Elliptic curve cryptography and the Weil pairing

Dias da Cruz Steve

A thesis presented for the degree of
Bachelor in Mathematics

Faculty of Sciences, Technology and Communication
University of Luxembourg
May 18, 2015

Abstract

In this short thesis, which is a part of my Bachelor degree in Mathematics at the university of Luxembourg, I will shortly describe some basic ideas about elliptic curves and their properties. Essentially, I will be interested in the group law and the computation of the addition inside an elliptic curve, which will later be used in different ways.

After that follows the definition of torsion points and divisors, which will be necessary for the most important part of this thesis: the Weil pairing and its application in cryptography. I am going to present two different possible definitions of the Weil pairing. The first one is easier to proof, but slower to compute when trying to implement. The second one is much faster for computations, but needs a little bit more work to be proven, which is why I simply show that both Weil Pairing definitions are equivalent. Afterwards, I present the Miller algorithm, which will be necessary for the implementation and calculation of the Weil Pairing.

In the last section, I am going to present some applications of the Weil Pairing and elliptic curves. There are some possible attack methods to break cryptographic applications, which are using elliptic curves, but on the other side, one can guarantee a high security level by applying the right methods and pairings to a system. There are a lot of possibilities and schemes, I chose to present two of them.

Last but not least, there will be a python implementation of elliptic curves, finite prime fields, the Weil pairing and a cryptographic scheme. Of course, there are still possible improvements to be done, why I invite the reader to analyze my implementations before using them for some real life applications.

I want to thank Professor Gabor Wiese for his help, confidence and for giving me the possibility to prepare a thesis about a vested interest.

Contents

1 Elliptic Curves

1.1 General definitions and properties

The general form of an elliptic curve is called the **generalized Weierstrass equation** and is given by the following equation

$$y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6, \tag{1}$$

where $a_1, \ldots, a_6$ are constants.

However, in this thesis, I will constantly work with the reduced form of an elliptic curve, which will be denoted by **E** and which is the graph of an equation of the form:

$$y^2 = x^3 + Ax + B, \tag{2}$$

where A and B are constants. This simplified form is called the **Weierstrass equation** for an elliptic curve. The unknowns **x** and **y**, as well as the constants **A** and **B** are usually elements of the fields $\mathbb{Q}$, $\mathbb{R}$ and $\mathbb{C}$ or of the finite fields of the form $\mathbb{F}_p$ and $\mathbb{F}_q$, where **p** is a prime number and $\mathbf{q} = p^k$, where $k \geq 1$.

The set of all point of an elliptic curve defined over a field $\mathbb{L} \supseteq \mathbb{K}$, is denoted by $E(\mathbb{L})$. By definition, this set always includes the point at infinity represented by ∞ and of coordinates (∞, ∞). We say, that a line is passing through the point ∞ if the line is vertical. Two vertical lines always intersect at the point ∞. Thus, the set of all points of an elliptic curve is given by

$$E(\mathbb{L}) = \{\infty\} \cup \{(x, y) \in \mathbb{L} \times \mathbb{L} \mid y^2 = x^3 + Ax + B\}. \tag{3}$$

In general, there is another condition, which has to be fulfilled. For elliptic curves, we do not allow singular points, i.e. multiple roots. The easiest way to verify if a given curve is an elliptic curve, is to check whether the following conditions is fulfilled or not:

$$\Delta = 4A^3 + 27B^2 \neq 0. \tag{4}$$

In the following of this thesis, I will neglect the discussions of the cases of fields of characteristic 2 or 3, because they will not be important for later cryptographic applications.

1.2 The group law

Let's suppose that we have two points

$$P_1 = (x_1, y_1), \qquad P_2 = (x_2, y_2), \tag{5}$$

of an elliptic curve E. We want to be able to produce a new, third point P_3 of the elliptic curve by using the points P_1 and P_2.

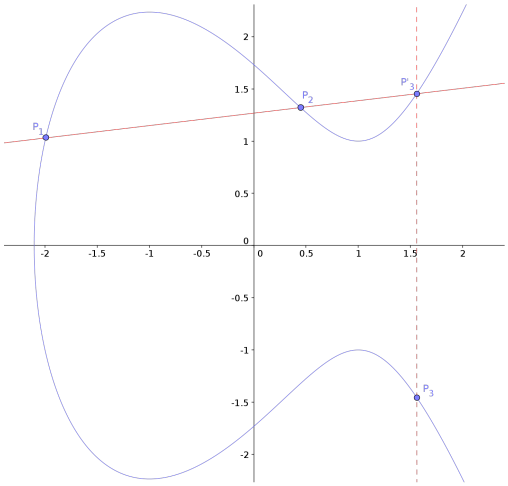

Figure 1: Adding two points on an elliptic curve of the form $y^2 = x^2 - 3x + 3$

The construction of the third point P_3 is given by the following steps:

1. Draw the line L passing through P_1 and P_2;
2. L intersects E at a third point P_3' (this is always possible);
3. Reflect P_3' across the x-axis to obtain P_3.

This construction will be denoted as the **addition** of two points of the elliptic curve and be represented by

$$P_1 + P_2 = P_3. \tag{6}$$

As the addition of two points is not given by the simple addition of the coordinates, we will need a formula to simply the computation. We need to distinguish between five different cases to get the coordinates of $P_3 = (x_3, y_3)$:

Theorem 1.1
Let E be an elliptic curve defined by $y^2 = x^3 + Ax + B$. Let $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$ be points of E. Then $P_3 = P_1 + P_2$ is described by the following cases:

Case 1: $P_1 \neq P_2$ **and** $x_1 \neq x_2$

$$x_3 = m^2 - x_1 - x_2, \qquad y_3 = m(x_1 - x_3) - y_1, \qquad \text{where } m = \frac{y_2 - y_1}{x_2 - x_1}$$

Case 2: $P_1 \neq P_2$ **and** $x_1 = x_2$

$$P_1 + P_2 = \infty$$

Case 3: $P_1 = P_2$ **and** $y_1 \neq 0$

$$x_3 = m^2 - 2x_1, \qquad y_3 = m(x_1 - x_3) - y_1, \qquad \textit{where } m = \frac{3x_1^2 + A}{2y_1}$$

Case 4: $P_1 = P_2$ **and** $y_1 = 0$

$$P_1 + P_2 = \infty$$

Case 5: $P_2 = \infty$

$$P_1 + \infty = P_1$$

By the above construction, $P_1 + P_2$ will have coordinates in $\mathbb{L}$. The addition in $E(\mathbb{L})$ is closed.

Theorem 1.2
The addition of points on an elliptic curve E, which was described in the previous theorem, satisfies the following properties:

1. *(commutativity)* $P_1 + P_2 = P_2 + P_1, \qquad \forall\ P_1, P_2$ *on* E.
2. *(existence of identity)* $P + \infty = P, \qquad \forall\ P$ *on* E.
3. *(existence of inverse)* $\forall P \in E \qquad \exists P' \in E \qquad | \ P + P' = \infty$
   P' *will be denoted as* $-P$.
4. *(associativity)* $(P_1 + P_2) + P_3 = P_1 + (P_2 + P_3), \qquad \forall P_1, P_2, P_3 \in E$.

The points on the elliptic curve E form an ***additive abelian group****, where* ∞ *is the identity element.*

Proof.

1. The line passing through P_1 and P_2 is the same as the line passing through P_2 and P_1.
2. Holds by definition.

3. Let P' be the reflection of P across the x-axis. Then $P' + P = \infty$

4. Associativity can easily be checked by computing the different possible cases using the different formulas.

□

At this point, I want to point out that the coordinates of $-P$, where $P = (x, y)$, are $-P = (x, -y)$. This is only true for the Weierstrass equation and not for the generalized Weierstrass equation.

Theorem 1.3
*The **double-and-add** algorithm is essentially to compute in a fast way a multiple of a point. Let k be a positive integer and let P be a point on an elliptic curve. The following algorithm computes kP in a faster way:*

1. *Start with $a = k$, $B = \infty$, $C = P$*
2. *If a is even, let $a = \frac{a}{2}$ and let $B = B$, $C = 2C$.*
3. *If a is odd, let $a = a - 1$ and let $B = B + C$, $C = C$*
4. *If $a \neq 0$, go to step 2.*
5. *Output B*

The output B is kP.

2 Weil Pairing

2.1 Torsion Points

Let E be an elliptic curve defined over a field $\mathbb{K}$. Let $\overline{\mathbb{K}}$ be the algebraic closure of $\mathbb{K}$ and let n be a positive integer. The **n-torsion points** are the points fulfilling the following condition:

$$E[n] = \{P \in E(\overline{\mathbb{K}}) \mid nP = \infty\} \tag{7}$$

Example 2.1
If the characteristic of $\mathbb{K}$ is not 2, then a point P satisfies the condition $2P = \infty$ iff the tangent line at P is vertical. This means that $y = 0$ and we get the following solution:

$$E[2] = \{\infty, (e_1, 0), (e_2, 0), (e_3, 0)\}, \tag{8}$$

where e_1, e_2, e_3 satisfy the representation of E under the form

$$y^2 = (x - e_1)(x - e_2)(x - e_3). \tag{9}$$

Example 2.2
If we want to determine $E[3]$ for a field $\mathbb{K}$ of characteristic not 2 and not 3, then a point P satisfies the condition $3P = \infty$ iff $2P = -P$. This means that the x-coordinate of $2P$ equals the x-coordinate of P. This leads to 9 possible points.

Theorem 2.1
Let E be an elliptic curve defined over a field $\mathbb{K}$ and let n be a positive integer. If the characteristic of $\mathbb{K}$ does not divide n, or is 0, then

$$E[n] \simeq \mathbb{Z}/n\mathbb{Z} \oplus \mathbb{Z}/n\mathbb{Z}.$$

If the characteristic of $\mathbb{K}$ is $p > 0$ and $p \mid n$, write $n = p^r n'$ with $p \nmid n'$. Then

$$E[n] \simeq \mathbb{Z}/n'\mathbb{Z} \oplus \mathbb{Z}/n'\mathbb{Z} \qquad or \qquad \mathbb{Z}/n\mathbb{Z} \oplus \mathbb{Z}/n'\mathbb{Z}.$$

An elliptic curve E in characteristic p is called **ordinary** if $E[p] \simeq \mathbb{Z}/p\mathbb{Z}$. It is called **supersingular** if $E[p] \simeq 0$.

Lemma 2.1
Let q be odd and $q \equiv 2 \pmod 3$. Let $B \in \mathbb{F}_q^\times$. Then the elliptic curve given by $y^2 = x^3 + B$ is supersingular.

Proof.
Let $\Phi : \mathbb{F}_q^\times \to \mathbb{F}_q^\times$ be the homomorphism defined by $\Phi(x) = x^3$. Since $q \equiv 2 \pmod 3$, this implies that 3 does not divide $q - 1$. Thus, there are no elements of order 3 in $\mathbb{F}_q^\times$. Thus, $\ker(\Phi) = 1 + (q)$, thus trivial. Since the cardinality of the source is the same as the cardinality of the target set, the homomorphism has to be surjective, thus an isomorphism. Consequently, for each y, there is exactly one x, such that the point $P = (x, y) \in E$. This is equivalent to saying, that $y^2 - B$ is the unique cube root of x, which is precisely what the isomorphism implies. There are q possible values for y, thus we obtain q points. Including the point ∞, we finally get that

$$|E(\mathbb{F}_q)| = q + 1.$$

Therefore, E is supersingular. □

Let n be a positive integer not divisible by the characteristic of $\mathbb{K}$. We can then choose a **basis** $\{\beta_1, \beta_2\}$ for $E[n] \simeq \mathbb{Z}/n\mathbb{Z} \oplus \mathbb{Z}/n\mathbb{Z}$. This means, that for every element $P \in E[n]$ there are integers m_1 and m_2 such that $P = m_1\beta_1 + m_2\beta_2$.

