optimization we can simplify Miller's algorithm to Algorithm 2.

Data: An elliptic curve $E(F_p)$ and two points $P \in E(F_p)$, $Q \in E(F_{p^k})$ of order n.
Result: The Tate pairing $e(P, Q)$.
Write n in base 2: $n = (n_t n_{t-1} \dots n_1)_2$.
Initialize $f = 1$, $T = P$
for $i = n, \dots, 1$ **do**
  Compute the tangent line to T: $l_T = aX + bY + c$.
  Update the point $T = 2T$.
  Update the value of the pairing $f = f^2 \cdot (aQ_x + bQ_y + c)$.
  if $n_i = 1$ **then**
    Compute the line through T and P: $l_{T,P} = aX + bY + c$.
    Update the point $T = T + P$. Update the value of the pairing $f = f \cdot (aQ_x + bQ_y + c)$.
  end
end
Return $f^{(q^k-1)/n}$.

Algorithm 2: Miller's algorithm for computing the Tate pairing, when the embedding degree is even.

When computing a multipairing we do not need to compute each pairing independently and then multiply the obtained values; with this trivial solution, we are doing m times the final exponentiation of Miller's algorithm, which can be avoided if we do this exponentiation after the final multiplication. But this is not the only thing we can improve: all squares in the *doubling step* can be done simultaneously if f is

the temporal value of the multipairing as one can see in Algorithm 3. In this final version of Miller's algorithm the update of all m points can be done simultaneously by using Montgomery's Trick. When computing the m doubles of a point, or when adding a point, we use exactly m inversions that could have been avoided.

But there is even a better optimization, if we use Montgomery's Trick to compute simultaneously all double points $2T_i$ — there are m field inversions involved in this computation — and all m adding steps $T_i + P$ — here there are also m base field inversions.

Data: An elliptic curve $E(F_p)$ and two list of m points $\{P_i\}_{i=1}^m \subset E(F_p)$, $\{Q_i\}_{i=1}^m \subset E(F_{p^k})$ of order n.
Result: The Tate multipairing $e(\{P_i\}_{i=1}^m, \{Q_i\}_{i=1}^m)$.
Write n in base 2: $n = (n_t n_{t-1} \dots n_1)_2$.
Initialize $f = 1$, $T_i = P_i$
for $i = n, \dots, 1$ **do**
    Compute the tangent lines to T_i, l_{T_i}.
    Update the points $T_i = 2T_i$.
    Update the value of the pairing $f = f^2 \cdot \prod_{i=1}^m v_{T_i}(Q)$.
    if $n_i = 1$ **then**
        Compute the lines through T_i and P, $l_{T_i,P}$.
        Update the points $T_i = T_i + P$. Update the value of the pairing $f = f \cdot \prod_{i=1}^m l_{T_i,P}(Q)$.
    end
end
Return $f^{(q^k-1)/n}$.

Algorithm 3: Miller's algorithm for computing the Tate multipairing, when the embedding degree is even.

Chapter 2
Some uses of pairings in cryptography

As it happens in the multiplicative groups $\mathbb{Z}_p^*$, in the group of points of an elliptic curve the discrete logarithm is also assumed to be unfeasible: given two points P and $Q \in \langle P \rangle$, computing an integer $a \in \mathbb{Z}$ such that $Q = aP$ is computationally hard. This property allowed Miller and Koblitz, [**16**] [**17**], to adapt previously known cryptographic schemes to the framework of elliptic curves. Later, an upper bound of the security of these groups was discovered, using pairings, by Menezes, Okamoto and Vanstones [**18**]: a scheme defined in an elliptic curve over $\mathbb{F}_q$ and a point of n torsion does not provide more security than the subgroup of n-th roots of unity in $\mathbb{F}_{q^k}$ where k is the embedding degree of the n-torsion subgroup. Although the first of use of pairings in cryptography was therefore negative, later pairings started to offer new positive solutions for constructing new cryptographic protocols. Here we provide some examples.

2.1. Identity based encryption

Traditional public-key cryptography (for encryption) is based in the idea that each user has a pair of keys: one public that everybody can use to encrypt messages to him, and one matching private, only known by him, that is used to decrypt. This paradigm has some problems:

- The generation of keys is done before the communication. If we want to send a message, first we need to contact the receiver and ask for his public key.
- The keys are chosen randomly, and there is no a direct link between public keys and users, which can lead to impersonation attacks.

By introducing a trusted third entity, often called *certification authority* (CA) or *trusted third party* (TTP), these two problems can be solved. The idea of a pair of keys, one public and one private, for each user is maintained, but now the public keys are somewhat related to the user's identity and the matching private keys are generated by the trusted third entity. Now if a user A wants to communicate with user B, then A can use B's identity directly as the public key to encrypt the

message, without any precommunication with B. Maybe the TTP has not even generated B's private key yet. When B receives the message, he should ask for the private key to the TTP (if he has not done it before). At this point, the TTP can check if the identity of B is true or not, and provide him the private key.

Shamir introduced this notion of *identity-based cryptography* in 1984 [**19**], but the first practical scheme was discovered in 2001 by Boneh and Franklin [**20**] and pairings over elliptic curves were used in their construction. Suppose we have a pairing e over an elliptic curve and two subgroups of n-torsion $G_1 = \langle P \rangle$ and $G_2 = \langle Q \rangle$ such that $e : G_1 \times G_2 \to \mu_n$. We also need two hash functions $H_1 : \{0,1\}^* \to G_1 \setminus \{P_\infty\}$, which is used to generate a point given an identity, and $H_2 : \mu_n \to \{0,1\}^l$, which maps a point of μ_n to a sequence of bits (one-time key) in order to hide a message. The scheme is then divided in four algorithms:

TTP key generation. The TTP has a secret key $t \in \mathbb{Z}_n^*$ and publishes the matching public information $T = t \cdot Q \in G_2$.

Users' secret key generation. Given an identity ID_A of user A, the TTP generates the private key of A as $SK_A = t \cdot H_1(ID_A) \in G_1$.

Encryption. If user A wants to send a message $m \in \{0,1\}^l$ user B with identity ID_B, he first computes $Q_B = H_1(ID_B)$. Now, the encryption of a message m is

$$R = r \cdot Q \qquad\qquad c = m \oplus H_2(e(Q_B, T)^r)$$

where r is a randomly choosen integer and $\oplus$ is the bit-by-bit XOR operator.

Decryption. When a user B receives a ciphertext (R, c), he can retrieve the message by doing

$$m = c \oplus H_2(e(SK_B, R))$$

where the secret key SK_B was asked by B to the TTP beforehand.

This decryption algorithm works because

$$\begin{aligned} e(SK_B, R) &= e(t \cdot H_1(ID_B), rQ) = e(H_1(ID_B), Q)^{tr} \\ &= e(H_1(ID_B), tQ)^r = e(Q_B, T)^r \end{aligned}$$

which was the information that A used to encrypt the message.

Notice that in this scheme the private key of the user, SK_B, is used as the first argument in the pairing, each time a ciphertext is decrypted. Therefore, assuming that the user will decrypt many messages, it is interesting to consider optimization techniques suitable for this situation where the pairing is always evaluated with the same fixed first argument.

2.2. Short signatures

When speaking about cryptography one usually thinks on how we can send an important message secretly through an insecure channel, like the Internet. But cryptography can also solve problems about the authentication of the sender or the integrity of the message: when we receive a letter asking us to pay a debt, how can we be sure the sender's identity is true? Was the message modified by some

third person? That is why signature schemes are also important in cryptography. In these schemes usually one appends to the (encrypted) sent information an additional element, called *signature*, which depends on the content of the message and some (private) key only known by the sender. By using some public information of the expected sender, the receiver can be sure the message really comes from the sender and it was not modified by a third person.

Sometimes we want to make very short signatures, in order to avoid that the total length of the sent information increases dramatically. In the letter example, one has a very limited space to place the signature and, therefore, the existence of very short signatures is very important [**10**].

By using elliptic curves one can achieve shorter signatures than with other mathematical structures: as we have seen in Chapter 1, a pairing is defined over two subgroups $G_1 = \langle P \rangle$, points of n-torsion in $E(\mathbb{F}_p)$, and $G_2 = \langle Q \rangle$, which lies in the bigger curve $E(\mathbb{F}_{p^k})$. If the signature scheme is defined so that the signature is a point in G_1 and k is big enough, then we can get a shorter signature with a very big security level. One of these short signature schemes, denoted as BLS scheme, was proposed by Boneh, Lihn and Shacham [**21**]. It contains three algorithms.

Key generation. Select a random element $x \in \mathbb{Z}_n^*$. The private key of the sender is x, and the public key is $\mu = x \cdot Q \in G_2$.

Signing algorithm. Given a private key x, and some message m — the content of the letter, or the contact information of the sender — we compute the signature $\sigma = x \cdot H(m) \in G_1$, where $H : \{0,1\}^* \to G_1 \setminus \{P_\infty\}$ is a secure hash function.

Verification. Once the receiver has read the message m, he can check the authenticity of the sender and the integrity of the message by verifying if

$$e(\sigma, Q) = e(H(m), \mu).$$

In this case, if one user is verifying a lot of signatures coming from the same user (with public key μ), the second argument μ is fixed in the evaluation of the second pairing.

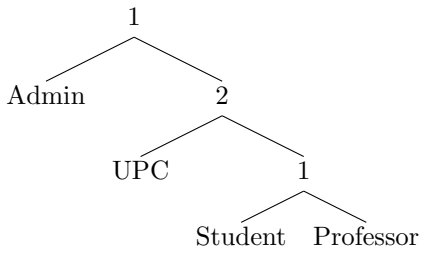
Again, it makes sense to consider optimizations in the computation of many pairings when one of the arguments (in this case, the second one) is fixed. Sometimes one can interchange the roles of the first and second arguments so that *the fixed argument* is always on the left (first argument), but in this case if we swap the roles, then the signature σ is a point of the full field, and so it is no longer a short signature.

2.3. Attribute based encryption

In identity based encryption each user has a property — his identity — which makes him totally distinguishable from other users. What happens if we substitute the idea of *identity* by some collection of properties which may be satisfied by many users at the same time? It may be useful when we want to send an encrypted message to a group, so that only users in posession of some properties are capable to decrypt. Attribute based encryption provides a solution for this problem. The scheme we are

going to explain is by Waters [**23**], adapted to the case of nonsymmetric pairings, where one user can decrypt a ciphertext if his attributes (properties) satisfy an *access policy* chosen by the sender.

Definition 2.3.1. An access policy is a boolean expression given by operators *and*, *or* and one variable for each attribute. Only users whose attributes evaluate positively this expression would be able to decrypt a ciphertext encrypted for this access policy. An access policy is usually represented as an *LSSS* matrix, in [**24**] is explained how to build this matrix given the boolean expression. Another more visual way of representing an access policy is by using trees, like in the following example



Attributes are put in the leaves of the tree, and in the interior nodes there are integers that indicate the number of subtrees the user should satisfy in order to access the data. In the example, one has acces to an *UPC intranet* either if he is an administrator or if he is a Student or Professor at UPC.

The protocols of Waters' attribute based encryption scheme work as follows.

Setup. Given a set of attributes U, for simplicity we will suppose that $U = \{1, \dots, u\}$, the TTP chooses $u = |U|$ random elements in G_1, $h_1, h_2, \dots, h_u$, and also two integers $a, \alpha \in \mathbb{Z}_n^*$.

The public parameters are

$$P, aP, G_1 = \langle P \rangle, Q, G_2 = \langle Q \rangle, e, e(P,Q)^\alpha, h_1, \dots, h_u,$$

and the master secret key of the TTP is

$$\alpha P.$$

Key generation. The key generation algorithm takes as input the master secret key and a subset S of attributes satisfied by an user. The algorithm first chooses a random $t \in \mathbb{Z}_n^*$, and creates the private key as

$$K = \alpha P + t \cdot aP \qquad L = t \cdot P \qquad \forall x \in S, K_x = t \cdot h_x$$

Encryption. This algorithm takes as input the public parameters and an access policy (M, ρ), where M is a matrix and the function ρ associates rows of M to attributes.

If M is a $l \times m$ matrix, the sender chooses a random vector $v = (s, y_2, \dots, y_m) \in \mathbb{Z}_n^m$. Now, he compute $\lambda = (\lambda_1, \dots, \lambda_l) = v \cdot M \in \mathbb{Z}_n^l$. In addition, the encryption

algorithm needs l extra random elements $r_1, \dots r_l \in \mathbb{Z}_n$. Finally, the ciphertext of a message $\tilde{m} \in \mu_n$ is

$$C = \tilde{m} \cdot e(P,Q)^{s\alpha}, \qquad C' = s \cdot P,$$

$$(C_i = \lambda_i \cdot aP - r_i \cdot h_\rho(i),\ D_i = r_i \cdot Q) \qquad \forall i = 1, \dots, l.$$

Decryption. This algorithm takes a ciphertext $(C, C', \{C_i\}_{i=1,\dots,l})$ for an acces policy (M, ρ) and a private key for a set S. Suppose that S satisfies the access policy with the attributes $I \subset S$, and let $\{\omega_i \in \mathbb{Z}_n\}_{i \in I}$ be a set of constants such that if $\{\lambda_i\}$ are valid values generated in the encryption algorithm, then $\sum_{i \in I} \omega_i \lambda_i = s$. This set of values ω_i always exists as explained in [**25**].

The decryption algorithm first computes

$$e(C', K) \cdot \prod_{i \in I} e(C_i, L)^{-\omega_i} e(D_i, K_{\rho(i)})^{-\omega_i}$$

or, equivalently,

$$e(C', K) \cdot e\left(-\sum_{i \in I} \omega_i C_i,\ L\right) \cdot \prod_{i \in I} e(D_i, -\omega_i K_{\rho(i)}),$$

and then divide this value in C to retrieve the message $\tilde{m}$.

Note that a user who decrypts many ciphertexts will have to compute a multipairing where the second arguments (the values $K_{\rho(i)}$) are fixed values of the user's secret key. Therefore, it is interesting to consider optimizations in the computation of multipairings for the particular situation where some arguments of the multipairing are fixed.

Chapter 3
Optimization of Miller's Algorithm

In Chapter 1 we mentioned some techniques that are used to optimize a single evaluation of a Tate pairing. But, as we have seen in the previous chapter, there are cryptographic applications of pairings where a lot of evaluations are done with one parameter fixed. In identity based encryption, the private key of an user is always used as a fixed parameter of a pairing evaluation when decrypting ciphertexts. In the BLS short signature scheme, the public key of the sender is used a lot of times if many signatures from the same sender are verified. Finally, in attribute based encryption the same points of the user's secret key, associated to attributes, are fixed parameters of a multipairing that is computed every time this user wants to decrypt a ciphertext. Therefore, it may make sense to spend some time preprocessing the fixed arguments of the pairings in order to obtain a faster evaluation algorithm, in the later execution of the cryptographic protocols.

The optimizations we are going to study need to have the first argument of the pairing fixed. Sometimes the roles of the first and second arguments can be interchanged, so this optimization can also be used to improve the efficiency of these schemes. When this is not possible, the use of other pairings, like the Ate pairing, may be useful.

This chapter is subdivided in three sections: the first two ones will explain how to define efficient functions $e_P : E(F_{p^k}) \to F_{p^k}$ such that given a point Q in $E(F_{p^k})$, with the same order as P, the equality $e_P(Q) = e(P, Q)$ holds. Finally, we will see that this optimization can also be applied to multipairings and to the Ate pairing.

3.1. Precomputed lines

In Miller's algorithm the two points (arguments of the pairing) have different roles:

- Multiples of the left argument are used to compute the coefficients of lines l_T and $l_{T,P}$. But the right argument is not involved in these computations.
- The right argument is evaluated on the lines. But when we have the coefficients of a line, the left argument is no longer involved.

This gives the idea of precomputing, and storing in memory, all possible lines produced by the fixed left argument P. Later, when we want to evaluate the pairing at Q, we only need to recover this information.

In Algorithm 4 we can find a sketch of the precomputing algorithm. Basically, it computes all the needed multiples of P and stores in memory the coefficients of its Miller lines. For each such line we need to store 3 coefficients in $\mathbb{F}_p$ and a boolean bit that tells us if the line comes from the doubling step, or the adding step. Later on we will need this extra bit to decide if we need to compute a square in order to update the value of the pairing.

In Algorithm 5 we explain how to compute a pairing $e(P, Q)$ given the information of precomputed lines. Notice that only the update operation is done, and no point operations are needed.

Data: An elliptic curve $E(F_p)$ and a point $P \in E(F_p)$ of order n.
Result: A list of tuples $[a_i, b_i, c_i, double]$ that represent the Miller lines $a_iX + b_iY + c_i$.
Write n in base 2: $n = (n_t n_{t-1} \dots n_1)_2$.
Initialize $T = P$
for $i = n, \dots, 1$ **do**
  Compute the tangent line to T: $l_T = aX + bY + c$.
  Update the point $T = 2T$.
  Store $[a, b, c, true]$ in the list.
  if $n_i = 1$ **then**
    Compute the line through T and P: $l_{T,P} = aX + bY + c$.
    Update the point $T = T + P$. Store $[a, b, c, false]$ in the list.
  end
end

Algorithm 4: Algorithm to precompute Miller lines.

Data: An elliptic curve $E(F_p)$, a point $Q \in E(F_{p^k})$ of order n, and the list of precomputed lines.
Result: The Tate pairing $e_P(Q) = e(P, Q)$.
Initialize $f = 1$
for *every element* $[a, b, c, double]$ *in the list* **do**
  if *double is true* **then**
    Update f, since we are in the doubling step: $f = f^2$.
  end
  Update the value of the pairing $f = f \cdot (aQ_x + bQ_y + c)$.
end
Return $f^{(q^k-1)/n}$.

Algorithm 5: Evaluation of a Tate pairing if we have a list of precomputed lines.

3.1.1. Computational cost and storage size. Two fundamental questions about this optimization are how big our storage memory needs to be and how much time the precomputation is going to take — maybe we need to compute a lot of pairings before the cost of the precomputation is amortized.

Regarding the storage, for every Miller line we need to store 3 elements in F_p and an extra bit. As there are $O(\log n)$ Miller lines, we need to store $3O(\log n)$ elements in F_p, and $O(\log n)$ extra bits. But as the base 2 representation of n usually has a very low Hamming weight, the adding step is almost always avoided and the real numbers are very close to $3\log n$ elements in F_p and $\log n$ extra bits.

As for the computational cost, we are also assuming there are so few adding steps that they have almost no effect in the computational cost.

At every step of the precomputing algorithm we need to compute:

- The double of a point in $E(F_p)$.
- The coefficients a, b, c.

As we saw in Example 1.2.5, if we are in affine coordinates this can be done in:

- Point operation: one division, 2 multiplications, 2 squares and 8 additions in F_p.
- Coefficients: 2 multiplications, one square and 4 additions in F_p.

In total, we need 1 division, 4 multiplications, 3 squares and 11 additions in F_p for every Miller line.

For the evaluating algorithm, we need to compute at each step:

- Evaluate $aQ_x + bQ_y + c$.
- A square f^2.
- Multiply $f \cdot (aQ_x + bQ_y + c)$.

The coefficients a, b, c are in the base field, and Q_x, Q_y are in the full field F_{p^k}. Therefore computing aQ_x and bQ_y requires $2k$ multiplications in F_p. The element f is also in the full field, therefore all multiplications and square operations are done there.

- $2k$ multiplications and $k+1$ additions in $\mathbb{F}_p$.
- A square in F_{p^n}.
- A multiplication in $\mathbb{F}_{p^n}$.

In Table 3.1 we give a summary of the costs and storage requirement of the two considered optimization technique, compared with those of the original Miller algorithm, and without considering the cost of computing the final exponentiation.

3.2. Merging N-lines

In [**11**] the authors noticed that computing and evaluating Miller lines was less expensive than evaluating the value of the pairing — a single full multiplication. They propose to optimize Miller's algorithm by putting some additional work in the computation of Miller lines in order to reduce the number of full multiplications needed in the evaluation algorithm. Their idea was to merge various steps of Miller's loop in only one update of the pairing value.

Algorithm	Storage	Computational cost
Original Miller	0	$(2k+4)\log n$ base multiplications and $\log n$ full multiplications. $3\log n$ base squares and $\log n$ full squares. $(k+12)\log n$ base additions. $\log n$ base divisions.
Precomputing alg.	$4\log n$ F_q $\log n$ extra bits	$4\log n$ base multiplications. $3\log n$ base squares. $11\log n$ base additions. $\log n$ divisions.
Evaluation alg.	0	$2k\log n$ base multiplication. $\log n$ full squares and mult. $k+1$ additions. 0 divisions.

TABLE 3.1. Computational costs and required storage for the Miller algorithm with the first argument fixed.

Suppose we are in step i and the bits i and $i-1$ of the group size are both 0 —we will not do the add step. The original Miller's loop does:

(1) Compute Miller line $l_T = a_iY + b_iX + c_i$.
(2) Update the pairing $f_i = f_{i+1}^2 \cdot l_T(Q)$.
(3) Now, in the next step it computes Miller line $l_{2T} = a_{i-1}Y + b_{i-1}X + c_{i-1}$.
(4) And finally, we update the pairing $f_{i-1} = f_i^2 \cdot l_{2T}(Q)$.