2.2 Divisors

Let E be an elliptic curve defined over a field $\mathbb{K}$. We define $[P]$ to be the symbol to represent a point $P \in E(\overline{\mathbb{K}})$. A **divisor** D on E is a finite linear combination of such points of E with integer coefficients defined the following way:

$$D = \sum_j a_j[P_j], \qquad a_j \in \mathbb{Z}. \tag{10}$$

The **group of divisors** of E is denoted as $\mathrm{Div}(E)$. The **sum** and the **degree** of a divisor is defined by:

$$\deg(\sum_j a_j[P_j]) = \sum_j a_j \in \mathbb{Z} \tag{11}$$

$$\mathrm{sum}(\sum_j a_j[P_j]) = \sum_j a_j P_j \in E(\overline{\mathbb{K}}) \tag{12}$$

The sum actually sums up the different P's by using the group law of the elliptic curve and thus is a point P of $E(\overline{\mathbb{K}})$. The degree sums up the integer coefficients of the points of the divisor, thus is an integer. The set of all divisors whose degree is equal to 0 form an important subgroup of $\mathrm{Div}(E)$, denoted $Div^0(E)$. A **function** on E is a rational function $f(x, y) \in \overline{\mathbb{K}}(x, y)$, which is defined for at least one point in $E(\overline{\mathbb{K}})$. Thus, there exists a point $P = (a, b) \in \overline{\mathbb{K}}(x, y)$ such that $f(a, b)$ is defined. A function has a **zero** at a point P, if $f(P) = 0$ and it has a **pole** at a point P', if $f(P') = \infty$.

If f is a function on E that is not identically 0, then we define the **divisor of f** by

$$\mathrm{div}(f) = \sum_{P \in E(\overline{\mathbb{K}})} \mathrm{ord}_P(f)[P] \in \mathrm{Div}(E), \tag{13}$$

which is a finite sum and where

$$\mathrm{ord}_P(f)[P]$$

defines the **order** of f at P (i.e. the order of the zero or pole at P).

Theorem 2.2
Let E be an elliptic curve. Let D be a divisor on E with $\deg(D) = 0$. *Then there is a function f on E with*

$$\mathrm{div}(f) = D \qquad \Longleftrightarrow \qquad \mathrm{sum}(D) = \infty \tag{14}$$

Lemma 2.2
Let f and h be two functions on E and suppose that $\mathrm{div}(f)$ *and* $\mathrm{div}(h)$ *have no points in common. Then,*

$$f(\mathrm{div}(h)) = h(\mathrm{div}(f))$$

This lemma is known under the name of ***Weil reciprocity***

Example 2.3
Consider the elliptic curve E over $\mathbb{F}_{11}$ given by

$$y^2 = x^3 + 4x.$$

Let

$$D = [(0,0)] + [(2,4)] + [(4,5)] + [(6,3)] - 4[\infty]$$

Then $\deg(D) = 1 + 1 + 1 + 1 - 4 = 0$ *and it is clear that* $\mathrm{sum}(D) = \infty$. *Therefore, D is the divisor of a function. We now want to find that function. The line through $(0,0)$ and $(2,4)$ is $y - 2x = 0$. It is tangent to E at $(2,4)$, thus its order is 2. So,*

$$\mathrm{div}(y - 2x) = [(0,0)] + 2[(2,4)] - 3[\infty].$$

The vertical line through $(2,4)$ is $x - 2 = 0$, but $(2,-4) \in E$ and $(2,-4) \in x - 2 = 0$, thus

$$\mathrm{div}(x - 2) = [(2,4)] + [(2,-4)] - 2[\infty].$$

The trick to find easily the order of $[\infty]$ is, that $\mathrm{sum}(div) = 0$. *Therefore, we can express certain coefficients of D by the divisor:*

$$D = [(2,4)] + \mathrm{div}\left(\frac{y - 2x}{x - 2}\right) + [(4,5)] + [(6,3)] - 3[\infty]$$

Because

$$\mathrm{div}\left(\frac{f(x)}{g(x)}\right) = \mathrm{div}(f(x)) - \mathrm{div}(g(x))$$

Similarly, one can find the following replacement:

$$[(4,5)] + [(6,3)] = [(2,4)] + [\infty] + \mathrm{div}\left(\frac{y + x + 2}{x - 2}\right)$$

Thus, we can express the divisor the following way:

$$D = [(2,-4)] + \mathrm{div}\left(\frac{y - 2x}{x - 2}\right) + [(2,4)] + \mathrm{div}\left(\frac{y + x + 2}{x - 2}\right) - 2[\infty]$$

Since we have already computed $\operatorname{div}(x-2)$ *one can replace the last coefficients and get:*

$$D = \operatorname{div}(x-2) + \operatorname{div}\left(\frac{y-2x}{x-2}\right) + \operatorname{div}\left(\frac{y+x+2}{x-2}\right)$$

Which finally leads to

$$D = \operatorname{div}\left(\frac{(y-2x)(y+x+2)}{x-2}\right) = \operatorname{div}(x^2-y)$$

2.3 Weil Pairing definition

The Weil Pairing on the n-torsion of an elliptic curve is very important in the field of cryptography. It can be used to attack the discrete logarithm problem for elliptic curves, but it can also be used to develop cryptosystems.

Let E be an elliptic curve over a field $\mathbb{K}$ and let n be a positive integer not divisible by the characteristic of $\mathbb{K}$. As seen before, we then get the isomorphism $E[n] \simeq \mathbb{Z}/n\mathbb{Z} \oplus \mathbb{Z}/n\mathbb{Z}$. The **group of nth roots of unity** in $\overline{\mathbb{K}}$ is defined as follow:

$$\mu_n = \{x \in \overline{\mathbb{K}} \mid x^n = 1\} \tag{15}$$

Since the characteristic of $\mathbb{K}$ does not divide n, the equation $x^n = 1$ has no multiple roots, hence has n roots in $\overline{\mathbb{K}}$ (the algebraic closure of a field contains a root for every non-constant polynomial). Therefore, μ_n is a cyclic group of order n (μ_n is a finite abelian group. $\exp(\mu_n) = n = |\mu_n|$, because $\operatorname{char}(\mathbb{K}) \nmid n$. Thus, μ_n is cyclic.) Any generator ζ of μ_n is called a **primitive nth root of unity**. This is equivalent to saying that $\zeta^k = 1$ if and only if n divides k. (i.e. the set μ_n is the set of all x such that the order of x is n or a divisor of n).

Let $T \in E[n] \subseteq E(\mathbb{K})$. By Theorem 2.2, there exists a function f such that

$$\operatorname{div}(f) = n[T] - n[\infty]. \tag{16}$$

Proof.

We just need to proof that $D = n[T] - n[\infty]$ is a divisor on E fulfilling all conditions. As $n - n = 0$, the condition $\deg(D) = 0$ is fulfilled. Thus, there exists a function f on E. $nT = \infty$, because $T \in E[n]$. Thus, $\operatorname{sum}(D) = \infty - n\infty = \infty$ Consequently, $\operatorname{div}(f) = D = n[T] - n[\infty]$. □

Let us now choose $T' \in E[n^2]$. Thus, $n^2T' = \infty = nT \Rightarrow nT' = T$ By Theorem 2.2, there exists a function g such that,

$$\operatorname{div}(g) = \sum_{R \in E[n]} ([T'+R] - [R]). \tag{17}$$

Proof.

We need to to verify that $\text{sum}(\text{div}(g)) = \infty$. We know that there are n^2 points R in $E[n]$ (because $E[n]$ is isomorphic to $\mathbb{Z}/n\mathbb{Z} \oplus \mathbb{Z}/n\mathbb{Z}$). The points R in $\sum[T' + R]$ and $\sum[R]$ cancel, so the sum is $n^2T' = nT = \infty$. □

Let $f \circ n$ be such that $f(n(P)) = f(nP)$. Define the points $P = T' + R$ with $R \in E[n]$. Those points P verify $nP = T$ (because $nP = nT' + nR$, but $nT' = T$ and $nR = \infty$ =identity element). Using (16) implies

$$\text{div}(f \circ n) = n\Big(\sum_R [T' + R]\Big) - n\Big(\sum_R [R]\Big) = n\,\text{div}(g) = \text{div}(g^n). \tag{18}$$

Therefore, $f \circ n$ is a constant multiple of g^n. By multiplying f by a suitable constant, we may assume that

$$f \circ n = g^n. \tag{19}$$

Let $S \in E[n]$ and let $P \in E(\overline{\mathbb{K}})$. Then

$$g(P + S)^n = f(n(P + S)) = f(nP + nS) = f(nP) = g(P)^n. \tag{20}$$

Therefore,

$$\frac{g(P + S)^n}{g(P)^n} = 1 \iff \Big(\frac{g(P + S)}{g(P)}\Big)^n = 1 \tag{21}$$

Thus,

$$\frac{g(P + S)}{g(P)} \in \mu_n. \tag{22}$$

Finally, the **Weil pairing** is defined and calculated by

$$e_n(S, T) = \frac{g(P + S)}{g(P)}. \tag{23}$$

This definition is independent of the choice of P and g (because g is determined up to a constant multiple of its divisor).

Theorem 2.3

Let E be an elliptic curve defined over a field $\mathbb{K}$ an let n be a positive integer. Assume that the characteristic of $\mathbb{K}$ does not divide n. The ***Weil pairing***

$$e_n : E[n] \times E[n] \to \mu_n,$$

then satisfies the following properties:

1. *e_n is bilinear in each variable. This means that*

$$e_n(S_1 + S_2, T) = e_n(S_1, T)e_n(S_2, T)$$

and

$$e_n(S, T_1 + T_2) = e_n(S, T_1)e_n(S, T_2)$$

for all $S, S_1, S_2, T, T_1, T_2 \in E[n]$.

2. *e_n is nondegenerate in each variable. This means that if $e_n(S, T) = 1$ for all $T \in E[n]$ then $S = \infty$ and also if $e_n(S, T) = 1$ for all $S \in E[n]$ then $T = \infty$.*

3. *$e_n(T, T) = 1$ for all $T \in E[n]$.*

4. *$e_n(T, S) = e_n(S, T)^{-1}$ for all $S, T \in E[n]$.*

5. *$e_n(\sigma(S), \sigma(T)) = \sigma(e_n(S, T))$ for all automorphisms σ of $\overline{\mathbb{K}}$ such that σ is the identity map on the on the coefficients of E (if E is in Weierstrass form, this means that $\sigma(A) = A$ and $\sigma(B) = B$).*

6. *$e_n(\alpha(S), \alpha(T)) = e_n(S, T)^{\deg(\alpha)}$ for all separable endomorphisms α of E. If the coefficients of E lie in a finite filed $\mathbb{F}_q$, then the statement also holds when α is the Frobenius endomorphism θ_q. (Actually, the statement holds for all endomorphisms α, separable or not.)*

Lemma 2.3
Let suppose that $\{T_1, T_2\}$ is a basis of $E[n]$. Then $e_n(T_1, T_2)$ is a primitive nth root of unity.

Proof.

Suppose that $e_n(T_1, T_2) = \xi$ with $\xi^d = 1$ (by the definition of the Weil Pairing, we get an element x such that $x^d = 1$). By the previous theorem, the following properties are verified. $e_n(T_1, dT_2) = e_n(T_1, T_2)^d = \xi^d = 1$. Also $e_n(T_2, dT_2) = e_n(T_2, T_2)^d = 1$ (by (1) and (3)) Let $S \in E[n]$. Then there are integers a and b such that $S = aT_1 + bT_2$ (because T_1 and T_2 form a basis). Therefore,

$$e_n(S, dT_2) = e_n(aT_1 + bT_2, dT_2) = e_n(aT_1, dT_2)e_n(bT_2, dT_2),$$

which implies that

$$e_n(S, dT_2) = e_n(T_1, dT_2)^a e_n(T_2, dT_2)^b = 1.$$

Since this hold for all S, (2) implies that $dT_2 = \infty$. Since $dT_2 = \infty$ if and only if $n \mid d$, it follows that ξ is a primitive nth root of unity. □

2.4 Faster Weil Pairing computation

The computation method for the Weil Pairing described in the previous section is good for smaller examples. However, in order to avoid massive computation, one needs to introduce an alternative, faster computation method.