Then

$$\begin{aligned} f_{i-1} &= f_i^2 \cdot l_{2T}(Q) = f_{i+1}^4 l_T(Q)^2 \cdot l_{2T}(Q) = \\ &= f_{i+1}^4 \cdot (a_iQ_y + b_iQ_x + c_i)^2 \cdot (a_{i-1}Q_y + b_{i-1}Q_x + c_{i-1}) \end{aligned}$$

The idea is to first compute the bigger polynomial $l_T^2 \cdot l_{2T}$ and then evaluate it in Q. One can also use the fact that $y^2 = h(x)$ so there are no monomials with $\deg Y > 1$.

$$f_{i-1} = f_{i+1}^4 \cdot \begin{pmatrix} a_i^2a_{i-1}Q_yh(Q_x) + a_i^2b_{i-1}h(Q_x)Q_x + a_i^2c_{i-1}h(Q_x) \\ +b_i^2a_{i-1}Q_yQ_x^2 + b_i^2b_{i-1}Q_x^3 + b_i^2c_{i-1}Q_x^2 \\ +2a_ib_ia_{i-1}h(Q_x)Q_x + 2a_ib_ib_{i-1}Q_x^2Q_y + 2a_ib_ic_{i-1}Q_yQ_x \\ +2a_ic_ia_{i-1}h(Q_x) + 2a_ic_ib_{i-1}Q_xQ_y + 2a_ic_ic_{i-1}Q_y \\ +2b_ic_ia_{i-1}Q_xQ_y + 2b_ic_ib_{i-1}Q_x^2 + 2b_ic_ic_{i-1}Q_x \\ +c_i^2a_{i-1}Q_y + c_i^2b_{i-1}Q_x + c_i^2c_{i-1} \end{pmatrix}$$

In this way, one replaces one full multiplication with a lot of simpler base multiplications and additions. But we should be aware that in that evaluation powers of Q_x, and its multiples by Q_y, need to be computed. If these operations are done each time a polynomial is evaluated then a lot of additional work is being added — since Q_x lies in the full field computing Q_x^3 is as bad as doing the two updates

that are being merged. This can be avoided because the point Q is fixed in all the polynomial evaluations performed in the evaluation of the pairing: one should precompute as many powers of Q_x, and its multiples by Q_y, as needed.

In [**11**], authors noticed that this approach does not improve the algorithm cost in all the cases, but it seems that as long as the embedding degree becomes bigger, the difference between a full multiplication and base operations is so big that this complicated optimization is faster. Later in [**12**], the algorithm was adapted for the case of our study: if the first parameter of the pairing is fixed, all these complicated operations can be done only once, and used later as many times as needed.

3.2.1. General case with offline precomputation. The previous idea can be generalized to merge N Miller steps in only one: suppose we have a temporal value of the pairing, f_1, and next we are going to compute N consecutive doubling steps of Miller algorithm. At the end, we have:

- N Miller lines, $l_{2^{i-1}T}$ for $i = 1, \ldots, N$.
- N updates of the pairing value

$$f_{i+1} = f_i^2 \cdot l_{2^{i-1}T}(Q) \quad i = 1, \ldots, N,$$

and then

$$f_{N+1} = f_1^{2^N} \cdot \prod_{i=1}^{N} (l_{2^{i-1}T}(Q))^{2^{N-i}}.$$

The idea of the algorithm is now to compute first the merged polynomial

$$G_N(X, Y) = \prod_{i=1}^{N} l_{2^{i-1}T}(X, Y)^{2^{N-i}}$$

to finally update the value of the pairing by

$$f_{N+1} = f_i^{2^N} \cdot G_N(Q).$$

This reduces $N - 1$ full multiplications, but we need a lot of work to compute, and evaluate in Q, the polynomial G_N.

3.2.1.1. *Computing the merged polynomial.* As we pointed out in the case $N = 2$, if the points in our elliptic curve satisfy $Y^2 = h(X)$ then we can use this information to reduce the degree of Y in the merged polynomial G_N. We will denote by f_N the monomials of G_N that are not a multiple of Y, and by $Y \cdot g_N$ the monomials of G_N that are a multiple of Y. Then we can express

$$G_N(X, Y) = f_N(X) + Y \cdot g_N(X).$$

LEMMA 3.2.1. *Suppose that N consecutive Miller loops are doubling steps and let $G_N(X, Y) = f_N(X) + Y g_N(X)$ be the N-merged polynomial. Then* $\deg g_N = \deg f_N - 1$ *and* $\deg f_N = 3(2^{N-1} - 1) + 1$.

PROOF. The proof is by induction on N.

If $N = 1$ the polynomial is $G_N = l_T = aY + bX + c$. Therefore $\deg f_1 = 1$ and $\deg g_1 = 0$.

This way of representing G_N allows us to write a recursion formula to compute G_N: if we know $G_{N-1} = f_{N-1}(X) + Y \cdot g_{N-1}(X)$ and $l_{2^{N-1}T} = aY + bX + c$ then

$$\begin{aligned} G_N &= G_{N-1}^2 \cdot l_{2^{N-2}T} \\ &= (f_{N-1} + Y g_{N-1})^2 \cdot l_{2^{N-2}T} \\ &= (f_{N-1}^2 + Y^2 g_{N-1}^2 + 2 f_{N-1} g_{N-1} Y) \cdot (aY + bX + c) \\ &= c f_{N-1}^2 + b f_{N-1}^2 X + Y(a f_{N-1}^2 + 2b f_{N-1} g_{N-1} X + 2c f_{N-1} g_{N-1}) \\ &\quad + Y^2(2a f_{N-1} g_{N-1} + g_{N-1}^2 c + g_{N-1}^2 bX) + Y^3 a g_{N-1}^2. \end{aligned}$$

Then, since $Y^2 = h(X)$ we obtain $G_N = f_N + Y g_N$ where

$$\begin{aligned} f_N &= c f_{N-1}^2 + b f_{N-1}^2 X + h(X)(2a f_{N-1} g_{N-1} + g_{N-1}^2 c + g_{N-1}^2 bX) \\ g_N &= a f_{N-1}^2 + 2b f_{N-1} g_{N-1} X + 2c f_{N-1} g_{N-1} + a g_{N-1}^2 h(X). \end{aligned}$$

Then by the induction hypothesis, we know that $\deg f_{N-1} = \deg g_{N-1} + 1$. Therefore

$$\begin{aligned} \deg f_N &= \deg f_{N-1} + \deg g_{N-1} + \deg h \\ &= 2 \deg f_{N-1} + \deg h - 1 = 2 \deg f_{N-1} + 2 \\ \deg g_N &= 2 \deg g_{N-1} + \deg h \\ &= 2 \deg f_{N-1} + \deg h - 2 = 2 \deg f_{N-1} + 1 \end{aligned}$$

If $\deg f_{N-1} = 3(2^{N-2} - 1) + 1$ then

$$\begin{aligned} \deg f_N &= 2 \deg f_{N-1} + 2 = 2(3 \cdot 2^{N-2} - 3 + 1) + 2 \\ &= 3 \cdot 2^{N-1} - 2 = 3(2^{N-1} - 1) + 1 \end{aligned}$$

□

The previous lemma gives us the exact degree of the merged polynomial G_N when all Miller steps did not need adding a point, in the proof we also have seen a constructive way to compute it. Now it remains to see what happens when some adding step is involved. Neither in [**11**] nor in [**12**] it is explained what is the best approach in this case. But there are clearly two possible solutions:

- We can count the doubling and adding steps as only one bigger step in the algorithm, i.e. we always update the merged polynomial with the adding steps.
- Instead of fixing the number of steps to merge, we can build the merged polynomial until we reach a maximum fixed degree.

In both cases we need an upper bound on the degree and some constructive way to build the merged polynomial G_N. These two things are given by the following lemma.

LEMMA 3.2.2. *Let $G_N(X,Y) = f_N(X) + Y g_N(X)$ be the N-merged polynomial. Then $\deg f_N$ and $\deg g_n$ are bounded by $6 \cdot 2^{N-1} - 3$ and $\deg g_n \leq \deg f_n - 2$.*

PROOF. To prove this lemma (again, by induction on N) we assume all Miller steps have doubling and adding steps, then the recursion for G_N is given by:

$$G_N = G_{N-1}^2 \cdot l_T \cdot l_{T,P}.$$

First, if $N = 1$ then

$$\begin{aligned} G_1 &= l_T \cdot l_{T,P} = (aY + bX + c) \cdot (iY + jX + k) \\ &= ck + (bk + cj)X + aih(X) + Y(ak + ci + X(aj + bi)) \end{aligned}$$

Then $\deg g_1 = 1 < \deg f_1 = 3 \leq 7 \cdot 2^0 - 4$.

Now, for N arbitrary we know that

$$G_N = G_{N-1}^2 \cdot l_T \cdot l_{T,P},$$

due to the case $N = 1$ we know that $l_T \cdot l_{T,P}$ can be represented as $l_x + Y l_y$, where $\deg l_x = 3$ and $\deg l_y = 1$; therefore

$$\begin{aligned} G_N = f_{N-1}^2 l_x + g_{N-1}^2 h l_x + 2 f_{N-1} g_{N-1} l_y h \\ + Y(f_{N-1}^2 l_y + g_{N-1}^2 h l_y + 2 f_{N-1} g_{N-1} l_x). \end{aligned}$$

And

$$\begin{aligned} \deg f_N &\leq \max\{2 \deg f_{N-1} + 3, 2 \deg g_{N-1} + 6, \deg f_N + \deg g_N + 4\} \\ \deg g_N &\leq \max\{2 \deg f_{N-1} + 1, 2 \deg g_{N-1} + 4, \deg f_N + \deg g_N + 3\}. \end{aligned}$$

Knowing that $\deg g_i \leq \deg f_i - 2$ we get

$$\begin{aligned} \deg f_N &\leq 2 \deg f_{N-1} + 3 \\ \deg g_N &\leq 2 \deg f_{N-1} + 1. \end{aligned}$$

Finally, if $\deg f_{N-1} \leq 6 \cdot 2^{N-2} - 3$, then

$$\deg f_N \leq 2(6 \cdot 2^{N-2} - 3) + 3 = 6 \cdot 2^{N-1} - 3$$

□

This bound is only tight when there are N consecutive bits different from 0, and we usually choose the point in the elliptic curve with low Hamming weight so that this happens almost never. In Table 3.2 we give the maximum degree found in our test cases compared to this two bounds.

N	Lower bound	Upper bound	In tests
1	1	3	3
2	4	9	6
3	10	21	12
4	22	45	24

TABLE 3.2. Maximum degree of the merged polynomial G_N and degree found in our tests.

3.2.1.2. *Computational cost of the algorithm.* In this optimization there are two main sub-algorithms whose computational cost must be studied: how hard is to evaluate a merged polynomial and how hard is to compute a merged polynomial?

The first question has the easiest answer: we have just seen that the polynomial G_N can be decomposed in two polynomials of degree at most $6 \cdot 2^{N-1} - 3$. Then, if we suppose that all powers of Q_x are precomputed we need exactly $6k \cdot 2^{N-1} - 3k$ base multiplications — to compute all $a_i \cdot Q_x^i$ and then $6 \cdot 2^{N-1} - 2$ full additions — or $6k \cdot 2^{N-1} - 2k$ base additions. Since we need to do this twice — for the Y part — we need at most $12k \cdot 2^{N-1} - 6k$ base multiplications and $12k \cdot 2^{N-1} - 4k + 1$ additions.