Theorem 2.4
Let $S, T \in E[n]$. Let D_S and D_T be divisors of degree 0 such that

$$\text{sum}(D_S) = S \qquad \textit{and} \qquad \text{sum}(D_T) = T \tag{24}$$

and such that D_S and D_T have no points in common. Let f_S and f_T be functions such that

$$\text{div}(f_S) = nD_S \qquad \textit{and} \qquad \text{div}(f_T) = nD_T. \tag{25}$$

Then the Weil pairing is given by

$$e_n(S,T) = \frac{f_T(D_S)}{f_S(D_T)} \tag{26}$$

Remark 2.1
$f(\sum a_i[P_i]) = \Pi_i f(P_i)^{a_i}$

Remark 2.2
A natural choice of divisors is

$$D_S = [S] - [\infty], \qquad D_T = [T+R] - [R] \tag{27}$$

for some randomly chosen point R. Then we have

$$e_n(S,T) = \frac{f_S(R) f_T(S)}{f_S(T+R) f_T(\infty)} \tag{28}$$

Example 2.4
Let E be the elliptic curve over $\mathbb{F}_7$ defined by

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

Then

$$E(\mathbb{F}_7)[3] \simeq \mathbb{Z}/3\mathbb{Z} \oplus \mathbb{Z}/3\mathbb{Z}$$

We want to compute

$$e_3((0,3),(5,1)),$$

where (0,3) and (5,1) are two points of E. So, we need to choose and calculate our divisors by the previous remark:

$$D_{(0,3)} = [(0,3)]-[\infty], \qquad D_{(5,1)} = [((5,1)+(6,1))]-[(6,1)] = [(3,6)]-[(6,1)]$$

Recall: (5,1)+(6,1) is calculated by using the formula of section 1.2.

$$m = \frac{y_2 - y_1}{x_2 - x_1} = \frac{1-1}{6-5} = 0$$

$$\Rightarrow x_3 = m^2 - x_1 - x_2 = 0 - 6 - 5 = -11 = 3$$

$$\Rightarrow y_3 = m(x_1 - x_3) - y_1 = 0 - 1 = -1 = 6$$

For finding the two functions, one can proceed the same way as in example 2.3. $y-3$ is the line passing through $(0,3)$ and ∞. $4x-y+1$ is the line passing through $(3,6)$. $5x-y-2$ is the line passing through $(6,1)$. Thus,

$$\operatorname{div}(y-3) = 3D_{(0,3)}, \qquad \operatorname{div}\left(\frac{4x-y+1}{5x-y-1}\right) = 3D_{(5,1)},$$

because

$$\operatorname{div}\left(\frac{4x-y+1}{5x-y-1}\right) = \operatorname{div}(4x-y+1)-\operatorname{div}(5x-y-1) = 3[(3,6)]-3[(6,1)] = 3D_{(5,1)}$$

$[(3,6)]$ is the only point of E on $4x-y+1$, thus its order is 3. $[(6,1)]$ is the only point of E on $5x-y-1$, thus its order is 3. Therefore, we take

$$f_{(0,3)} = y-3, \qquad f_{(5,1)} = \frac{4x-y+1}{5x-y-1}.$$

We have

$$f_{(0,3)}(D_{(5,1)}) = \frac{f_{(0,3)}(3,6)}{f_{(0,3)}(6,1)} = \frac{6-3}{1-3} \equiv 2 \pmod 7$$

Similarly,

$$f_{(5,1)}(D_{(0,3)}) = 4.$$

Therefore,

$$e_3((0,3),(5,1)) = \frac{4}{2} \equiv 2 \pmod 7.$$

The number 2 is a cube root of unity, since $2^3 \equiv 1 \pmod 7$. The easiest way to compute $f_{(5,1)}(\infty)$ is to use projective coordinates:

$$f_{(5,1)}(x:y:z) = \frac{4x-y+z}{5x-y-z}.$$

Then

$$f_{(5,1)}(\infty) = f_{(5,1)}(0:1:0) = 1.$$

Theorem 2.5
In order to implement the Weil Pairing, one needs to find functions fulfilling the definition of the pairing and which will be evaluated at the given points. Let $E(\mathbb{K})$ be an elliptic curve, let $P, Q \in E(\mathbb{K})$ such that $P \neq Q$. Then we can compute the Weil Pairing by

$$e_n(P,Q) = (-1)^n \frac{f_{n,P}(Q)}{f_{n,Q}(P)},$$

*which can be obtained by **Millers algorithm**. The support of the divisors still need to be different, i.e. P, Q are linearly independent. Usually, the calculation would require to find the functions and to calculate their divisor at given points. The method given by the algorithm is much simpler and efficient to implement.*

Algorithm 1 Millers algorithm using double-and-add

Require: Elliptic curve $E(\mathbb{K})$, points $P, Q \in E(\mathbb{K}) \setminus \{\infty\}$, positive integer n in its binary representation $n = \sum_{j=0}^{m} b_j 2^j$
Ensure: value $t \in \mathbb{Z}_n$

```
t ← 1
V ← P
i ← m − 1
while i > −1 do
    t ← t² · g_{V,V}(Q)
    V ← 2V
    if b_i = 1 then
        t ← t · g_{V,P}(Q)
        V ← V + P
    end if
    i ← i − 1
end while
return t
```

2.5 Equivalence of the two Weil Pairing computations

I presented two different definitions and computations for the Weil Pairing. However, one still needs to prove that these two definitions are equivalent in order to prove that the second Weil Pairing is actually correct. This section is entirely devoted to prove this statement. Through this section, e_n will denote the pairing I defined in section 2.3 by the equation (23). As a reminder, this means that

$$e_n(S,T) = \frac{g(P+S)}{g(P)}$$

Let $V, W \in E[n^2]$. As in the definition of section 2.3, let's introduce

$$\mathrm{div}(f_{nV}) = n[nV] - n[\infty], \qquad g_{nV}^n = f_{nV} \circ n.$$

As a reminder, in section 2.3 we took $f \circ n = g^n$, $T' \in E[n^2]$ such that $nT' = T \in E[n]$ and $\mathrm{div}(f) = n[T] - n[\infty]$. Then, we can define

$$c(nV, vW) = \frac{f_{nV+nW}(X)}{f_{nV}(X) f_{nW}(X - nV)}, \qquad d(V,W) = \frac{g_{nV+nW}(X)}{g_{nV}(X) g_{nW}(X - V)},$$

where X is a variable point on E. However, the left hand side of the equations does not include X. The reason for that, is given by the following lemma:

Lemma 2.4
c(nV,nW) and d(V,W) are constants and

$$d(V,W)^n = c(nV, nW).$$

Proof.

First, we need to prove that $\mathrm{div}(c(nV,nW)) = 0$, which will prove that $c(nV,nW)$ is constant:

$$\mathrm{div}(c(nV,nW)) = n[nV+nW] - n[\infty] - (n[nV] - n[\infty] + n[nW] - n[\infty])$$

and thus,

$$\mathrm{div}(c(nV,nW)) = n[nV]+n[nW]-n[\infty]-n[\infty]-n[nV]+n[\infty]-n[nW]+n[\infty] = 0$$

The computation for $\mathrm{div}(d(V,W))$ is identical and will also result in 0, thus, $d(V,W)$ is also constant. Second, since $g_{nV}^n = f_{nV} \circ n$, we have

$$d(V,W)^n = \frac{f_{nV+nW}(nX)}{f_{nV}(nX) f_{nW}(nX - nV)} = \frac{f_{nV+nW}(X')}{f_{nV}(X') f_{nW}(X' - nV)} = c(nV, nW),$$

because $c(nV,nW)$ is independent of X. □

For the next few lemmas, we choose points U,V and W such that $U, V, W \in E[n^2]$ and we will try to relate c and d.

Lemma 2.5
$d(V, W + nU) = d(V, W)$ *and* $d(V + nU, W) = d(V, W)e_n(nU, nW)$.

Proof.
First, notice the following

$$n(W + nU) = nW + n^2U = nW + \infty = nW.$$

This implies that the functions $g_{n(W+nU)}$ and g_{nW} are equal. Therefore,

$$d(V, W+nU) = \frac{g_{nV+n(W+nU)}(X)}{g_{nV}(X)g_{n(W+nU)}(X-V)} = \frac{g_{nV+nW}(X)}{g_{nV}(X)g_{nW}(X-V)} = d(V, W).$$

In the same way $n(V + nU) = nV$ and one can prove that

$$d(V+nU, W) = \frac{g_{n(V+nU)+nW}(X)}{g_{n(V+nU)}(X)g_{nW}(X-V-nU)} = \frac{g_{nV+nW}(X)}{g_{nV}(X)g_{nW}(X-V-nU)}.$$

We can manipulate the last equality by

$$\frac{g_{nV+nW}(X)}{g_{nV}(X)g_{nW}(X-V-nU)}\frac{g_{nW}(X-V)}{g_{nW}(X-V)} = \frac{g_{nV+nW}(X)}{g_{nV}(X)g_{nW}(X-V)}\frac{g_{nW}(X-V)}{g_{nW}(X-V-nU)}$$

and conclute finally that

$$d(V + nU, W) = d(V, W)e_n(nU, nW),$$

where for the last equations one uses the definition of the Weil Pairing. □

Lemma 2.6

$$\frac{d(U, V)}{d(V, U)} = \frac{d(V, W)d(U + W, V)}{d(V, U + W)d(W, V)}$$

Proof.
By the definition of d and by multiplying the equality by the denominator we get:

$$g_{nU+(nV+nW)}(X) = d(U, V + W)g_{nU}(X)g_{nV+nW}(X - U)$$

and applying the definition of d another time for $g_{nV+nW}(X - U)$ one gets

$$g_{nV+nW}(X - U) = d(V, W)g_{nV}(X - U)g_{nW}(X - U - V),$$

because it is independent of X and one could choose $X = X - U$. Thus, we get

$$g_{nU+(nV+nW)}(X) = d(U, V+W)g_{nU}(X)d(V,W)g_{nV}(X-U)g_{nW}(X-U-V).$$

In the same way, one can compute the following expression

$$g_{(nU+nV)+nW}(X) = d(U+V, W)g_{nU}(X)d(U,V)g_{nV}(X-U)g_{nW}(X-U-V).$$

. Since $g_{nU+(nV+nW)} = g_{(nU+nV)+nW}$, we can cancel equal terms and get

$$d(U, V+W)d(V,W) = d(U+V, W)d(U,V) \tag{29}$$

Consequently,

$$d(U,V) = \frac{d(U, V+W)d(V,W)}{d(U+V, W)}$$

and

$$d(V,U) = \frac{d(V, U+W)d(U,W)}{d(U+V, W)}$$

which leads to

$$\frac{d(U,V)}{d(V,U)} = \frac{d(U, V+W)d(V,W)}{d(V, U+W)d(U,W)} \tag{30}$$

Now, let's switch V and W in (29) and solve for $d(U,W)$

$$d(U,W) = \frac{d(U, V+W)d(W,V)}{d(U+W, V)}$$

and replace $d(U,W)$ in equation (30), which leads to

$$\frac{d(U,V)}{d(V,U)} = \frac{d(U, V+W)d(V,W)d(U+W,V)}{d(V, U+W)d(U, V+W)d(W,V)} = \frac{d(V,W)d(U+W,V)}{d(V, U+W)d(W,V)}$$

□

Remark 2.3
The left-hand side of the previous equation does not depend on W. This will be useful for the proof of the next lemma.

Lemma 2.7
Let $S, T \in E[n]$. Then

$$e_n(S,T) = \frac{c(S,T)}{c(T,S)}.$$

Proof.

Choose $U, V \in E[n^2]$ such that $nU = S, nV = T$. By the previous remark, the expression proved in lemma 2.6 does not depend on W. Thus, we can evaluate that expression at different W's. Let's take $W = jU$ for $0 \leq j < n$. By lemma 2.4 we get

$$\frac{c(S,T)}{c(T,S)} = \frac{c(nU,nV)}{c(nV,nU)} = \Big(\frac{d(U,V)}{d(V,U)}\Big)^n$$

As these functions d do not depend on W, we can take a different $W = jU$ for every factor. Applying lemma 2.6 for each of these n factors, we get

$$\Big(\frac{d(U,V)}{d(V,U)}\Big)^n = \prod_{j=0}^{n-1} \frac{d(V,jU)d(U+jU,V)}{d(V,U+jU)d(jU,V)}$$

Almost all factors cancel out, except for some of the cases $j = 0$ and $j = n-1$. Additionaly $0U = \infty$., Thus, we get

$$\frac{c(S,T)}{c(T,S)} = \frac{d(V,\infty)d(nU,V)}{d(V,nU)d(\infty,V)} \tag{31}$$

Now, using lemma 2.5, we replace in the first equation $W = \infty$ and we get

$$d(V, \infty + nU) = d(V, nU) = d(V, \infty)$$

Using the second equation of lemma 2.5, we replace $V = \infty$ and $W = V$ to get

$$d(\infty + nU, V) = d(nU, V) = d(\infty, V)e_n(nU, nV).$$

Thus, we get by equation (31)

$$\frac{c(S,T)}{c(T,S)} = \frac{d(V,\infty)d(\infty,V)e_n(nU,nV)}{d(V,\infty)d(\infty,V)} = e_n(nU,nV) = e_n(S,T).$$

□

Finally, we can prove that both definitions are equivalent:

Theorem 2.6
The definition of the Weil Pairing defined in 2.3 is equivalent to the definition of the faster Weil Pairing defined in 2.4.