We need to precompute all powers of Q_x because, otherwise, when we will evaluate the merged polynomial we will have to do as many full multiplications as we wanted to avoid with this optimization.

Therefore, the cost of a single evaluation of this optimized Miller's algorithm is the cost of precomputing all the needed powers of Q_x and its multiples by Q_y — about $12 \cdot 2^{N-1} - 6$ full multiplications — and the cost of evaluating $\log n/N$ merged polynomials. The big number of possible powers of Q_x that we need to compute is not as bad as evaluating $\log n/N$ polynomials, so we do not really focus on this computation — even though a good way of computing these powers will benefit the whole algorithm.

Regarding the computational cost of computing the merged polynomials, we first analyze how many multiplications and additions are needed to compute an update of the merged polynomial; suppose we have G_i and we want to compute G_{i+1} then

$$\begin{aligned} G_{i+1} = f_i^2 l_x + g_i^2 h l_x + 2 f_i g_i l_y h \\ + Y(f_i^2 l_y + g_i^2 h l_y + 2 f_i g_i l_x). \end{aligned}$$

where l_x is a polynomial of degree 3 and l_y is a polynomial of degree 1. And we also know that $\deg f_i \leq 6 \cdot 2^{i-1} - 3$ and $\deg g_i \leq 6 \cdot 2^{i-1} - 5$. Let f and g be the number of coefficients of f_i and g_i.

Assuming we need about f^2, g^2 and fg multiplications and additions to compute f_i^2, g_i^2 and $f_i \cdot g_i$, then we need about

$$6f^2 + 4g^2 + 9fg$$

multiplications and additions to compute G_{i+1}. But since $g = f - 2$, we need about

$$19f^2 - 34f + 16$$

multiplications and additions. Since $f = 6 \cdot 2^{i-1} - 2$ and we are doing this for $i = 1, \ldots, N-1$, we need in total about

$$\sum_{i=1}^{N-1} 19 \cdot (6 \cdot 2^{i-1} - 3) - 34 \cdot (6 \cdot 2^{i-1} - 3) + 16 = 684 \frac{4^{N-1} - 1}{3} - 660 \cdot (2^{N-1} - 1) + 160(N-1)$$

multiplications and additions.

But this is not a realistic upper bound of the number of multiplications: not all the bits of n are 1, and therefore not always we will need to add so much information

	Worst case	Best case
Initialize Q_x	$2 \cdot (6 \cdot 2^{N-1} - 3)\ m_1$	$2 \cdot (3 \cdot 2^{N-1} - 2)\ m_1$
Miller loop	$\frac{\log n}{N}(12k \cdot 2^{N-1} - 6k)\ m_1$ $\frac{\log n}{N}(12k \cdot 2^{N-1} - 4k + 1)\ a_1$	$\frac{\log n}{N}(6k \cdot (2^{N-1} - 4k))\ m_1$ $\frac{\log n}{N}(6k \cdot 2^{N-1} - 2k + 1)\ a_1$

TABLE 3.3. Summary of how many multiplications, m_1, and additions, a_1, are needed to compute a single pairing when we merge N steps. The first row gives the cost of precomputing all possible powers of Q_x. The second row gives the cost of evaluating one polynomial, multiplied with the number of steps of the algorithm.

	Worst case
Merge polynomial	$\frac{\log n}{N}\left(684\frac{4^{N-1}-1}{3} - 660 \cdot (2^{N-1} - 1) + 160(N-1)\right)$
Updating points	$\frac{\log n}{N}$ point doubling and $\frac{\log n}{N}$ point addition.
	Best case
Merge polynomial	$135\frac{4^{N-1}-1}{3} - 135 \cdot (2^{N-1} - 1) + 34$
Updating points	$\frac{\log n}{N}$ point doubling.

TABLE 3.4. Summary of how many multiplications and additions are needed to compute all the merged polynomials.

to the polynomial. On the other hand, if we assume that no adding steps are performed then the approximated number of operations is

$$4f^2 + 4g^2 + 7fg$$

and doing the same computations with $f = 3 \cdot 2^{i-1} - 1$ and $g = f - 1$ then we need about

$$135\frac{4^{N-1} - 1}{3} - 135 \cdot (2^{N-1} - 1) + 34$$

multiplications and additions.

In Tables 3.3 and 3.4 we summarize the computational cost of the two algorithms involved in this optimization. One can see that it may be feasible for small values of N, but when N grows, the extra work we are adding in the evaluating algorithm is much bigger than the decrease in the number of Miller steps. Therefore we only expect some real optimization for very small values of N.

3.2.2. Related optimizations. There is a natural adaptation of the merging optimization to compute a multipairing $\prod_{i=1}^{m} e(P_i, Q_i)$: since all the first arguments have the same order n, the resulting merged polynomials of each point are merging the same steps of the original Miller's loop. Therefore, if $G_{N,P_i}(x, y)$ are the N-merged polynomials of points $P_1, \ldots, P_m$, then the merged polynomial of the multipairing would be

$$G_N(Q_1, \ldots, Q_m) = \prod_{i=1}^{m} G_{N,P_i}(Q_i).$$

But contrary to the merging optimization, we do not want to compute first the big polynomial G_N and then evaluate at $Q_1, \ldots, Q_n$. In this case the resulting polynomial can be very big — since m can be quite big in some cryptographic applications like attribute based encryption. Therefore, we are going to evaluate each merged polynomial and finally multiply all the resulting values. In this way, we are reducing the operations needed to compute N original Miller steps. And since we are updating the multipairing in the same temporal variable, we are also reducing the number of squares we need to compute it.

Even though we did not talk about the Ate pairing in the mathematical preliminaries, the optimization we have presented for the Tate pairing also holds for the Ate pairing. The definition of the Ate pairing is similar to that of the Tate pairing, but now the point P lies in the bigger elliptic curve $E(\mathbb{F}_{p^k})$ and the point Q in the base elliptic curve $E(\mathbb{F}_p)$. With this change we are allowed to reduce the number of Miller steps we need to compute, but all point operations are done in the bigger elliptic curve. The merged optimization works exactly in the same way, but:

- When computing the merged polynomial, operations are done in the full field instead of the base field. This computation will take much more time.
- When evaluating a polynomial, the precomputation of powers of $Q_x \in \mathbb{F}_p$ is no longer needed since
$$aQ_x^2 + bQ_x + c$$
needs as many multiplications and additions as
$$c + Q_x(b + aQ_x).$$

Chapter 4
Numerical results

Along with the theoretical explanation of the merging algorithm, we wanted to do an implementation of it, in order to test whether the results are significant, and to know how costly are the precomputations costs.

We decided to do this implementation and numerical tests with the PBC (Pairing-Based Cryptography) Library developed by Ben Lynn [**27**]. This library is built over an arbitrary precision library, GMP Library, which is used to perform computations in fields $\mathbb{F}_p$. All constructions for tower fields, elliptic curves, pairings and some basic optimizations were done by Ben Lynn.

This is not the only known library for this kind of tests. For example, in [**15**] the library called MIRACL [**29**] is used. This library is very well documented regarding arithmetics over finite fields, was thought to work in different architectures and mobile devices, and more constructions of elliptic curves are available there than in the PBC Library. But elliptic curve algorithms are not so well documented and we had some problems to understand how the constructions were done and where the data was stored.

4.1. Preliminary tests

We first tested the cost of operations over finite fields. A summary was found in Tables 1.1 and 1.2. Computations are done for a pack of 1000 operations because these operations are done very quickly and sometimes (when the size of the field is small) one cannot appreciate any difference in μs if only one operation is considered.

Two fundamental comments about these results are that divisions can be very expensive. So the optimization that allowed us to remove divisions is a very significant optimization in the cost of computing pairings. Another thing we noticed is that tower fields $\mathbb{F}_p \subset \mathbb{F}_{p^{10}}$ may have bad properties for the implementation: the cost of computing divisions in a field of dimension about $10 \cdot 2046 = 20460$ (1.2 seconds to compute 1000 inversions) is comparable to the cost in a field of dimension $6 \cdot 5295 = 31770$ (1.4 seconds) or even worse than in a field of dimension $12 \cdot 2456 = 29476$ (0.8 seconds), even if the first full field has a better base field.

bits of p	cost mult.	cost invr.	Point Addition	Point Multiplication
32	0.090	0.370	1	1
58	0.088	0.553	1	1
124	0.060	1.489	3	2
196	0.082	2.480	4	4
437	0.179	5.271	7	6
677	0.386	8.489	12	10
1357	1.357	18.610	26	23
5295	17.175	118.335	224	168

TABLE 4.1. Comparison of the cost of Point Addition and Multiplication in Type D elliptic curves with the cost of base multiplications and inversions. Times are in μs and measure one operation.

So we do not expect to be able to obtain spectacular results with elliptic curves of embedding degree 10.

4.1.1. Families of elliptic curves. The PBC Library has some families of elliptic curves already implemented. They are called Type A, B, C, D, E, F and G. We were only interested in Type D, F and G since the other ones either have embedding degree 1 or do not have an implementation provided. In Tables 4.1, 4.2 and 4.3 we can find the costs of computing a single point operation in $E(\mathbb{F}_p)$ for these curves; we did not run this test in the bigger elliptic curve because no point operations are done there.

4.1.1.1. *Type D elliptic curves.* These are ordinary curves over $\mathbb{F}_p$ of the form $y^2 = x^3 + ax + b$ with embedding degree 6. These curves have order $h \cdot r$ where r is a prime, and h is a small constant (so that it is easy to check whether a point P has large order).

The tower field is constructed as follows

$$\mathbb{F}_p \subseteq \mathbb{F}_{p^3} \subseteq \mathbb{F}_{p^6}$$

like we defined in Example 1.1.6.

The documentation in the library recommended a curve of order about 170 bits, with a similar security as a typical ElGamal multiplicative group $\mathbb{Z}_p^*$ with 1024 bits.

4.1.1.2. *Type G elliptic curves.* These are ordinary curves over $\mathbb{F}_p$ of embedding degree 10. No more information is provided in the general documentation, but a quick look in the implementation shows that they are very similar to Type D elliptic curves, and the tower field is constructed as follows:

$$\mathbb{F}_p \subseteq \mathbb{F}_{p^5} \subseteq \mathbb{F}_{p^{10}}.$$

We expect slightly better results than in Type D elliptic curves, since the embedding degree is bigger and they all have a similar underlying structure. But on the other hand, since operations in $\mathbb{F}_{p^5}$ are more expensive than in $\mathbb{F}_{p^3}$ we may find some problems.

bits of p	cost mult.	cost invr.	Point Addition	Point Multiplication
102	0.039	1.275	2	2
127	0.052	1.612	3	2
257	0.130	3.267	5	4
418	0.181	4.872	7	6
2456	3.818	38.797	59	51

TABLE 4.2. Comparison of the cost of Point Addition and Multiplication in Type G elliptic curves with the cost of base multiplications and inversions. Times are in μs and measure one operation.

bits of p	cost mult.	cost invr.	Point Addition	Point Multiplication
105	0.033	0.876	2	2
124	0.032	1.063	2	2
250	0.070	1.422	5	4
348	0.128	2.089	7	6
2046	2.594	22.499	44	38

TABLE 4.3. Comparison of the cost of Point Addition and Multiplication in Type F elliptic curves with the cost of base multiplications and inversions. Times are in μs and measure one operation.

4.1.1.3. *Type F elliptic curves.* The last family of curves over $\mathbb{F}_p$ has the biggest embedding degree, 12. The documentation recommends this kind of elliptic curves to implement signature schemes: as we explained in Chapter 3 a higher embedding degree allows us to construct shorter signatures with more security.

In this case, the curve has the form $y^2 = x^3 + b$, so we expect faster point operations (and less optimization in this aspect).

The tower field in Type F curves are

$$\mathbb{F}_p \subset \mathbb{F}_{p^2} \subset \mathbb{F}_{p^{12}}.$$

The last jump in the tower could be improved by adding an intermediate step

$$\mathbb{F}_p \subset \mathbb{F}_{p^2} \subset \mathbb{F}_{p^6} \subset \mathbb{F}_{p^{12}}$$

in order to use optimizations like in Example 1.1.6. This was not used because with the tower $\mathbb{F}_p \subset \mathbb{F}_{p^2} \subset \mathbb{F}_{p^{12}}$ one can construct G_2 such that all points have lots of zeros in their representation. And therefore, a better multiplication algorithm could be possible. So in this type of elliptic curves we may not find a very significant optimization: the structure of G_2 is so good that a well implemented multiplication could be simply better than this general optimization.

4.2. Pairing tests

For testing our optimization we are going to run two different tests: for small values of N we want to see which is the gain of this algorithm, and how many pairings we

bits	original	precomp	time	%	$\%_{pre}$	#	$\#_{pre}$
179	3.423	1.441	2.488	72	97	2	24
252	6.219	2.613	4.260	68	94	2	11
522	31.982	10.516	21.677	68	88	1	4
1021	141.330	39.697	94.441	67	89	1	4
5295	14105.709	2663.285	10443.196	74	85	1	2

TABLE 4.4. Comparison of the time spent in computing a single pairing in type D curves. Optimizations are done with the merging technique with $N = 1$.

bits	original	precomp	time	%	$\%_{pre}$	#	$\#_{pre}$
179	3.423	2.287	2.282	66	89	2	5
252	6.219	4.022	3.913	63	87	2	4
522	31.982	15.204	19.984	63	82	2	3
1021	141.330	57.642	87.293	62	82	1	3
5295	14105.709	4368.717	9806.592	69	80	1	2

TABLE 4.5. Comparison of the time spent in computing a single pairing in type D curves. Optimizations are done with the merging technique with $N = 2$.

bits	original	precomp	time	%	$\%_{pre}$	#	$\#_{pre}$
179	3.423	3.411	2.289	67	90	3	13
252	6.219	6.287	3.876	62	86	3	10
522	31.982	26.348	19.375	60	79	2	5
1021	141.330	102.337	85.276	60	80	2	5
5295	14105.709	9226.473	9534.591	67	78	2	4

TABLE 4.6. Comparison of the time spent in computing a single pairing in type D curves. Optimizations are done with the merging technique with $N = 3$.

need to compute in order to amortize the time spent in the precomputation phase. Finally, we will run the multipairing algorithm for $m = 2, 10$ and 30.

The idea behind the first test is to see the cost of computing a basic Identity Based Encryption and a Short Signature scheme like BLS. The second test is to see how the decryption time of an attribute based encryption scheme is improved (assuming 10 or 30 attributes are involved in the decryption policy), and the same for an identity based encryption scheme [**22**], whose decryption phase evaluates a multipairing with $m = 2$.

In Tables 4.4, 4.5, 4.6 and 4.7 we give the times spent in order to compute a single pairing in type D elliptic curves. We can see how much time is used to compute the pairing with the original Miller algorithm, how much time is spent in the precomputation phase of the considered optimization, and how much time is spent in a single evaluation with our optimization (using the precomputed values).

bits	original	precomp	time	%	$\%_{pre}$	#	$\#_{pre}$
179	3.423	5.669	2.502	73	98	7	123
252	6.219	9.911	4.040	65	90	5	22
522	31.982	43.641	20.271	63	83	4	11
1021	141.330	180.354	89.534	63	84	4	11
5295	14105.709	18059.789	9947.841	70	82	5	9

TABLE 4.7. Comparison of the time spent in computing a single pairing in type D curves. Optimizations are done with the merging technique with $N = 4$.

bits	original	$N = 1$	#	$N = 2$	#	$N = 3$	#	$N = 4$	#
179	3.423	2.488	2	**2.282**	2	2.289	3	2.502	7
252	6.219	4.260	2	3.913	2	**3.876**	3	4.040	5
522	31.982	21.677	1	19.984	2	**19.375**	2	20.271	4
1021	141.330	94.441	1	87.293	1	**85.276**	2	89.534	4
5295	14105.709	10443.196	1	9806.592	1	**9534.591**	2	9947.841	5

TABLE 4.8. Summary of time spent in computing a single pairing in type D curves. In bold the best time for each curve.

bits	original	precomp	time	%	$\%_{pre}$	#	$\#_{pre}$
86	3.611	0.359	3.228	89	97	1	4
127	6.869	0.920	5.794	85	95	1	4
231	18.002	2.386	14.981	83	92	1	2
418	60.434	6.223	51.004	84	94	1	2
2456	6061.466	327.435	5145.309	85	92	1	1

TABLE 4.9. Comparison of the time spent in computing a single pairing in type G curves. Optimizations are done with the merging technique with $N = 1$.

bits	original	precomp	time	%	$\%_{pre}$	#	$\#_{pre}$
86	3.611	0.633	3.076	85	92	2	2
127	6.869	1.481	5.299	77	87	1	2
231	18.002	3.450	13.973	78	87	1	2
418	60.434	9.344	46.886	76	86	1	2
2456	6061.466	511.550	4761.308	79	85	1	2

TABLE 4.10. Comparison of the time spent in computing a single pairing in type G curves. Optimizations are done with the merging technique with $N = 2$.

To compare these times we include in the column % the percentage of time spent in an evaluation in the optimization when compared with the original algorithm; we also include in column $\%_{pre}$ the percentage of time of the considered optimization

bits	original	precomp	time	%	$\%_{pre}$	#	$\#_{pre}$
86	3.611	1.006	3.156	87	94	3	6
127	6.869	2.347	5.336	78	88	2	4
231	18.002	5.791	13.411	75	83	2	2
418	60.434	15.792	45.293	75	83	1	2
2456	6061.466	1013.878	4581.405	75	82	1	1

TABLE 4.11. Comparison of the time spent in computing a single pairing in type G curves. Optimizations are done with the merging technique with $N = 3$.

bits	original	precomp	time	%	$\%_{pre}$	#	$\#_{pre}$
86	3.611	1.590	3.336	92	100	6	–
127	6.869	3.676	5.391	78	88	3	5
231	18.002	8.792	13.724	76	85	2	4
418	60.434	25.279	45.467	75	83	2	3
2456	6061.466	1877.872	4577.809	75	82	2	2

TABLE 4.12. Comparison of the time spent in computing a single pairing in type G curves. Optimizations are done with the merging technique with $N = 4$.

bits	original	$N = 1$	#	$N = 2$	#	$N = 3$	#	$N = 4$	#
86	3.611	3.228	1	**3.076**	2	3.156	3	3.336	6
127	6.869	5.794	1	**5.299**	1	5.336	2	5.391	3
231	18.002	14.981	1	13.973	1	**13.411**	2	13.724	2
418	60.434	51.004	1	46.886	1	**45.293**	1	45.467	2
2456	6061.466	5145.309	1	4761.308	1	4581.405	1	**4577.809**	2

TABLE 4.13. Summary of time spent in computing a single pairing in type G curves. In bold the best time for each curve.

bits	original	$N = 1$	#	$N = 2$	#	$N = 3$	#	$N = 4$	#
71	**7.058**	7.553		7.483		7.718		8.443	
105	**10.432**	10.784		10.559		10.780		11.620	
192	22.320	21.903	7	21.479	5	**21.286**	5	22.105	587
348	54.820	53.803	4	53.256	5	**52.110**	2	53.214	17
2046	3113.022	2924.441	1	2962.689	2	2813.475	2	**2793.080**	5

TABLE 4.14. Summary of time spent in computing a single pairing in type F curves. In bold the best time for each curve.

when compared with the basic optimization of precomputed lines. Finally, we include in column # (resp. column $\#_{pre}$) how many pairing evaluations must be done before the precomputation time is amortized in our optimization, with respect to the basic Miller algorithm (resp. with respect to the basic optimization

		179	#	252	#	522	#	1021	#
$m = 2$	original	4.985		9.013		47.727		205.019	
	$N = 2$	3.346	3	5.675	3	28.814	2	122.714	2
	$N = 3$	3.338	4	5.398	4	27.778	3	128.993	3
$m = 10$	original	17.254		31.161		170.247		774.527	
	$N = 2$	11.897	5	20.385	4	99.434	3	408.606	2
	$N = 3$	11.183	6	19.214	6	91.388	4	387.0011	3
$m = 30$	original	48.181		85.527		480.572		2149.255	
	$N = 2$	33.280	5	56.311	4	270.303	3	1122.135	2
	$N = 3$	31.043	7	53.391	6	252.811	4	1054.756	3

TABLE 4.15. Summary of execution times of a multipairing in type D elliptic curves, without counting precomputation. Column # gives the number of executions needed to amortize precomputation.

of precomputed lines). In Table 4.8 we give a summary of the important results: time spent to compute a single evaluation (discarding the precomputation time) and the evaluations needed to amortize the precomputation effort.

We obtained the same results as in [**12**]: the best result is obtained in the cases $N = 2$ or $N = 3$. But we also have seen that not so many pairings are needed to amortize the expensive precomputation algorithm: with only 3 evaluations we amortize the off-line precomputation phase.

We find similar results in type G elliptic curves, as one can see in Table 4.13. But in this case we also obtained some elliptic curves where the optimization with $N = 4$ was also useful. This was expected since a larger embedding degree produces a bigger difference of computations in $\mathbb{F}_p$ and $\mathbb{F}_{p^k}$. In Tables 4.9, 4.10, 4.11 and 4.12 we give a more detailed information about the execution times in type G curves.

The most surprising results were obtained for type F elliptic curves. As one can see in Table 4.14 the original Miller algorithm is the best solution for almost every curve in our test. But we noticed that the (few) elliptic curves that led to good results for our optimization are the ones with large order. There are three main reasons why this happens. (1) Point operations do not seem expensive compared to the whole pairing. (2) The gain in every Miller loop is not that high, since points in G_2 have a lot of zeros and a good algorithm for *full* multiplication is used; but the merged optimization needs full multiplications. (3) Finally, if the order of the group G_1 is not so high, the time we save in the Miller loops may be less than the time we need to spend in the online precomputation of powers of Q_x.