Proof.

The definition of c gives us

$$e_n(S,T) = \frac{c(S,T)}{c(T,S)} = \frac{f_{S+T}(X)f_T(X)f_S(X-T)}{f_S(X)f_T(X-S)f_{T+S}(X)} = \frac{f_T(X)f_S(X-T)}{f_S(X)f_T(X-S)}, \quad (32)$$

which is again independent of X. Let's define

$$D'_S = [S] - [\infty], \qquad D'_T = [X_0] - [X_0 - T],$$

where X_0 is choosen such that D'_S and D'_T are disjoint divisors. Define

$$F'_S(X) = f_S(X), \qquad F'_T(X) = \frac{1}{f_T(X_0 - X)}.$$

Then we get that,

$$\operatorname{div}(F'_S) = \operatorname{div}(f_S(X) = n[S] - n[\infty] = nD'_S$$

and

$$\operatorname{div}(F'_T) = -\operatorname{div}(f_T(X_0 - X)) = -(n[X_0 - T] - n[X_0]) = n[X_0] - n[X_0 - T] = nD'_T.$$

By the equation (32) we get finally

$$e_n(S,T) = \frac{F'_T(D'_S)}{F'_S(D'_T)},$$

because the condition of theorem 2.4 are fulfilled and thus it can be applied. Consequently, the theorem is true for the choice of the divisors D'_S and D'_T, which were not chosen arbitrary. We need to treat the case, when we consider arbitrary choices. Let D_S be any divisor of degree 0 such that $\operatorname{sum}(D_S) = S$ and let D_T be any divisor of degree 0 such that $\operatorname{sum}(D_T) = T$. Then for some functions h_1, h_2 we have,

$$D_S = \operatorname{div}(h_1) + D'_S, \qquad D_T = \operatorname{div}(h_2) + D'_T$$

Define

$$F_S = h_1^n F'_S, \qquad F_T = h_2^n F'_T.$$

Then,

$$nD_S = \operatorname{div}(F_S), \qquad nD_T = \operatorname{div}(F_T) \qquad \operatorname{div}(F'_S) = nD'_S \qquad \operatorname{div}(F'_T) = nD'_T$$

First, we assume that the divisors D'_S and D_S are disjoint from D'_T and D_T. Then, we get

$$\frac{F_T(D_S)}{F_S(D_T)} = \frac{h_2(D_S)^n F'_T(D_S)}{h_1(D_T)^n F'_S(D_T)} = \frac{h_2(\operatorname{div}(h_1))^n h_2(D'_S)^n F'_T(\operatorname{div}(h_1)) F'_T(D'_S)}{h_1(\operatorname{div}(h_2))^n h_1(D'_T)^n F'_S(\operatorname{div}(h_2)) F'_S(D'_T)}$$

The Weil reciprocity (Lemma 2.2) implies that $h_2(\mathrm{div}(h_1)) = h_1(\mathrm{div}(h_2))$. Furthermore, we get

$$h_2(D'_S)^n = h_2(nD'_S) = h_2(\mathrm{div}(F'_S)) = F'_S(\mathrm{div}(h_2)).$$

Similarly,

$$h_1(D'_T)^n = F'_T(\mathrm{div}(h_1)),$$

which implies that

$$\frac{F_T(D_S)}{F_S(D_T)} = \frac{h_2(\mathrm{div}(h_1))^n F'_S(\mathrm{div}(h_2)) F'_T(\mathrm{div}(h_1)) F'_T(D'_S)}{h_1(\mathrm{div}(h_2))^n F'_T(\mathrm{div}(h_1)) F'_S(\mathrm{div}(h_2)) F'_S(D'_T)} = \frac{h_2(\mathrm{div}(h_1))^n F'_T(D'_S)}{h_2(\mathrm{div}(h_1))^n F'_S(D'_T)}$$

and we finally get

$$\frac{F_T(D_S)}{F_S(D_T)} = \frac{F'_T(D'_S)}{F'_T(D'_S)} = e_n(S,T).$$

If D'_S and D_S are not disjoint from D'_T and D_T, one can proceed in two steps. First,

$$D''_S = [X_1 + S] - [X_1], \qquad D''_T = [Y_1 + T] - [Y_1],$$

where X_1 and Y_1 are chosen so that D'_S and D''_S are disjoint from D'_T and D''_T and so that D''_S and D_S are disjoint from D''_T and D_T. After that, the preceding argument show that,

$$\frac{F_T(D_S)}{F_S(D_T)} = \frac{F''_T(D''_S)}{F''_S(D''_T)} = \frac{F'_T(D'_S)}{F'_S(D'_T)} = e_n(S,T).$$

□

3 Elliptic curve cryptography

3.1 Discrete logarithm problem

For a general introduction, let **p** be a prime number and let **a** and **b** be integers that are nonzero $\pmod p$. Suppose we know that there exists an integer **k**, without knowing its exact value. Then, the **classical discrete logarithm problem** is to find k, fulfilling the following condition:

$$a^k \equiv b \pmod p. \tag{33}$$

In our case, we are interested in the discrete logarithm problem over an elliptic curve $E(\mathbb{F}_q)$, which has a group law. Let a and b be two points of $E(\mathbb{F}_q)$, then we are trying to find k such that

$$ka = b. \tag{34}$$

The security of cryptosystems using the discrete logarithm problem depends on how difficult it is to solve the used discrete logarithm problem. In general, one uses very large prime numbers to define a finite field. Thus, a brute force attack trying to solve the discrete logarithm problem is very impractical as the answer might be an integer of several hundred digits.

3.2 Attack using the Weil Pairing

In this section I am going to present the **MOV attack**, which transforms a discrete logarithm problem in $E(\mathbb{F}_q)$ to one in $\mathbb{F}_{q^m}^{\times}$, which can be computed much faster (under the condition that $\mathbb{F}_{q^m}$ is not much larger than $\mathbb{F}_q$).

Let E be an elliptic curve over the field $\mathbb{F}_q$ and let $P, Q \in E(\mathbb{F}_q)$. Let N be the order of P (i.e. $NP = \infty$). Assume that

$$\gcd(N, q) = 1.$$

We want to find k such that $Q = kP$. First, we need to check, that a such k exists.

Lemma 3.1
Let N be the order of P and $gcd(N, q) = 1$. There exists k such that $Q = kP$ if and only if $NQ = \infty$ and the Weil Pairing $e_N(P, Q) = 1$.

Proof.

$\Rightarrow$ If $Q = kP$, this implies that $NQ = kNP = k\infty = \infty$ (because N is the order of P). We also know that,

$$e_N(P, Q) = e_N(P, kP) = e_N(P, P)^k = 1^k = 1,$$

by theorem 2.3.

$\Leftarrow$ If $NQ = \infty$, then N is the order of Q and thus $Q \in E[n]$. As we know that $gcd(N, q) = 1$, we know that N does not divide q. Thus, by theorem 2.1, we have that $E[n] \simeq \mathbb{Z}/N\mathbb{Z} \oplus \mathbb{Z}/N\mathbb{Z}$. Now, we choose a point R such that $\{P, R\}$ forms a basis of $E[N]$. Thus, there are some integers a and b such that we can define Q as follows:

$$Q = aP + bR.$$

Consequently, by lemma 2.3 $e_N(P, R) = \xi$ is a primitive Nth root of unity. If $e_N(P, Q) = 1$, we get

$$1 = e_N(P, Q) = e_N(P, aP + bR) = e_N(P, aP)e_N(P, bR)$$

Thus, we get

$$1 = e_N(P,P)^a e_N(P,R)^b = 1\xi^b = \xi^b.$$

This implies that b is a divider of N, which is the order of P. Thus, $b \equiv 0 \pmod{N}$ and $bR = \infty$ (because $R \in E[N]$). Therefore, we get

$$Q = aP + bR = aP + \infty = aP,$$

which is, what we were looking for. □

Theorem 3.1
Choose m such that

$$E[N] \subseteq E(\mathbb{F}_{q^m}).$$

Since all the points of $E[N]$ have coordinates in $\overline{\mathbb{F}_q} = \bigcup_{j\geq 1} \mathbb{F}_{q^j}$, such an m exists. The group μ_N of Nth roots of unity is contained in $\mathbb{F}_{q^m}$. All the calculations will be done in $\mathbb{F}_{q^m}$. The MOV attack algorithm is described by the following steps:

1. *Choose a random point $T \in E(\mathbb{F}_{q^m})$.*
2. *Compute the order M of T.*
3. *Let $d = gcd(N, M)$ and let $T_1 = \frac{M}{d}T$. Then T_1 has order d, which divides N, so $T_1 \in E[N]$.*
4. *Compute $\zeta_1 = e_N(P, T_1)$ and $\zeta_2 = e_N(Q, T_1)$. Then both ζ_1 and ζ_2 are in $\mu_d \subseteq \mathbb{F}_{q^m}^{\times}$.*
5. *Solve the discrete logarithm problem $\zeta_2 = \zeta_1^k$ in $\mathbb{F}_{q^m}^{\times}$. This will give $k \pmod{d}$.*
6. *Repeat with random points in T until the least common multiple of the various $d's$ obtained is N. This determines $k \pmod{N}$*

This algorithm and therefore the MOV attack might not be very useful for some cases, because m could still be very large. However, for **supersingular** elliptic curves, i.e. elliptic curves where

$$a \equiv 0 \pmod{p}$$

one can usually take $m = 2$, which reduces the problem significantly. This implies a much easier computation of the discrete logarithm problem as for arbitrary elliptic curves. In general, two possible cases can occur for **supersingular** elliptic curves:

1. If $a = 0$, then the discrete logarithm problem over $\mathbb{F}_q$ can be reduced to a calculation in $\mathbb{F}_{q^2}^{\times}$.

2. If $a \neq 0$, then one could usually take $m = 3, 4$ or 6 and still speed up the computation significantly.

3.3 A cryptosystem based on the Weil Pairing

Elliptic curves are used in cryptographic applications, because they provide security equivalent to classical systems, while using fewer bits. This can be very interesting for different occasions, because the computation can be much faster whereas it can ensure the same security level. One typical usage is for example on the chips of a passport, where the storage capacity is limited.

The method I am going to present is due to Boneh and Franklin. I presented in the previous section that the Weil Pairing can be used to reduce the discrete logarithm problem on elliptic curves to a discrete logarithm problem for the multiplicative group of a finite field. This method will use the Weil Pairing on these curves (other pairing could also be used). At a first glance, this might sound paradox, but quicker computation will be essential for the algorithm I will present. Additionally, as a reminder: the reduced discrete logarithm problem is still not trivial, if the field is large enough.

For simplicity, I will use the following elliptic curve E to present the algorithm:

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

over the field $\mathbb{F}_p$, where $p \equiv 2 \pmod 3$.

Let $\omega \in \mathbb{F}_{p^2}$ be a primitive cube root of unity. Define a map

$$\beta : E(\mathbb{F}_{p^2}) \longrightarrow E(\mathbb{F}_{p^2}), \qquad (x, y) \longrightarrow (\omega x, y), \qquad \beta(\infty) = \infty.$$

Suppose P has order n. Then $\beta(P)$ has also order n. Now, let's define the **modified** Weil Pairing:

$$\tilde{e}_n(P_1, P_2) = e_n(P_1, \beta(P_2)),$$

where e_n is the usual Weil Pairing and $P_1, P_2 \in E[n]$.

Lemma 3.2
Let $P \in E(\mathbb{F}_p)$ and let n be the order of P. If $3 \nmid n$, then $\tilde{e}_n(P, P)$ is a primitive nth root of unity.

Proof.

Let a and b be integers such that $aP = b\beta(P)$. Then

$$\beta(bP) = b\beta(P) = aP \in E(\mathbb{F}_p),$$

because $P \in E(\mathbb{F}_p)$ and E a group.

1. If $bP = \infty$, then $aP = \infty \Rightarrow a \equiv 0 \pmod{n}$, because n is the order of P. Thus, a = n or a divider of n.

2. If $bP \neq \infty$, then $bP = (x, y)$, with $x, y \in \mathbb{F}_p$. Then,

$$(\omega x, y) = \beta(bP) \in \mathbb{F}_p,$$

by definition of β. But $\omega \notin \mathbb{F}_p$,because the order of $\mathbb{F}_p^\times$ is $p - 1$, which is not a multiple of 3 (Remember: ω is a primitive cube root of unity). This implies that $x = 0$, because $(\omega x, y) \in \mathbb{F}_p$. Therefore, $bP = (0, \pm 1)$, which has order 3 (because we are working over an elliptic curve, thus $3P = \infty \iff 2P = -P$). That is impossible, because we assumed that $3 \nmid n$. Thus, the only possible relation of the form $aP = b\beta(P)$ is that $a \equiv 0 \pmod{n}$ and $b \equiv 0 \pmod{n}$. This implies that P and $\beta(P)$ form a basis of $E[n]$. Finally, by lemma 2.3 we get that $\tilde{e}_n(P, P) = e_n(P, \beta(P))$ is a primitive nth root of unity.

□

Since E is supersingular, by Proposition 2.1, $E(\mathbb{F}_p)$ has order $p + 1$. In addition, one needs to assume that $p = 6l - 1$ for some prime l. Then $6P$ has order l or 1 for each $P \in E(\mathbb{F}_p)$.

For the following scenario, we pretend that Alice wants to send a message to Bob. Description of the situation:

1. Each user (Alice and Bob) has a public key, based on her or his identity (such as an email adresse).

2. A central trusted authority assigns a corresponding private key to each user.

3. The authentication happens in the initial communication between Bob and the trusted authority. After that, Bob is the only one who has the information, which are necessary to decrypt messages that are encrypted using his public identity.

The system described as the following gives the basic idea, but is not secure against certain attacks. There are, however, methods to strengthen the system, which will not be discussed in this thesis. The trusted authority needs to do the following things:

1. Chooses a large prime $p = 6l - 1$ as above.

2. Chooses a point P of order l in $E(\mathbb{F}_p)$.

3. Chooses hash functions H_1 and H_2. The function H_1 takes a string of bits of arbitrary length and ouputs a point of order l on E. The function H_2 inputs an element of order l in $\mathbb{F}_{p^2}^{\times}$ and outputs a binary string of length n, where n is the length of the messages that will be sent.

4. Chooses a secret random $s \in \mathbb{F}_l^{\times}$ and computes $P_{pub} = sP$.

5. Makes $p, H_1, H_2, n, P, P_{pub}$ public, while keeping s secret.

If a user with identity ID wants a private key, the trusted authority does the following:

1. Computes $Q_{ID} = H_1(ID)$. This is a point on E.

2. Lets $D_{ID} = sQ_{ID}$.

3. After verifying that ID is the identification for the user with whom he is communicating, send D_{ID} to his user.

If Alice wants to send message M to Bob, she does the following:

1. Looks up Bob's identity, for example, $ID = bob@uni.lu$ (written as a binary string) and computes $Q_{ID} = H_1(ID)$.

2. Chooses a random $r \in \mathbb{F}_l^{\times}$.

3. Computes $g_{ID} = \tilde{e}_l(Q_{ID}, P_{pub})$.

4. Lets the ciphertext be the pair

$$c = (rP, M \oplus H_2(g_{ID}^r)),$$

where $\oplus$ denotes XOR (=bitwise addition $\pmod 2$).

Bob decrypts a ciphtertext (u, v) as follows:

1. Uses his private key D_{ID} to compute $h_{ID} = \tilde{e}_l(D_{ID}, u)$.

2. Computes $m = v \oplus H_2(h_{ID})$.

The decryption works because

$$\tilde{e}_l(D_{ID}, u) = \tilde{e}_l(sQ_{ID}, rP) = \tilde{e}_l(Q_{ID}, P)^{sr} = \tilde{e}_l(Q_{ID}, sP)^r = \tilde{e}_l(Q_{ID}, P_{pub})^r = g_{ID}^r.$$

Therefore,

$$m = v \oplus H_2(\tilde{e}_l(D_{ID}, u)) = (M \oplus H_2(g_{ID}^r)) \oplus H_2(g_{ID}^r) = M.$$

Alice	Bob	Public	Trusted authority
$Q_{ID} = H_1(ID)$	ID	p, l	$p = 6l - 1$ =prime
$r \in \mathbb{F}_l^\times$	D_{ID}	H_1, H_2	P order l in $E(\mathbb{F}_p)$
$g_{ID} = \tilde{e}_l(Q_{ID}, P_{pub})$		n, P	$s \in \mathbb{F}_p$
$c = (rP, M \oplus H_2(g_{ID}^r))$		P_{pub}	$Q_{ID} = H_1(ID) \in E$
$c = (u, v)$		ID	$D_{ID} = sQ_{ID}$
			$P_{pub} = sP$

3.4 Security of the scheme

First, an overview to show you, who has gotten which information.

The crucial information, which are only known by the proper person, are colorized in red. Important information, which can be intercepted, (not necessarily difficult to get) are colorized in blue. As shown before, in order to decrypt the message, one needs to compute

$$v \oplus H_2(h_{ID}) = (M \oplus H_2(g_{ID}^r)) \oplus H_2(g_{ID}^r)$$

$M \oplus H_2(g_{ID}^r) = v$ is known or can easily be intercepted. So, you need to get the value of g_{ID}^r for being able to decrypt the message. Bob can decrypt it, because

$$\tilde{e}_l(D_{ID}, u) = g_{ID}^r = \tilde{e}_l(Q_{ID}, P)^{sr}$$

Here is, what the scheme makes secure. Eve knows (or can easily compute) the values for Q_{ID} and P. However, she should not know r and s, which are necessary do decrypt the message. This could be interpreted as being equivalent to the prime factorization problem of the RSA scheme. In our case, this problematic is called the **Bilinear Diffie-Hellman Problem (BDH)**. Consequently, the security of this scheme depends on the hardness of solving the BDH problem. In this case, the BDH problem is generalized by

Theorem 3.2
The Bilinear Diffie-Hellman Problem (BDH) in $(\mathbb{F}_p, \mu_n, \tilde{e}_l)$, where P is a generator of $\mathbb{F}_p$ and n is the order of $\mathbb{F}_p$, is defined by: given (P, aP, bP, cP) for some $a, b, c \in \mathbb{F}_l^\times$, compute $v \in \mu_n$ such that $v = \tilde{e}_l(P, P)^{abc}$.

Proof. The proof of the security of the BDH problem is based on the intractability of the problem itself, which will not be given in this thesis. □

Eve knows the values for $sP = P_{pub}$ and $rP = u$, but because Eve does not know the specific value for s and r, she will not be able to compute the value for

$$\tilde{e}_l(sQ_{ID}, rP) = \tilde{e}_l(Q_{ID}, P)^{sr} = \tilde{e}_l(Q_{ID}, sP)^r,$$

which would eventually lead to the decryption of the message.

3.5 How to use the Boneh-Franklin python implementation

In this section I am going to show how to use my python implementation for encrypting and decrypting messages. As mentioned earlier, the Boneh-Franklin scheme is not chosen ciphertext secure. There is a universal transformation method due to Fujisaki and Okamoto that allows for conversion to a scheme having this property. My implementation is very basic in order to show easily how one can establish such a scheme. For a real world application, one would need to split the scripts, extend and adapt them to the needed environment.

One needs to start with the boneh-chiff.py script, which is going to fix the parameters and which will encrypt the message. It is necessary to fix a prime number, to create the finite fields and to choose a third root of unity.

```
#l is a prime number such that 6*l-1 is also a prime number. This
    is important, otherwise computations will not work.
#in this case, 56453 is a suitable prime number (but way too small
    for secure cryptographic applications)
l = int(56453)
#p is the prime number, over which we will work.
p = int(int(6) * l - int(1))

#define the two fields for later purposes
#important to give the irreducible polynomial, otherwise a random
    irreducible polynomial is created.
#thus, random outcomes will occur, which is not usable.
#create the field F_p
Fp = FiniteField(p,1)
#create the Field F_{p^2}, with the ideal generator 1+x^2+x^3 with
    coordinates in F_p
Fp2 = FiniteField(p,2,
    Polynomial([ModP(1,p),ModP(1,p),ModP(1,p)],p))

#define ONE third root of unity of Fp^2
#in this case, it is the polynomial 0+x
b = Fp2([0,1])
```

Next, an algorithm will choose a random point on the elliptic curve, which will be used as a base point. If necessary, one could skip this and fix the point on your own.

```
#C is the elliptic curve, which is created by the script.
```

```
P = findPoint2(C,l,p)
print(P)
#the point P needs to have order l. Thus, l*P = infinity
print(l*P)
```

Continuying in the script, it will be necessary to fix some more elements.

```
#s and r are both in F_l^x
#one is chosen by the authority, the other one is chosen by the
    person who wants to encrypt his message.
#for simplicity and for purpose of illustration, both are chosen
    by myself here
s = int(13)
r = int(7)
```

If you run the script, you will have to enter an ID. This ID will then generate a point, which is called DID. The coordinates of this point are important for the decryption. Thus, one needs to pass them to the person who wants to decrypt the message.

After that, the script will ask you to enter your message, which you want to encrypt.

Once finished, the script will have created a textfile, which includes all necessary information. To enable your partner to decrypt the message, he will need the coordinates of the point DID and the very last line of the textfile. This is your decrypted message.

If you have changed some parameters in the file boneh-chiff.py, you will also need to change some parameters in the file boneh-dechiff.py

```
# the same prime number l as chosen in the other script
l = 56453
# the same prime number p, as generated in the other script
p = int(6 * l - 1)
# you need to create the same field F_{p^2}
# this is the reason, why it is necessary to fix an irreducible
    polynomial which will be the ideal generator
Fp2 = FiniteField(p,2,
    Polynomial([ModP(1,p),ModP(1,p),ModP(1,p)],p))
# this one is not necessary to update. it just generates the
    elliptic curve
E2 = EllipticCurve2( Fp2([0]), Fp2([1]), Fp2)
# this one is the third root of unity chosen in the first script
b = Fp2([0,1])
# these coordinates depend on the base point generated in the
```

```
   previous script, as well as on the chosen number r
cypherACordX = 240099
cypherACordY = 283222
```

Once the parameters are fixed and adapted to the choices made in the first script, you can run the script. You will be asked to enter the coordinates of the point DID. After that, you will be asked to enter the encrypted message. The script should then decrypt the message and print out the original message. To pass the parameters to the third party, one can usually use one of the existing key exchange protocols available (for example, the Diffie-Hellman key exchange).

References

[1] Paulo S. L. M. Barreto and Michael Naehrig. Pairing-friendly elliptic curves of prime order.

[2] Dan Boneh and Matthew Franklin. Identity-based encryption from the weil pairing.

[3] Darrel Hankerson, Alfred Menezes, and Scott Vanstone. *Guide to Elliptic Curve Cryptography*. Springer Professional Computing. Springer, 2003.

[4] Myungsun Kim and Kwangjo Kim. A new identification scheme based on the bilinear diffie-hellman problem.

[5] Luther Martin. *Introduction to Identity-based Encryption*. Artech House information security and privacy series. Artech House, 2008.

[6] L.C. Washington. *Elliptic curves: number theory and cryptography*. Discrete Mathematics and Its Applications Series. Chapman & Hall/CRC, 2008.

A Python implementations

There are several test scripts available to show how to use the following scripts. I invite the reader to analyze carefully my implementations before using them for some real life applications. Also notice, that I implemented the scripts in python3.

My implementations are largely inspired, but adapted to my own purposes, by the very detailed and informative website:

http://jeremykun.com/2014/02/08/introducing-elliptic-curves/

The Weil Pairing is identical to the Sage implementation. Once installed, you can find it in the subdirectory: sage\src\sage\schemes\elliptic_curves\ell_point.py

The Boneh-Franklin scheme is completely developed by my own.