Obviously, in the multipairing algorithm we obtained similar results: in both optimizations we reduce the same number of full multiplications, and we merge the same number of Miller loops. In Tables 4.15, 4.16, 4.17 we give a summary of the times we obtained. For type D curves we did not put values for $p \approx 5000$ bits because the test failed (not enough memory), and for type F curves we only include here the tests that gave some positive results for the considered optimization.

		86	#	127	#	231	#	418	#
$m = 2$	original	4.360		8.552		23.861		77.697	
	$N = 2$	3.921	3	7.052	2	18.001	2	58.470	1
	$N = 3$	3.891	5	6.677	3	16.795	2	55.191	2
$m = 10$	original	11.721		25.542		72.889		274.170	
	$N = 2$	9.653	3	19.310	3	48.883	1	155.526	1
	$N = 3$	9.765	5	17.848	3	43.312	2	137.570	2
$m = 30$	original	29.287		64.461		197.507		630.962	
	$N = 2$	24.372	4	50.356	3	125.601	2	391.914	1
	$N = 3$	24.888	7	45.636	4	109.329	2	342.119	2

TABLE 4.16. Summary of execution times of a multipairing in type G elliptic curves, without counting precomputation. Column # gives the number of executions needed to amortize precomputation.

		192	#	348	#	2046	#
$m = 2$	original	25.398		67.283		3937.400	
	$N = 2$	24.788	10	61.142	3	3452.705	2
	$N = 3$	24.641	12	59.224	3	3248.099	2
$m = 10$	original	67.544		171.655		10973.758	
	$N = 2$	50.458	2	128.426	2	7611.773	1
	$N = 3$	49.659	7	117.765	3	6481.589	2
$m = 30$	original	142.747		436.899		25153.521	
	$N = 2$	115.563	4	294.455	2	17845.022	2
	$N = 3$	113.255	6	263.022	3	14294.065	2

TABLE 4.17. Summary of execution times of a multipairing in type F elliptic curves, without counting precomputation. Column # gives the number of executions needed to amortize precomputation.

Chapter 5
Conclusion and future work

A lot of work has been done in order to optimize Miller's algorithm [**14**] [**13**] [**2**] [**11**] [**12**], and we decided to analyze in more detail the last optimization given by Costello *et al.* in [**12**]. This optimization is very useful when an argument of the pairing is fixed in various evaluations, and Scott mentioned [**15**] that this is a common situation in some public-key cryptographic protocols, like the schemes we reviewed in Chapter 2.

We have added two main contributions to Costello *et al.* optimization. First we have analyzed the computational cost of the algorithm in the worst case, instead of the *artificial* best case scenario that was considered in [**11**] and [**12**]. We have also analyzed theoretically the complexity of the off-line precomputations stage.

Second, we have implemented this optimization in an already existing cryptographic library, PBC Library, in order to check numerically the results given by [**12**] and the tightness of our complexity upper bound. Positive results have been found, since we have obtained better times than the classical algorithm with precomputed lines. We have also observed that very few evaluations, at most 6 but 2 or 3 if the parameter N is well chosen, are enough to amortize the computational cost of the precomputation phase. Therefore, the conclusion is that Costello *et al.* optimization can be very useful in some modern public-key cryptographic protocols. We have got some bad results in elliptic curves with embedding degree 10, but we have been able to detect the reason for this: a better way of updating the pairing value is used in those elliptic curves, but our optimized algorithm always do a *full* multiplication, which is worse. This gives less optimization in every Miller step, and the overall gain is overcome by the cost of the online precomputations in almost all the tested elliptic curves. One should not overlook particular optimizations for a specific family of elliptic curves, since this could provide an even better optimization.

We think that there is still some work to be done in the study of Costello *et al.* optimization.

- Since Ate pairings are also defined over two point sets with different sizes, the same full multiplications can be avoided by using the same idea. In fact, since Ate pairing has shorter Miller loops, the computational cost of a single evaluation would decrease. But, on the other hand, the off-line computations would

be a lot heavier since now the merged polynomial has coefficients in the full field. Is it also true that a few evaluations would amortize precomputations? Or would it be useless since a lot of evaluations must be done in a short time?

- When implementing the Costello *et al.* optimization for multipairings, all the first arguments were precomputed with the same value of N. It seems possible to adapt the algorithm in order to accept a few (not fixed) arguments without precomputations. This is something needed in some attribute based encryption schemes like the one described in Chapter 2.
- Modern computers and mobile devices have processors which are able to execute multiple threads at the same time. This concurrent execution capability could bring some improvements in the studied optimization: one thread could do an online computation of the merged polynomial and then pass it to the other thread, which would evaluate the polynomial and updates the value of the pairing. This optimization would not give us a faster algorithm, but we would reduce the memory needed to store all the merged polynomials.

References

[1] Neal Koblitz, Alfred J. Menezes. *A Survey of Public-Key Cryptosystems*. Society for Industrial and Applied Mathematics. SIAM Rev, Vol. 46, Number 4. (April 2004)

[2] Daniel M. Gordon. *A survey of fast exponentiation methods*. Journal of Algorithms, Vol. 27, p.p. 129-146 (1998).

[3] J. L. Brown, Jr., *On Lamé's Theorem*, Fibonacci Quarterly, v5, n2 (April 1967), 153-160.

[4] Laurie Genelle1, Emmanuel Prouff, and Michäel Quisquater. *Montgomery's Trick and Fast Implementation of Masked AES*. AFRICACRYPT 2011, LNCS 6737, pp. 153-169, 2011.

[5] Skander Belhaj and Haïthem Ben Kahla. *On the complexity of computing the GCD of two polynomials via Hankel matrices*. ACM Communications in Computer Algebra, Vol 46, No. 3, Issue 181, September 2012

[6] André Weimerskirch and Christof Paar. *Generalizations of the Karatsuba Algorithm for Efficient Implementations*. (2006)

[7] Robert Bix. *Conics and Cubics: A Concrete Introduction to Algebraic Curves*. Undergraduate Texts in Mathematics. Springer, 2006.

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

[9] Lawrence C. Washington, *Elliptic Curves. Number Theory and Cryptography*. Chapman & Hall/CRC, Second Edition, 2008.

[10] Leon A. Pintsov and Scott A. Vanstone. *Postal Revenue Collection in the Digital Age*. In Proceedings of Financial Cryptography. Springer-Verlag. 2000.

[11] Craig Costello, Colin Boyd, Juan Manuel Gonzalez Nieto, and Kenneth Koon-Ho Wong. *Delaying Mismatched Field Multiplications in Pairing Computations*. Arithmetic of Finite Fields. Lecture Notes in Computer Science Volume 6087, 2010, pp 196-214. Springer.

[12] Craig Costello and Douglas Stebila. *Fixed Argument Pairings*. Progress in Cryptology - LATINCRYPT 2010. Lecture Notes in Computer Science Volume 6212, 2010, pp 92-108. Springer.

[13] Paulo S. L. M. Barreto, Ben Lynn, Michael Scott. *Efficient Implementation of Pairing-Based Cryptosystems*. Journal of Cryptology (2004) 17: 321?334

[14] Paulo S. L. M. Barreto, Hae Y. Kim, Ben Lynn, and Michael Scott *Efficient Algorithms for Pairing-Based Cryptosystems*. CRYPTO 2002. LNCS, vol. 2442, pp. 354-368. Springer, Heidelberg (2002).

[15] Michael Scott. *On the Efficient Implementation of Pairing-Based Protocols*. IMACC'11 Proceedings of the 13th IMA international conference on Cryptography and Coding. p.p. 296-308.

[16] Victor S. Miller. *Use of elliptic curves in cryptography*. Lecture notes in computer sciences; 218 on Advances in cryptology - CRYPTO 85. Pages 417-426.

[17] Neal Koblitz. *Elliptic Curve Cryptosystems*. Mathematics of Computation, Vol. 48 (1987) pp. 203?209.

[18] Alfred Menezes, Scott Vanstone, Tatsuaki Okamoto. *Reducing Elliptic Curve Logarithms to Logarithms in a Finite Field*. STOC '91 Proceedings of the twenty-third annual ACM symposium on Theory of computing. pp. 80-89.

[19] Adi Shamir. *Identity-Based Cryptosystems and Signature Schemes*. Advances in Cryptology: Proceedings of CRYPTO 84, Lecture Notes in Computer Science, 7:47–53, 1984.

[20] Dan Boneh, Matthew K. Franklin. *Identity-Based Encryption from the Weil Pairing*. Advances in Cryptology - Proceedings of CRYPTO 2001 (2001)

[21] Dan Boneh, Ben Lynn, and Hovav Shacham. *Short Signatures from the Weil Pairing*. Journal of Cryptology 17 (2004). pp. 297?319.

[22] Xavier Boyen. *The BB1 Identity-Based Cryptosystem: A Standard for Encryption and Key Encapsulation*. IEEE P1363.3 draft. August 14, 2006.
[23] Brent Waters. *Ciphertext-Policy Attribute-Based Encryption: An Expressive, Efficient, and Provably Secure Realization*. Cryptology ePrint Archive, Report 2008/290.
[24] Zhen Liu and Zhenfu Cao. *On Efficiently Transferring the Linear Secret-Sharing Scheme Matrix in Ciphertext-Policy Attribute-Based Encryption*. Cryptology ePrint Archive, Report 2010/374.
[25] Amos Beimel. *Secure Schemes for Secret Sharing and Key Distribution*. PhD thesis, Israel Institute of Technology, Technion, Haifa, Israel, 1996.
[26] Antoine Joux. `https://listserv.nodak.edu/cgi-bin/wa.exe?A2=NMBRTHRY;49bb494e.1305`
[27] Pairing-Based Cryptography. `http://crypto.stanford.edu/pbc/`.
[28] The GNU Multiple Precision Arithmetic Library. `http://gmplib.org/`.
[29] MIRACL. `http://www.certivox.com`.

Appendix A
Source code

```c
#include <pbc.h>
#include "darray.h"
#include <stdbool.h>

typedef struct type_lnodepoly {
    struct type_lnodepoly *next;
    int n, stps;
    element_ptr polynomial, polynomialY;
} lnodepoly;

typedef struct type_lpoly {
    struct type_lnodepoly *first, *last;
    int MAXD; // Maximum degree in the list
} lpoly;

lpoly* lpoly_init(); // Init a new list of polynomials
void lpoly_free(lpoly *list); // Free memory

// Add a new merged polynomial to the list.
// n is the number of 'Doubling steps' it contains.
void lpoly_add(lpoly *list, element_ptr polynomial, element_ptr polynomialY, int n);

// To retrieve the merged polynomial
element_ptr get_polynomial(lnodepoly *element);
element_ptr get_polynomialY(lnodepoly *element);
int get_steps(lnodepoly *list); // Number of 'Doubling steps'

lnodepoly* first(lpoly *list);
lnodepoly* next(lnodepoly *list);
bool islast(lnodepoly *list);
```

FIG. A.1. *lpoly.h*. Specification of a linked list to store merged polynomials.

```
#include <pbc.h>
#include <pbc_poly.h>
#include "lpoly.h"

#ifndef __PBC_PRECOMP_H__
#define __PBC_PRECOMP_H__

// Change the value of N in Costello's optimization.
void change_NMAX(int n);

// To precompute the merged polynomials.
// Precomputed data is stored in *list
void precompute(lpoly *list, mpz_t q, element_t P);

// To evaluate the pairing using Costello's optimization.
void compute_miller(element_ptr out, lpoly *list, element_ptr Q, pairing_t pairing);
void compute_multimiller(element_ptr out, lpoly *list, element_ptr Q[], int n_prod,
     pairing_t pairing);

#endif
```

FIG. A.2. *precompute.h* Public specification of the precomputation algorithm.