A.1 Modular arithmetic - modular.py

```
#creating a class for being able to represent finite fields
# for instance: 7 (mod 11)
class ModP (object):
   def __init__(self, n, p):
      #prime number p for (mod p)
      self.p = int(p)
      #class of n in (mod p)
      self.n = int(int(n) % p)
      #name of the field
      self.name = "Z/%dZ" % p

   #method for adding two elements of the same field
   def __add__(self, other):
      if isinstance(other, int):
         return ModP(self.n + other, self.p)
      if other.p == self.p:
         return ModP(self.n + other.n, self.p)
      else:
         raise Exception("Different fields")

   def __radd__(self, other):
      if isinstance(other, int):
         return self + other
      if other.p == self.p:
         return ModP(self.n + other.n, self.p)
      else:
```

```
            raise Exception("Different fields")

    #method for subtracting two elements of the same field
    def __sub__(self, other):
        if isinstance(other, int):
            return ModP(self.n - other, self.p)
        if other.p == self.p:
            return ModP(self.n - other.n, self.p)
        else:
            raise Exception("Different fields")

    #method for multiplying two elements of the same field
    def __mul__(self, other):
        if isinstance(other, int):
            other = ModP(other, self.p)
        if other.p == self.p:
            return ModP(self.n * other.n, self.p)
        else:
            raise Exception("Different fields")

    def __rmul__(self, other):
        return self * other

    #method for dividing two elements of the same field
    def __truediv__(self, other):
        if isinstance(other, int):
            other = ModP(other, self.p)
        if other.p == self.p:
            return self * other.inverse()
        else:
            raise Exception("Different fields")

    #method for dividing two elements of the same field
    def __div__(self, other):
        if isinstance(other, int):
            other = ModP(other, self.p)
        if other.p == self.p:
            return self * other.inverse()
        else:
            raise Exception("Different fields")

    #method for getting the inverse of an element
    def __neg__(self):
```

```
        return ModP(-self.n, self.p)

    #check if two elements are the same
    def __eq__(self, other):
        if isinstance(other, int):
            other = ModP(other,self.p)
        return other.p == self.p and self.n == other.n

    #absolute value of an element
    def __abs__(self):
        return abs(self.n)

    #string representation
    def __str__(self):
        return "%d (mod %d)" % (self.n, self.p)

    #usual representation
    def __repr__(self):
        return "%d (mod %d)" % (self.n, self.p)

    #multiplicative inverse of an element is calculated by Euclidean
        Algorithm
    def inverse(self):
        g, x, y = EuclideanAlgo(self.n, self.p)
        if g != 1:
            raise ValueError
        return ModP(x, self.p)

#Euclidean Algorithm for computing the inverse of an element
#http://en.wikipedia.org/wiki/Extended_Euclidean_algorithm
def EuclideanAlgo(a,b):
    lastremainder, remainder = abs(a), abs(b)
    x, lastx, y, lasty = 0, 1, 1, 0
    while remainder:
        lastremainder, (quotient, remainder) = remainder,
            divmod(lastremainder, remainder)
        x, lastx = lastx - quotient*x, x
        y, lasty = lasty - quotient*y, y
    return lastremainder, lastx * (-1 if a < 0 else 1), lasty * (-1
        if b < 0 else 1)
```

A.2 Polynomials - polynomial.py

```
import fractions
import itertools
import modular
import random

ModP = modular.ModP

#strip all copies of eraseValue of the end of the list
#i.e. erase all 0 at the end (in our case)
def strip(L, eraseValue):
   if len(L) == 0:
      return L

   i = len(L)-1

   while i>=0 and L[i] == eraseValue:
      i = i-1

   return L[:i+1]


#define polynomials under the form of:
#a + b*x + c*x^2 + ...
class Polynomial(object):
   def __init__(self, c, p):
      if type(c) is Polynomial:
         self.coefficients = c.coefficients
      elif isinstance(c, ModP):
         self.coefficients = [c]
      elif not hasattr(c, '__iter__') and not hasattr(c, 'iter'):
         self.coefficients = [ModP(c,p)]
      else:
         self.coefficients = c

      self.p = p
      self.coefficients = strip(self.coefficients, ModP(0,p))
      self.name = '(Z/%dZ)[x]' % p


   #check if the polynomial is 0
```

```
def isZero(self):
   return self.coefficients == []

#function to print the polynomial
def __repr__(self):
   if self.isZero():
      return '0'
   #iterate through the list of coefficients and add them to one
       string
   else:
      return ' + '.join(['%s x^%d' % (a,i) if i>0 else '%s' % a
          for i,a in enumerate(self.coefficients)])

#length of the polynomial
def __abs__(self):
   return len(self.coefficients)

#length of the polynomial
def __len__(self):
   return len(self.coefficients)

#subtract to polynomials by subtracting their coeff.
def __sub__(self, other):
   return self + (-other)

def __rsub__(self, other):
   return -self + other

#iterate through the coefficients
def __iter__(self):
   return iter(self.coefficients)

#negative of a polynomial
def __neg__(self):
   return Polynomial([-a for a in self],self.p)

#iterate through polynomial
def iter(self):
   return self.__iter__()

#the leading coefficient of a polynomial
def leadingCoefficient(self):
   return self.coefficients[-1]
```

```
#the degree of a polynomial, ie largest exponent
def degree(self):
   return abs(self)-1

#check whether two polynomials are equal or not by comparing
    coefficients and same degree
def __eq__(self,other):
   return self.degree() == other.degree() and all([x==y for
       (x,y) in zip (self,other)])

#add two polynomials by adding their coefficients
def __add__(self,other):
   #if integer, than one needs to make a constant polynomial
   if isinstance(other, int):
      other = Polynomial([other],self.p)
   #adding the coefficients together. fillvalue defines the
       value to use if one polynomial
   #has a smaller degree than the other one.
   newCoefficients = [sum(x) for x in
       itertools.zip_longest(self,other, fillvalue =
       ModP(0,self.p))]
   return Polynomial(newCoefficients, self.p)

def __radd__(self, other):
   return self + other

#multiplication of two polynomials
def __mul__(self,other):
   if isinstance(other, int):
      return self*Polynomial([other],self.p)
   if self.isZero() or other.isZero():
      return Zero(self.p)
   else:
      #set all coefficients to zero
      newCoefficients = [ModP(0,self.p) for _ in range(len(self)
          + len(other) - 1)]

      #general formula for the coefficients of the
          multiplication of two poly.
      for i,a in enumerate(self):
         for j,b in enumerate(other):
            newCoefficients[i+j] = newCoefficients[i+j] + a*b
```

```
        return Polynomial(newCoefficients,self.p)

    def __rmul__(self, other):
        return self * other

    #divmod for polynomials
    def __divmod__(self,divisor):
        quotient = Zero(self.p)
        remainder = self
        divisorDeg = divisor.degree()
        divisorLC = divisor.leadingCoefficient()

        while remainder.degree() >= divisorDeg:
            StockExponent = remainder.degree() - divisorDeg
            StockZero = [ModP(0,self.p) for _ in range(StockExponent)]
            StockDivisor = Polynomial(StockZero +
                [remainder.leadingCoefficient() / divisorLC], self.p)

            quotient = quotient + StockDivisor
            remainder = remainder - (StockDivisor * divisor)

        return quotient, remainder

    #modular function for polynomials
    def __mod__(self, divisor):
        x,y = divmod(self, divisor)
        return y

    def __pow__(self, p):
        x = self
        r = Polynomial(1,self.p)
        while p != 0:
            if p % 2 == 1:
                r = r * x
                p = p - 1

            x = x * x
            p = p / 2
        return r

    #polynomial to the power p modulo other
    def powmod(self, p, other):
```

```
      x,y = divmod(self**p, other)
      return y

   #usual division
   def __truediv__(self, divisor):
      if divisor.isZero():
         raise ZeroDivisionError
      x,y = divmod(self, divisor)
      return x

   #usual division
   def __div__(self, other):
      return self.__truediv__(other)



#returns a Zero polynomial
def Zero(p):
      return Polynomial([],p)

#check whether a polynomial is irreducible or not
def isIrreducible(polynomial, p):
   #polynomial "x"
   x = Polynomial([ModP(0,p), ModP(1,p)],p)
   powerTerm = x
   isUnit = lambda p: p.degree() == 0;

   for _ in range( int(polynomial.degree() / 2)):
      powerTerm = powerTerm.powmod(p, polynomial)
      gcdOverZmodp = gcd(polynomial, powerTerm - x)
      if not isUnit(gcdOverZmodp):
         return False

   return True

#greatest common divisor
def gcd(a,b):
   if abs(a) < abs(b):
      return gcd(b,a)

   while abs(b) > 0:
      q,r = divmod(a,b)
      a,b = b,r
```

```
    return a

#returns an irreducible polynomial
def generateIrrduciblePolynomial(p, degree):

    while True:
        coefficients = [ModP(random.randint(0, p-1),p) for _ in
            range(degree)]
        randomMonicPolynomial = Polynomial(coefficients +
            [ModP(1,p)],p)

        if isIrreducible(randomMonicPolynomial, p):
            return randomMonicPolynomial


def extentedEuclideanAlgorithm(a, b):
    if abs(b) > abs(a):
        (x,y,d) = extentedEuclideanAlgorithm(b,a)
        return (y,x,d)

    if abs(b) == 0:
        return (1,0,a)

    x1, x2, y1, y2 = 0,1,1,0
    while abs(b) > 0:
        q, r = divmod(a,b)
        x = x2 - q*x1
        y = y2 - q*y1
        a, b, x2, x1, y2, y1 = b, r, x1, x, y1, y

    return (x2, y2, a)
```

A.3 Finite fields - finiteField.py

```
import fractions
import itertools
import modular
import random
import polynomial

generateIrrduciblePolynomial =
    polynomial.generateIrrduciblePolynomial
ModP = modular.ModP
Polynomial = polynomial.Polynomial

#one can fix a polynomialModulus if you define a finite field
#however, it needs to be irreducible
def FiniteField(p, m, polynomialModulus=None):

   #generates a random irreducible polynomial
   #not very good for Test purposes! because always
   #new examples will appear randomly
   if polynomialModulus is None:
      polynomialModulus = generateIrrduciblePolynomial(p, m)

   class Fq(object):
      fieldsize = int(p**m)
      primeSubfield = p
      idealGenerator = polynomialModulus

      def __init__(self, poly):
         if type(poly) is Fq:
            self.poly = poly.poly
         elif type(poly) is int:
            self.poly = Polynomial([ModP(poly,p)],p)
         elif isinstance(poly, ModP):
            self.poly = Polynomial([ModP(poly.n,p)],p)
         elif isinstance(poly, Polynomial):
            self.poly = poly % polynomialModulus
         else:
            self.poly = Polynomial([ModP(x,p) for x in poly],p) %
                polynomialModulus

         self.field = Fq
```

```
def __add__(self, other):
   return Fq(self.poly + other.poly)

def __sub__(self, other):
   return Fq(self.poly - other.poly)

def __mul__(self, other):
   return Fq(self.poly * other.poly)

def __eq__(self, other):
   return isinstance(other, Fq) and self.poly == other.poly

#fast polynomial multiplication
def __pow__(self,n):
   x = self
   r = Fq([1])
   while n != 0:
      if n % 2 == 1:
         r = r * x
         n = n - 1

      x = x * x
      n = n / 2

   return Fq(r.poly)

def __neg__(self):
   return Fq(-self.poly)

def __abs__(self):
   return abs(self.poly)

def __repr__(self):
   return repr(self.poly) + ' over ' + self.__class__.__name__

def __divmod(self, divisor):
   q, r = divmod(self.poly , divisor.poly)
   return (Fq(q), Fq(r))

#inverse of an element
def inverse(self):
   if self == Fq(0):
```

```
            raise ZeroDivisionError

        x,y,d = extentedEuclideanAlgorithm(self.poly,
            self.idealGenerator)
        return Fq(x) * Fq(d.coefficients[0].inverse())

    #dividing
    def __div__(self,other):
        return self * other.inverse()

    #dividing
    def __truediv__(self, other):
        return self * other.inverse()

    #dividing
    def __rdiv__(self,other):
        return self * other.inverse()
    #dividing
    def __rtruediv__(self, other):
        return self * other.inverse()

Fq.__name__ = 'F_{%d^%d}' % (p,m)
return Fq


def extentedEuclideanAlgorithm(a, b):
    if abs(b) > abs(a):
        (x,y,d) = extentedEuclideanAlgorithm(b,a)
        return (y,x,d)

    if abs(b) == 0:
        return (1,0,a)

    x1, x2, y1, y2 = 0,1,1,0
    while abs(b) > 0:
        q, r = divmod(a,b)
        x = x2 - q*x1
        y = y2 - q*y1
        a, b, x2, x1, y2, y1 = b, r, x1, x, y1, y

    return (x2, y2, a)
```

A.4 Elliptic curves over prime fields - ellipticCurveMod.py

```
import modular

ModP = modular.ModP

#to understand the concept behind these conditions, check the
    documentation
#of elliptic curve in my bachelor thesis
class EllipticCurve(object):
   def __init__(self,a,b):
      #this construction only works for the Weierstrass form
      # y^2 = x^3 + a*x + b
      self.a = a
      self.b = b
      self.p = self.a.p

      #compute the discriminant to check whether there are multiple
          roots or not
      self.discriminant = ModP(4,self.p)*a*a*a+ ModP(27,self.p)*b*b

      #if the curve has multiple roots, then we have to raise an
          exception
      #we do not want to work with curves with multiple roots
      if self.isSingular():
         raise Exception("The curve %s has multiple roots. Bad
             choice!" % self)

   #function to check if there are multiple roots
   def isSingular(self):
      # if the discriminant is zero, the method returns true
      # thus, there are multiple roots
      return self.discriminant == ModP(0,self.p)

   #function to check whether a point is on the curve or not
   def isPoint(self, x,y):
      # enter the coordinates of the point into the equation of the
          curve
      # return true if the point is on the curve
      # int cast necessary to prevent wrong multiplication as:
          1*1*1 = 3 ... (yes, this happened)
```

```
        x1 = int(x.n)
        y1 = int(y.n)
        a1 = int(self.a.n)
        b1 = int(self.b.n)
        return (ModP(y1*y1,x.p)) == (ModP(x1*x1*x1 + a1 * x1 +
            b1,x.p))

    #define the string for a elliptic curve to print it properly
    def __str__(self):
        return "y^2=x^3+ %Gx+ %G" % (self.a.n, self.b.n)

    #check whether two elliptic curves are equal or not
    def __eq__(self, other):
        return (self.a, self.b) == (other.a, other.b)


#class for a point on the elliptic curve (!)
class Point(object):
    #we need to take the curve as an argument, because we
        immediately check
    #if the point is on the curve or not
    def __init__(self, curve, x, y):
        self.curve = curve
        self.x = x
        self.y = y
        self.p = self.x.p

        #if the point is not on the curve, we do not need to create
            the point
        if not self.curve.isPoint(x,y):
            raise Exception("The point %s is not on the curve %s" %
                (self, self.curve))

    #function to output the point
    def __str__(self):
        return "(%s,%s)" % (self.x,self.y)

    #the negative/opposite point P s.t. P-P=infinity
    def __neg__(self):
        return Point(self.curve, self.x, -self.y)

    #we are going to define the addition of two points, check
        documentation
```

```
def __add__(self, P):
   #if no point, the addition does not make sense
   if isinstance(P, Infinity):
      return self

   if (self.x, self.y) == (P.x, P.y) :
      if self.y == ModP(0,self.p):
         return Infinity(self.curve)

      else:
         m = ( ModP(3,self.p) * self.x * self.x + self.curve.a )
             / ( ModP(2,self.p) * self.y )
         p = m * m - ModP(2,self.p) * self.x
         q = m * ( self.x - p ) - self.y
         return Point(self.curve, p ,q)

   else:
      if self.x == P.x:
         return Infinity(self.curve)

      else:
         m = (P.y - self.y) / (P.x - self.x)
         p = m*m - self.x - P.x
         q = m*(self.x - p) - self.y
         return Point(self.curve, p, q)

#method for subtracting
def __sub__(self, P):
   #adding with a negative point
   return self + -P

#adding a point several times to itself
def __mul__(self, n):
   #if not an integer, does not make sense
   if not isinstance(n, int):
      raise Exception("You need to input an integer")

   else:
      #if zero times, it results infinity
      if n == 0:
         return Infinity(self.curve)

      if n == 1:
```

```
            return self

        #if negative integer, adding the negative point n times
        if n < 0:
            return -self * -n

        else:
            #double-and-add algorithm for faster addition
            #not in binary, might change that later
            #http://en.wikipedia.org/wiki/Elliptic_curve_point_multiplication
            P = self
            Q = Infinity(self.curve)

            while n > 0:

                if (n % 2) == 1:
                    Q = P + Q
                    n = n-1
                else:
                    P = P + P
                    n = n / 2
            return Q

    def __rmul__(self, n):
        return self * n

    def __eq__(self, other):
        if isinstance(self, Infinity) and isinstance(other, Infinity):
            return True
        if isinstance(self, Infinity) and not isinstance(other,
            Infinity):
            return False
        if not isinstance(self, Infinity) and isinstance(other,
            Infinity):
            return False
        if self.x == other.x and self.y == other.y and self.p ==
            other.p:
            return True

#general remark: we always have integer < string
class Infinity(Point):
    def __init__(self,curve):
        self.curve = curve
```

```
      #the infinity point is always on the curve

   #maybe I will change the notation to make it look more
       mathematically
   def __str__(self):
      return "Infnity"

   #stays the same, because neutral element
   def __neg__(self):
      return self

   #infinity is neutral element
   def __add__(self, P):
      return P

   #infinity is neutral element
   def __sub__(self,P):
      return P

   #adding infinity n times to itself
   def __mul__(self, n):
      #if not integer, doesnt make sense
      if not isinstance(n, int):
         raise Exception("You need to input an integer")

      else:
         return self


#there is an explanation of the algorithm in my thesis
def MillerFunction(P, R, Q):

   if isinstance(P, Infinity) or isinstance(R, Infinity):
      if P == R:
         return ModP(1, Q.p)
      if isinstance(P, Infinity):
         return Q.x - R.x
      if isinstance(R, Infinity):
         return Q.x - P.x

   else:
      if P != R:
         if P.x == R.x:
```

```
            return Q.x - P.x
         else:
            l = ( R.y - P.y ) / (R.x - P.x)
            return Q.y - P.y - l * (Q.x - P.x)
      else:
         numerator = ModP(3, P.p) * (P.x * P.x) + P.curve.a
         denominator = ModP(2, P.p) * P.y
         if denominator == ModP(0, P.p):
            return Q.x - P.x
         else:
            l = numerator / denominator
            return Q.y - P.y - l * (Q.x - P.x)


def Miller(P, Q, m):
   t = ModP(1,P.p)
   V = P
   S = 2*V
   mylist = list(bin(m)[2:])
   i = 1
   while i < len(mylist):
      S = 2*V
      t = (t*t)*(MillerFunction(V,V,Q) / MillerFunction(S,-S,Q))
      V = S
      if mylist[i] == '1':
         S = V + P
         t = t * (MillerFunction(V,P,Q) / MillerFunction(S,-S,Q))
         V = S
      i = i + 1
   return t


def WeilPairing(P,Q,m):
   if not isinstance(m*P, Infinity) or not isinstance(m*Q,
       Infinity):
      raise Exception("The two points do not have order %d" %m)
   if P == Q:
      return ModP(1,P.p)
   if isinstance(P, Infinity) or isinstance(Q, Infinity):
      return ModP(1, P.p)
   fmPQ = Miller(P,Q,m)
   fmQP = Miller(Q,P,m)
```

```
    if fmQP == ModP(0, P.p):
        return ModP(1, P.p)
    return (ModP((int(-1))**(m),P.p))*(fmPQ / fmQP)
```

A.5 Elliptic curves over all finite fields - ellipticCurve.py

```
import finiteField
import modular

ModP = modular.ModP

#to understand the concept behind these conditions, check the
    documentation
#of elliptic curve in my bachelor thesis
class EllipticCurve(object):
   def __init__(self,a,b, field):
      #this construction only works for the Weierstrass form
      # y^2 = x^3 + a*x + b
      self.a = a
      self.b = b
      self.field = field
      self.p = field.primeSubfield

      #compute the discriminant to check whether there are multiple
          roots or not
      self.discriminant = self.field([4])*a*a*a+
          self.field([27])*b*b

      #if the curve has multiple roots, then we have to raise an
          exception
      #we do not want to work with curves with multiple roots
      if self.isSingular():
         raise Exception("The curve %s has multiple roots. Bad
             choice!" % self)

   #function to check if there are multiple roots
   def isSingular(self):
      # if the discriminant is zero, the method returns true
      # thus, there are multiple roots
      return self.discriminant == self.field([0])

   #function to check whether a point is on the curve or not
   def isPoint(self, x,y):
      # enter the coordinates of the point into the equation of the
          curve
      # return true if the point is on the curve
```

```
        return (y*y - x*x*x - self.a * x - self.field([1]) ==
            self.field([0]))

    #define the string for a elliptic curve to print it properly
    def __str__(self):
        return "y^2=x^3+ %s*x+ %s" % (self.a, self.b)

    #check whether two elliptic curves are equal or not
    def __eq__(self, other):
        return (self.a, self.b) == (other.a, other.b)


#class for a point on the elliptic curve (!)
class Point(object):
    #we need to take the curve as an argument, because we
        immediately check
    #if the point is on the curve or not
    def __init__(self, curve, x, y):
        self.curve = curve
        self.field = curve.field
        self.x = x
        self.y = y

        #if the point is not on the curve, we do not need to create
            the point
        if not self.curve.isPoint(x,y):
            raise Exception("The point %s is not on the curve %s" %
                (self, self.curve))

    #function to output the point
    def __str__(self):
        return "(%s,%s)" % (self.x,self.y)

    #the negative/opposite point P s.t. P-P=infinity
    def __neg__(self):
        return Point(self.curve, self.x, -self.y)

    #we are going to define the addition of two points, check
        documentation
    def __add__(self, P):
        #if no point, the addition does not make sense
        if isinstance(P, Infinity):
            return self
```

```
    if (self.x, self.y) == (P.x, P.y) :
        if self.y == self.field([0]):
            return Infinity(self.curve)

        else:
            m = ( self.field([3]) * self.x * self.x + self.curve.a )
                / ( self.field([2]) * self.y )
            p = m * m - self.field([2]) * self.x
            q = m * ( self.x - p ) - self.y
            return Point(self.curve, p ,q)

    else:
        if self.x == P.x:
            return Infinity(self.curve)

        else:
            m = (P.y - self.y) / (P.x - self.x)
            p = m*m - self.x - P.x
            q = m*(self.x - p) - self.y
            return Point(self.curve, p, q)

#method for subtracting
def __sub__(self, P):
    #adding with a negative point
    return self + -P

#adding a point several times to itself
def __mul__(self, n):
    #if not an integer, does not make sense
    if not isinstance(n, int):
        raise Exception("You need to input an integer")

    else:
        #if zero times, it results infinity
        if n == 0:
            return Infinity(self.curve)

        if n == 1:
            return self

        #if negative integer, adding the negative point n times
        if n < 0:
```

```
            return -self * -n

        else:
            #double-and-add algorithm for faster addition
            #not in binary, might change that later
            #http://en.wikipedia.org/wiki/Elliptic_curve_point_multiplication
            P = self
            Q = Infinity(self.curve)

            while n > 0:

                if (n % 2) == 1:
                    Q = P + Q
                    n = n-1
                else:
                    P = P + P
                    n = n / 2
            return Q

    def __rmul__(self, n):
        return self * n

    def __eq__(self, other):
        if isinstance(self, Infinity) and isinstance(other, Infinity):
            return True
        if isinstance(self, Infinity) and not isinstance(other,
            Infinity):
            return False
        if not isinstance(self, Infinity) and isinstance(other,
            Infinity):
            return False
        if self.x == other.x and self.y == other.y:
            return True

#general remark: we always have integer < string
class Infinity(Point):
    def __init__(self,curve):
        self.curve = curve
        #the infinity point is always on the curve

    #maybe I will change the notation to make it look more
        mathematically
    def __str__(self):
```

```
        return "Infnity"

    #stays the same, because neutral element
    def __neg__(self):
        return self

    #infinity is neutral element
    def __add__(self, P):
        return P

    #infinity is neutral element
    def __sub__(self,P):
        return P

    #adding infinity n times to itself
    def __mul__(self, n):
        #if not integer, doesnt make sense
        if not isinstance(n, int):
            raise Exception("You need to input an integer")

        else:
            return self


def MillerFunction(P, R, Q):
    field = P.curve.field
    if isinstance(P, Infinity) or isinstance(R, Infinity):
        if P == R:
            return field([1])
        if isinstance(P, Infinity):
            return Q.x - R.x
        if isinstance(R, Infinity):
            return Q.x - P.x

    else:
        if P != R:
            if P.x == R.x:
                return Q.x - P.x
            else:
                l = ( R.y - P.y ) / (R.x - P.x)
                return Q.y - P.y - l * (Q.x - P.x)
        else:
            numerator = field([3]) * (P.x * P.x) + P.curve.a
```

```
        denominator = field([2]) * P.y
        if denominator == field([0]):
            return Q.x - P.x
        else:
            l = numerator / denominator
            return Q.y - P.y - l * (Q.x - P.x)

def Miller(P, Q, m):
    field = P.curve.field
    t = field([1])
    V = P
    S = 2*V
    mylist = list(bin(m)[2:])
    i = 1
    while i < len(mylist):
        S = 2*V
        t = (t*t)*(MillerFunction(V,V,Q) / MillerFunction(S,-S,Q))
        V = S
        if mylist[i] == '1':
            S = V + P
            t = t * (MillerFunction(V,P,Q) / MillerFunction(S,-S,Q))
            V = S
        i = i + 1
    return t


def WeilPairing(P,Q,m):
    field = P.curve.field
    if P == Q:
        return field([1])
    if isinstance(P, Infinity) or isinstance(Q, Infinity):
        return field([1])
    fmPQ = Miller(P,Q,m)
    fmQP = Miller(Q,P,m)
    if fmQP == field([0]):
        return field([1])
    return (field([(-1)**(m)]))*(fmPQ / fmQP)


def ModifWeilPairing(P,Q,m,b):
    if isinstance(Q, Infinity):
        return WeilPairing(P,Q,m)
    else:
```

```
    Q = Point(Q.curve, b*Q.x, Q.y)
    return WeilPairing(P,Q,m)
```