FIG. A.3. *precompute.c* Implementation of A.2 for type D and G elliptic curves.

```
#include <pbc.h>
#include <pbc_poly.h>
#include "darray.h"
#include "precompute.h"
#include "lpoly.h"
#include <stdbool.h>
#include <stdlib.h>

/*************************************************************************
/ Fixed argument optimization.
/ Implementation for (d,g)-type curves
/
/ P in E(F_q)
/ Q in E(F_qd) --> E(F_q2d)
/ TWIST: (x,y) --> ( xv^{-1}, xv^{-2} sqrt{v} )
/
/ INFO: When computing G_N(X, Y) all X^i lie in F_qk (with Im = 0)
/ and all YX^i lie in F_qk (with Re = 0)
*************************************************************************/

element_ptr curve_equation;
int STEPMAX = 4;

void change_NMAX(int n){ STEPMAX = n; }

// For some reason, element_set0(element in F_p6) returns ([1,0,0],[0,0,0]) instead
     of ([0,0,0], [0,0,0])
void KSET0(element_t out){
    element_set0(out);
    element_ptr re_out = element_x(out);
    element_set0(element_item(re_out,0));
}

/************************************
   DATA STRUCTURE OF (D,G)-curves
************************************/

typedef struct {
  field_ptr field; // The field where the curve is defined.
  element_t a, b; // The curve is E: Y^2 = X^3 + a X + b.
  // cofac == NULL means we're using the whole group of points.
  // otherwise we're working in the subgroup of order #E / cofac.
  mpz_ptr cofac;
  // A generator of E.
  element_t gen_no_cofac;
  // A generator of the subgroup.
  element_t gen;
  // A non-NULL quotient_cmp means we are working with the quotient group of
  // order #E / quotient_cmp.
  mpz_ptr quotient_cmp;
} *curve_data_ptr;
```

```
typedef struct {
  field_t Fq, Fqx, Fqd, Fqk; // The fields F_q, F_q[x], F_q^d, F_q^k.
  field_t Eq, Etwist;      // The curves E(F_q) and E'(F_q^d).
  // Let v be the quadratic nonresidue used to construct F_q^k from F_q^d,
  // namely Fqk = Fqd[sqrt(v)].
  element_t nqrinv, nqrinv2; // The constants v^-1 and v^-2.
  mpz_t tateexp;           // The Tate exponent
  int k;                   // The embedding degree, usually 6.
  // Let x be the element used to build Fqd from Fq, i.e. Fqd = Fq[x].
  element_t xpowq, xpowq2; // x^q and x^{2q} in F_q^d.
} *pptr;

// Add an element x in F_q to a big element in F_qk
// Used when a polynomial is evaluated
void point_add(element_ptr e0, element_ptr y, element_t x){
   element_ptr x_out = element_x(e0); // e0 = x + iy
   element_ptr y_out = element_x(y); // e0 = x + iy
   element_add(element_item(x_out, 0), element_item(y_out, 0), x);
}
// Computes x*Y when x in F_q and y in F_qk (but Im = 0).
// Used when a polynomial in X is evaluated
void basic_mult(element_ptr out, element_t x, element_ptr y){
    element_ptr re_out = element_x(out); element_ptr re_y = element_x(y);
    int d = polymod_field_degree(re_out->field);
    for (int i = 0; i < d; i++) {
      element_mul(element_item(re_out, i), element_item(re_y, i), x);
    }
}
// Computes x*Y when x in F_q and y in F_qk (but Re = 0).
// Used when a polynomial in XY is evaluated
void basic_mult2(element_ptr out, element_ptr x, element_ptr y){
    element_ptr im_out = element_y(out); element_ptr im_y = element_y(y);
    int d = polymod_field_degree(im_out->field);
    for (int i = 0; i < d; i++) {
      element_mul(element_item(im_out, i), element_item(im_y, i), x);
    }
}
// Computes x + y when x, y in F_qk (but Im = 0)
// Use when a polynomial in X is evaluated
void basic_add(element_ptr out, element_ptr x, element_ptr y){
    element_ptr re_x = element_x(x); element_ptr re_y = element_x(y);
    element_ptr re_out = element_x(out);
    element_add(re_out, re_x, re_y);
}
// Computes x + y when x, y in F_qk (but Re = 0)
// Use when a polynomial in XY is evaluated
void basic_add2(element_ptr out, element_ptr x, element_ptr y){
    element_ptr im_x = element_y(x); element_ptr im_y = element_y(y);
    element_ptr im_out = element_y(out);
    element_add(im_out, im_x, im_y);
}
```

```
/*************************************
    PRECOMPUTATION ALGORITHMS
*************************************/

// Add a new step in the precomputed data structure
//
// list -> list of precomputed polynomials
// a, b, c -> coeff of the new step: aX + bY + C
// aa = 1 if the new step comes from Doubling; 0 from the Adding part.

void add_poly(lpoly *list, element_t a, element_t b, element_t c, int aa){
    // Init the new step
    field_ptr tbase = pbc_malloc(sizeof(*tbase));
    field_init_poly(tbase, a->field);

    element_ptr ppx, ppy;
    ppx = pbc_malloc( sizeof(*ppx));
    ppy = pbc_malloc( sizeof(*ppy));
    element_init(ppx, tbase);
    element_init(ppy, tbase);

    // p = ax + c + y(b)
    poly_set_coeff(ppy, b, 0); poly_set_coeff(ppx, c, 0); poly_set_coeff(ppx, a, 1);

    if(last(list)->n != -1 && STEPMAX > last(list)->n){
        // If not empty list
        // and previous merged polynomial includes less than STEPMAX Miller's loops.

        // INIT
        element_ptr polyx = last(list)->polynomial;
        element_ptr polyy = last(list)->polynomialY;

        element_ptr newx, newy, tmp1, tmp2;
        newx = pbc_malloc( sizeof(*newx));
        newy = pbc_malloc( sizeof(*newy));
        tmp1 = pbc_malloc( sizeof(*tmp1));
        tmp2 = pbc_malloc( sizeof(*tmp2));
        element_init(newx, tbase);
        element_init(newy, tbase);
        element_init(tmp1, tbase);
        element_init(tmp2, tbase);
        element_set0(newx); element_set0(newy);
        element_set0(tmp1); element_set0(tmp2);
```

```
        if(aa == 1){
            // Doubling step
            element_mul(tmp1, polyx, polyy);
            element_double(tmp1, tmp1);
            element_mul(tmp2, tmp1, ppy);
            element_mul(tmp2, tmp2, curve_equation);
            element_add(newx, newx, tmp2);
            element_mul(tmp2, tmp1, ppx);
            element_add(newy, newy, tmp2);

            element_square(tmp1, polyy);
            element_mul(tmp2, tmp1, ppy);
            element_mul(tmp2, tmp2, curve_equation);
            element_add(newy, newy, tmp2);
            element_mul(tmp2, tmp1, ppx);
            element_mul(tmp2, tmp2, curve_equation);
            element_add(newx, newx, tmp2);

            element_square(tmp1, polyx);
            element_mul(tmp2, tmp1, ppy);
            element_add(newy, newy, tmp2);
            element_mul(tmp2, tmp1, ppx);
            element_add(newx, newx, tmp2);
        }
        else{
            // If Adding, we only need to multiply the new line
            element_mul(tmp1, polyx, ppx);
            element_add(newx, newx, tmp1);
            element_mul(tmp1, polyy, ppy);
            element_mul(tmp1, tmp1, curve_equation);
            element_add(newx, newx, tmp1);

            element_mul(tmp1, polyx, ppy);
            element_add(newy, newy, tmp1);
            element_mul(tmp1, polyy, ppx);
            element_add(newy, newy, tmp1);
        }

        last(list)->polynomial = newx;
        last(list)->polynomialY = newy;

        last(list)->n = last(list)->n + aa;
        last(list)->stps = last(list)->stps + 1;

        // Update the maximum degree
        int d = poly_degree(last(list)->polynomial);
        if(d > list->MAXD){ list->MAXD = d; }
    }else{
        // First step or previous merged polynomial is full.
        lpoly_add(list, ppx, ppy, aa);
        last(list)->stps = 1;

        if(list->MAXD < 1){ list->MAXD = 1; }
    }
}
```

```
// Modified Miller algorithm:
// It doesn't compute the pairing, but the lines that are needed.
void precompute(lpoly *list, mpz_t q, element_t P) {
  element_t Z, a, b, c, t0, one; element_ptr Zx, Zy;
  const element_ptr cca = curve_a_coeff(P);
  const element_ptr Px = curve_x_coord(P);
  const element_ptr Py = curve_y_coord(P);

  curve_equation = pbc_malloc(sizeof(*curve_equation));
  field_ptr tbase = pbc_malloc(sizeof(*tbase));
  field_init_poly(tbase, curve_a_coeff(P)->field);
  element_init(one, curve_a_coeff(P)->field); element_set1(one);
  element_init(curve_equation, tbase);
  poly_set_coeff(curve_equation, one, 3);
  poly_set_coeff(curve_equation, curve_a_coeff(P), 1);
  poly_set_coeff(curve_equation, curve_b_coeff(P), 0);

  void do_tangent(void) {
    // a = -(3 Zx^2 + cc->a); b = 2 * Zy
    // c = -(2 Zy^2 + a Zx);
    element_square(a, Zx); element_mul_si(a, a, 3);
    element_add(a, a, cca); element_neg(a, a);
    element_add(b, Zy, Zy);
    element_mul(t0, b, Zy); element_mul(c, a, Zx);
    element_add(c, c, t0); element_neg(c, c);
    add_poly(list, a, b, c, 1);
  }

  void do_line(void) {
    // a = -(B.y - A.y) / (B.x - A.x); b = 1;
    // c = -(A.y + a * A.x); but we multiply by B.x - A.x to avoid division.
    element_sub(b, Px, Zx); element_sub(a, Zy, Py);
    element_mul(t0, b, Zy); element_mul(c, a, Zx);
    element_add(c, c, t0); element_neg(c, c);
    add_poly(list, a, b, c, 0);
  }

  element_init(a, Px->field); element_init(b, a->field);
  element_init(c, a->field); element_init(t0, a->field);
  element_init(Z, P->field); element_set(Z, P);
  Zx = curve_x_coord(Z); Zy = curve_y_coord(Z);

  int m = mpz_sizeinbase(q, 2) - 2;
  for(;;) {
    do_tangent();
    if (!m) break;
    element_double(Z, Z);
    if (mpz_tstbit(q, m)) {
      do_line(); element_add(Z, Z, P);
    }
    m--;
  }
  element_clear(Z); element_clear(a);
  element_clear(b); element_clear(c); element_clear(t0);
}
```

```
// Evaluates two polynomials: one in X and the other in XY.
void compute_polynomial(element_t out, element_t out2, lnodepoly *poly, element_ptr
     *point_precomp, element_ptr *point_precomp2)
{
    element_ptr p = get_polynomial(poly);
    int i;
    element_t tmp;
    element_init(tmp, out->field);

    d = poly_degree(p)+1;
    for(i=1; i < d; i++){
        KSET0(tmp);
        basic_mult(tmp, poly_get_coeff(p, i), point_precomp[i-1]);
        basic_add(out, out, tmp);
    }
    point_add(out, out, poly_get_coeff(p, 0));

    p = get_polynomialY(poly);
    d = poly_degree(p)+1;
    for(i=0; i < d; i++){
        element_set0(tmp);
        basic_mult2(tmp, poly_get_coeff(p, i), point_precomp2[i]);
        basic_add2(out2, out2, tmp);
    }
}

// As 'compute_polynomial', but adapted to be used in multipairings
void compute_polynomialN(element_t out, element_t out2, lnodepoly *poly, int row,
     int MAXD, element_ptr point_precomp[][MAXD], element_ptr
     point_precomp2[][MAXD])
{
    element_ptr p = get_polynomial(poly);
    int d = poly_degree(p)+1;

    int i;
    element_t tmp;
    element_init(tmp, out->field);

    for(i=1; i < d; i++){
        KSET0(tmp);
        basic_mult(tmp, poly_get_coeff(p, i), point_precomp[row][i-1]);

        basic_add(out, out, tmp);
    }
    point_add(out, out, poly_get_coeff(p, 0));

    p = get_polynomialY(poly);
    d = poly_degree(p)+1;

    element_init(tmp, out->field);
    for(i=0; i < d; i++){
        element_set0(tmp);
        basic_mult2(tmp, poly_get_coeff(p, i), point_precomp2[row][i]);
        basic_add2(out2, out2, tmp);
    }
}
```

```
/*********************************
      EVALUATION ALGORITHMS
*********************************/
void precomp_miller(element_t res, lpoly *list, element_ptr Qx, element_ptr Qy){
       lnodepoly *actual = first(list);

    element_ptr *point_precomp, *point_precomp2;

    point_precomp = (element_ptr *)malloc((list->MAXD)*sizeof(element_ptr));
    point_precomp2 = (element_ptr *)malloc((list->MAXD)*sizeof(element_ptr));

    point_precomp[0] = Qx; point_precomp2[0] = Qy;

    int i = 0;
    for(i = 1; i < list->MAXD; i++){
        point_precomp[i] = pbc_malloc(sizeof(element_ptr));
        element_init(point_precomp[i], point_precomp[0]->field);
        element_mul(point_precomp[i], point_precomp[i-1], Qx);

        point_precomp2[i] = pbc_malloc(sizeof(element_ptr));
        element_init(point_precomp2[i], point_precomp[0]->field);
        element_mul(point_precomp2[i], point_precomp2[i-1], Qx);
    }

    element_set1(res);
    element_t out, out2;
    element_init(out, res->field);
    element_init(out2, res->field);

    while(islast(actual) != true){

        int steps = get_steps(actual);
        while(steps > 0 && !element_is1(res)){
            element_square(res,res);
            steps--;
        }

        KSET0(out); KSET0(out2);

        compute_polynomial(out, out2, actual, point_precomp, point_precomp2);

        element_ptr im_out = element_y(out);
        element_set(im_out, element_y(out2));

        element_mul(res, res, out);

        actual = next(actual);
    }

    for(i=1; i < list->MAXD; i++){
        pbc_free(point_precomp[i]); pbc_free(point_precomp2[i]);
    }
    free(point_precomp); free(point_precomp2);
    element_clear(out); element_clear(out2);
}
```

```
void precomp_millers(element_t res, lpoly list[], element_t Qx[], element_t Qy[],
     int n_prod) {

    int MAXD = 0;
    int i = 0;
    for(i=0; i < n_prod; i++){
        if(list[i].MAXD > MAXD){
            MAXD = list[i].MAXD;
        }
    }

    element_ptr point_precomp[n_prod][MAXD];
    element_ptr point_precomp2[n_prod][MAXD];

    for(i = 0; i < n_prod; i++){
        point_precomp[i][0] = Qx[i];
        point_precomp2[i][0] = Qy[i];
    }

    int j = 0;
    for(j = 0; j < n_prod; j++){
        for(i = 1; i < MAXD; i++){
            point_precomp[j][i] = pbc_malloc(sizeof(element_ptr));
            element_init(point_precomp[j][i], Qx[0]->field);
            element_mul(point_precomp[j][i], point_precomp[j][i-1], Qx[j]);

            point_precomp2[j][i] = pbc_malloc(sizeof(element_ptr));
            element_init(point_precomp2[j][i], Qx[0]->field);
            element_mul(point_precomp2[j][i], point_precomp2[j][i-1], Qx[j]);
        }
    }

    element_set1(res);
    element_t out, out2;
    element_init(out, res->field);
    element_init(out2, res->field);

    lnodepoly *actuales[n_prod];
    for(i=0; i<n_prod; i++){
        actuales[i] = first(&list[i]);
    }

    int still_something_to_do(){
        int j = 0;
        for(j = 0; j < n_prod; j++){
            if(islast(actuales[j]) != true) return 1;
        }
        return 0;
    }
```

```
    while(still_something_to_do() == 1){
        int steps = get_steps(actuales[0]);
        while(steps > 0){
            element_square(res,res);
            steps--;
        }

        for(i = 0; i < n_prod; i++){
            if(islast(actuales[i]) != true){
                KSET0(out);
                KSET0(out2);

                compute_polynomialN(out, out2, actuales[i], i, MAXD, point_precomp,
                     point_precomp2);

                element_ptr im_out = element_y(out);
                element_set(im_out, element_y(out2));
                element_mul(res, res, out);
            }
        }

        for(i=0; i<n_prod; i++){
            if(islast(actuales[i]) != true) actuales[i] = next(actuales[i]);
        }
    }
}
```

```
/****************************
     PUBLIC METHODS
****************************/
void compute_miller(element_ptr out, lpoly *list, element_ptr Q, pairing_t pairing)
    {
  element_ptr Qbase = Q;
  element_t Qx, Qy;
  pptr p = pairing->data;

  element_init(Qx, p->Fqd);
  element_init(Qy, p->Fqd);
  // Twist: (x, y) --> (v^-1 x, v^-(3/2) y)
  // where v is the quadratic nonresidue used to construct the twist.
  element_mul(Qx, curve_x_coord(Qbase), p->nqrinv);
  // v^-3/2 = v^-2 * v^1/2
  element_mul(Qy, curve_y_coord(Qbase), p->nqrinv2);

  element_t tmp;
  element_init(tmp, out->field);
  element_set0(tmp);
  element_ptr im_out = element_y(tmp); // e0 = x + iy
  element_set(im_out, Qy);

  element_t tmp2;
  element_init(tmp2, out->field);
  element_set0(tmp2);
  im_out = element_x(tmp2); // e0 = x + iy
  element_set(im_out, Qx);

  precomp_miller(out, list, tmp2, tmp);

  pairing->finalpow(out);

  element_clear(Qx);
  element_clear(Qy);
  element_clear(tmp);
  element_clear(tmp2);
}
```

```
// 'compute_miller', but adapted to multipairings.
void compute_multimiller(element_ptr out, lpoly *list, element_ptr Q[], int n_prod,
    pairing_t pairing) {
  element_t Qx[n_prod], Qy[n_prod];
  element_t tmp[n_prod], tmp2[n_prod];
  pptr p = pairing->data;

  int i = 0;
  for(i = 0; i < n_prod; i++){
    element_init(Qx[i], p->Fqd);
    element_init(Qy[i], p->Fqd);

    element_mul(Qx[i], curve_x_coord(Q[i]), p->nqrinv);
    element_mul(Qy[i], curve_y_coord(Q[i]), p->nqrinv2);

    element_init(tmp[i], out->field);
    element_init(tmp2[i], out->field);
    element_set0(tmp[i]); KSET0(tmp2[i]);

    element_ptr im_out = element_y(tmp[i]); element_set0(im_out);
    element_ptr re_out = element_x(tmp[i]); element_set0(re_out);
    element_add(im_out, im_out, Qy[i]);
    im_out = element_x(tmp2[i]);
    element_add(im_out, im_out, Qx[i]);
  }

  precomp_millers(out, list, tmp2, tmp, n_prod);

  pairing->finalpow(out);
}
```