A.6 Boneh-Franklin Scheme initialization and encryption - boneh-chiff.py

```
#python modules
import hashlib
import binascii
import os
import random

#general modules written by myself
import modular
import ellipticCurveMod
import ellipticCurve
import finiteField
import polynomial

#if we want to work of fields different than Z/pZ
FiniteField = finiteField.FiniteField
Polynomial = polynomial.Polynomial
ModifWeil = ellipticCurve.ModifWeilPairing
EllipticCurve2=ellipticCurve.EllipticCurve
Point2=ellipticCurve.Point

#if we want to work only over the field Z/pZ
EllipticCurve=ellipticCurveMod.EllipticCurve
Point=ellipticCurveMod.Point
Infinity=ellipticCurveMod.Infinity
ModP = modular.ModP


#class to store the ciphertext
class Ciphertext (object):
   def __init__(self, a, b):
      self.a = a
      self.b = b

   def __str__(self):
      return "%s and %s" % (self.a, self.b)

#find a point on the elliptic curve
#starting with the y coordinate
def findPoint(C,l, p):
```

```
    i = int(3)
    while True:
        #replace y by value an find the x value
        Py = ModP(i,p)
        Px = (Py*Py-ModP(1,p))
        Px = ModP(Px.n**(1/3.0),p)
        #if a point and order correct, return it
        if C.isPoint(Px,Py):
            P = Point(C, Px, Py)
            if isinstance(P*l, Infinity):
                return P
            #6*P is of order 1 or l, check proposition
            elif isinstance(6*P*l, Infinity):
                return 6*P
        i = i + 1
        #if we tried too much possibilities, we stop the program
        if i > 300000:
            raise Exception("No point could be found")

#find a point on the elliptic curve
#starting with the x coordinate
#test have shown that in most cases this one finds faster a point
def findPoint2(C,l, p):
    i = int(3)
    while True:
        #replace x by value and find y
        Px = ModP(i,p)
        Py = (Px*Px*Px+ModP(1,p))
        Py = ModP((Py.n)**(1/2.0),p)
        #if a point and order correct, return it
        if C.isPoint(Px,Py):
            P = Point(C, Px, Py)
            if isinstance(P*l, Infinity):
                return P
            #6*P is of order 1 or l, check proposition
            elif isinstance(6*P*l, Infinity):
                return 6*P
        i = i + 1
        #if tried too much possibilites, try to find a point
        #by starting with y value
        if i > 300000:
            P = findPoint(C,l,p)
            return P
```

```
def hash (ID, C, p, Q, l):
   i = int(0)
   while True:
      #always initialize the hash function, so that both parties
          can find the same hashed point
      hash1 = hashlib.md5()
      hash1.update(ID.encode('utf-8'))
      #get the integer from the hash value and multiply it to the
          base point
      k = int.from_bytes(hash1.digest(), byteorder='big')+i
      P = Q*k
      #if point of order l, return it
      if isinstance(P*l, Infinity) and P!=Q:
         return P
         #6*P is of order 1 or l, check proposition
      elif isinstance(6*P*l, Infinity) and P!=Q:
         return 6*P
      i = i + 1
      #if no point found, try to replace y by a value and find x
      if i > 300000:
         raise Exception("No point could be found")

#second hash function: input an element of order l in Fp^2 and
    outputs a string of length n
#where the length of the message is n
def hash3 (value, lengthMessage):
   sum = 0
   #sum the coefficients
   for i,a in enumerate(value.poly):
      sum = sum + a
   value = sum.n

   length = lengthMessage

   #Knuth's multiplicative method:
   hash = value * 2654435761 % (2**32)
   hash = bin(hash)
   hash = hash + hash[2:] + hash[2:] + hash[2:] + hash[2:] +
       hash[2:]
   hash = hash[:length]

   output = bytearray(hash.encode())
```

```
   return output

#xor function: bitwise addition
def xor (a,b):
   c = bytearray(len(a))
   for i in range(len(a)):
      c[i] = a[i] ^ b[i]
   return c



#open file to save parameters gotten
#through this computation
print("--------------------------------")

outputFile = open('parameters.txt','w')

outputFile.write('If you are going to change the parameters in the
    encrypting file, you need to adapt some parameters in the
    decryption file'+'\n')
outputFile.write('You will always need to adapt the coordinates of
    DID and you need to copy the last output line of this file to
    properly decrypt'+'\n')
outputFile.write('--------------------------------'+'\n')

print("--------------------------------")



print("--------------------------------")
print("Initializing:")

#l=109
#l=127
#l=199
#l=56453

#defining the values for l and p of the scheme
#l = int(127) #working!!!
l = int(56453)
p = int(int(6) * l - int(1))
```

```
#define the two fields for later purpose
#Fp = FiniteField(p,1) #for l=127
#important to give the irreducible polynomial
#Fp2 = FiniteField(p,2,
    Polynomial([ModP(6,p),ModP(758,p),ModP(1,p)],p)) #for l=127

Fp = FiniteField(p,1) #for l=56453
Fp2 = FiniteField(p,2,
    Polynomial([ModP(1,p),ModP(1,p),ModP(1,p)],p)) #for l=56453

#define ONE third root of unity of Fp^2
#b = Fp2([249,341]) #for l = 127
b = Fp2([0,1]) #for l=56453

print("The prime number l is:")
print(l)
outputFile.write('The prime number l is: '+str(l)+'\n')
print("The prime number p is:")
print(p)
outputFile.write('The prime number p is: '+str(p)+'\n')
print("The third root of unity is:")
print(b)
outputFile.write('Third root of unit is: '+str(b)+'\n')

#condition of the scheme to work properly
if (p-2) % 3 != 0:
   raise Exception("p does not verifiy the condition 2 mod 3")

print("--------------------------------")
print("The elliptic curve is:")
C = EllipticCurve(ModP(0,p),ModP(1,p))
print(C)


print("The choosen point of order %d is:" % l)
P = findPoint2(C,l,p)
print(P)
print("Check if the order is correct:")
print(l*P)


ID = input("Enter the ID you want to use: ")
#print("The ID is:")
```

```
#ID = "steve@uni.lu"
print(ID)

#s in F_l^x
print("s is equal to :")
s = int(13)
print(s)
Ppub = s*P

print("Ppub is equal to %s " % Ppub)

print("--------------------------------")
print("The hashed point is:")

QID = hash(ID,C,p,P,l)
print(QID)

DID = s*QID
print("DID is equal to: %s" % DID)
outputFile.write('DID is equal to: '+str(DID)+'\n')

print("--------------------------------")
print("Alice part:")
QIDAlice = hash(ID,C,p, P, l)

print(QIDAlice)
#r in F_l^x
r = int(7)
print("r is equal to:")
print(r)


print("--------------------------------")
print("Test if points are of order")
print(l)
print("Point l*QID Alice")
print(l*QIDAlice)
print("Point l*Ppub")
print(l*Ppub)

print("--------------------------------")
print("Weil pairing and verification")
```

```
#define the points of the Elliptic Curve to the new
#elliptic curve. we have an inclusion
#E(Fp) C E(Fp^2)
#but for further computation, we need to be able to
#work over the new elliptic curve (for the Weil pairing)
E2 = EllipticCurve2( Fp2([0]), Fp2([1]), Fp2)
QIDAlice2 = Point2(E2, Fp2([QIDAlice.x.n]), Fp2([QIDAlice.y.n]))
Ppub2 = Point2(E2, Fp2([Ppub.x.n]), Fp2([Ppub.y.n]))

gID = ModifWeil(QIDAlice2, Ppub2, l , b)

print("gID is equal to:")
print(gID)
print("Check if it is a lth rooth:")
print(gID**(l))


print("--------------------------------")
print("Encryption")

print("\n")
M = input("Enter the message you want to encrypt: ")

#M = " hello, this is a test. are you sure this is working? I
    could easily break your decryption!"
lengthMessage = len(M)

print("The message to encrypt is : %s" % M)

#decode the message to bytes and hash it
b1 = bytearray(M.encode('utf-8'))
hash = hash3(gID**(r), lengthMessage)

#bitwise addition
xor1 = xor(b1,hash)

#create the cyphertext to send it to someone else
cypher = Ciphertext(r*P,xor1)

outputFile.write('First value of the cyphertext:
    '+str(cypher.a)+'\n')
outputFile.write('Second value of the cyphertext:
    '+str(cypher.b)+'\n')
```

```
print("The message after encryption in bytes: ")
print(xor1)

#create a hex representation of the encrypted message. this way,
    it is easier to communicate to a third party
#and independent of the machine which is running.
decoded = binascii.hexlify(xor1)

outputFile.write("This is a hex representation of the encrypted
    message. This hex-code needs to be entered to the decryption
    script: " + str(decoded)[2:len(str(decoded))-1])

#close file
outputFile.close()
```

A.7 Boneh-Franklin Scheme decryption - boneh-dechiff.py

```
#python modules
import hashlib
import binascii
import os
import random
import binascii

#general modules written by myself
import modular
import ellipticCurveMod
import ellipticCurve
import finiteField
import polynomial

#if we want to work of fields different than Z/pZ
FiniteField = finiteField.FiniteField
Polynomial = polynomial.Polynomial
ModifWeil = ellipticCurve.ModifWeilPairing
EllipticCurve2=ellipticCurve.EllipticCurve
Point2=ellipticCurve.Point

#if we want to work only over the field Z/pZ
EllipticCurve=ellipticCurveMod.EllipticCurve
Point=ellipticCurveMod.Point
Infinity=ellipticCurveMod.Infinity
ModP = modular.ModP

#second hash function: input an element of order l in Fp^2 and
    outputs a string of length n
#where the length of the message is n
def hash3 (value, lengthMessage):
   sum = 0
   #sum the coefficients
   for i,a in enumerate(value.poly):
      sum = sum + a
   value = sum.n

   length = lengthMessage

   #Knuth's multiplicative method:
```

```
    hash = value * 2654435761 % (2**32)
    hash = bin(hash)
    hash = hash + hash[2:] + hash[2:] + hash[2:] + hash[2:] +
        hash[2:]
    hash = hash[:length]

    output = bytearray(hash.encode())

    return output

#xor function: bitwise addition
def xor (a,b):
    c = bytearray(len(a))
    for i in range(len(a)):
        c[i] = a[i] ^ b[i]
    return c


l = 56453

p = int(6 * l - 1)

Fp2 = FiniteField(p,2,
    Polynomial([ModP(1,p),ModP(1,p),ModP(1,p)],p)) #for l=56453

E2 = EllipticCurve2( Fp2([0]), Fp2([1]), Fp2)

b = Fp2([0,1])

DIDCordX = input("Enter the X-coordinate for the Point DID as an
    integer: ")

DIDCordY = input("Enter the Y-coordinate for the Point DID as an
    integer: ")

cypherACordX = 240099

cypherACordY = 283222


print("-------------------------------")
print("Decryption")
```

```
DID = Point2(E2, Fp2([DIDCordX]), Fp2([DIDCordY]))
cypherA = Point2(E2, Fp2([cypherACordX]), Fp2([cypherACordY]))

cypherB = input("Enter the encrypted message, which you want to
    decrypt: ")
#get the binary representation of the encrypted message in hex
cypherB = binascii.unhexlify(cypherB)

length = len(cypherB)


print("The first value of the cyphertext is:")
print(cypherA)

hID = ModifWeil(DID, cypherA, l , b)

print("hID is equal to:")
print(hID)

hash = hash3(hID, length)

print("The decrypted message is:")
c = xor(cypherB , hash)
print(c.decode())
```