FIG. A.4. In type F elliptic curves, only point-operation code is modified in A.3

.

```
// Computes y*X when y in F_qd and X in F_qk
// The output is in F_qk
void point_mult(element_t e0, element_t x, element_t y) {
  element_ptr re_x = element_x(x); element_ptr im_x = element_y(x);
  element_ptr re_out = element_x(e0); element_ptr im_out = element_y(e0);

  element_mul(re_out, re_x, y);
  element_mul(im_out, im_x, y);
}

// Add an element x in F_q to a big element in F_qk
// Used when a polynomial is evaluated
void point_add(element_t e0, element_t y, element_t x){
   element_ptr re_out = element_x(e0); // e0 = x + iy
   element_ptr y_out = element_x(y); // e0 = x + iy
   element_add(element_item(re_out, 0), element_item(y_out, 0), x);
}

// Computes x*Y when x in F_q and y in F_qk.
// Used when a polynomial in X is evaluated
void basic_mult(element_t out, element_t x, element_ptr y){
    point_mult(element_item(out, 0), element_item(y, 0), x);
    point_mult(element_item(out, 2), element_item(y, 2), x);
    point_mult(element_item(out, 4), element_item(y, 4), x);
}

// Computes x*Y when x in F_q and y in F_qk.
// Used when a polynomial in XY is evaluated
void basic_mult2(element_t out, element_t x, element_ptr y){
    point_mult(element_item(out, 1), element_item(y, 1), x);
    point_mult(element_item(out, 3), element_item(y, 3), x);
    point_mult(element_item(out, 5), element_item(y, 5), x);
}

// Computes x + y when x, y in F_qk
// Used when a polynomial in X is evaluated
void basic_add(element_t out, element_t x, element_t y){
    for(int i=0; i < 6; i++){
        element_add(element_item(out,i), element_item(x,i), element_item(y,i));
    }
}

// Computes x + y when x, y in F_qk
// Used when a polynomial in XY is evaluated
void basic_add2(element_t out, element_t x, element_t y){ basic_add(out, x, y); }
```