Darmstadt University of Technology

Department of Mathematics

Diploma Thesis

Efficient Algorithms for Pairing-Based Cryptosystems

Marcus Stögbauer

4th January 2004

supervised by: Dr. Tsuyoshi Takagi, Assistant Professor

Ich bedanke mich bei Dr. Tsuyoshi Takagi für das interessante Thema und die sehr gute Betreuung während der Ausarbeitung der Diplomarbeit. Desweiteren bedanke ich mich bei Harald Baier und Birgit Henhapl, durch die ich während des Studiums das Thema Elliptische Kurven Kryptographie kennengelernt habe, sowie bei vielen meiner Kommilitonen und Freunden für die Unterstützung und Hilfe, besonders bei Sonja Seifert, Jochen Hechler und Sebastian Schwierz.
Außerdem bedanke ich mich bei meinen Eltern und meiner Familie für die Geduld und Unterstützung während des gesamten Studiums.

Contents

1 Mathematical Background **7**

1.1 Finite Fields 7

1.1.1 Groups, Rings and Fields 7

1.1.2 Polynomial Rings 8

1.1.3 Extension Fields 9

1.2 Elliptic Curve Background 10

1.2.1 Elliptic Curves 10

1.2.2 Representations for Points on Curves 11

1.2.3 Elliptic Curves Group Law 11

1.2.4 m-torsion Points and other Curve Properties 14

1.3 Divisors 15

1.4 The Weil/Tate Pairing 17

1.4.1 The Weil Pairing 17

1.4.2 The Tate Pairing 18

2 Computation of the Weil/Tate Pairing **21**

2.1 Miller's Algorithm 21

2.1.1 TADD 22

2.1.2 TDBL 23

2.1.3 Choice of the Divisor $\mathcal{A}_Q$ 24

2.1.4 The Complete Algorithm 24

2.2 Security Aspects for Cryptographic Applications 26

2.3 Enhancements to Miller's Algorithm 27

2.3.1 Choice of Groups . . . 27
2.3.2 Eliminating Factors in the Update . . . 28
2.3.3 Coordinate System for Points . . . 28
2.3.4 Evaluation of the Line Functions . . . 29
2.3.5 Addition Formula for $[2^w]P$ . . . 30

3 Implementation of the Tate Pairing 31
3.1 Overview over the Class Structure . . . 31
3.2 $\mathbb{F}_{p^k}$ Fields and Arithmetic . . . 33
3.2.1 Extension Fields . . . 33
3.2.2 Addition and Subtraction . . . 34
3.2.3 Multiplication . . . 34
3.2.4 Inversion . . . 35
3.2.5 Exponentiation . . . 36
3.2.6 Square Test . . . 36
3.2.7 Square Root Extraction . . . 37
3.3 Elliptic Curve Arithmetic over $\mathbb{F}_{p^k}$ . . . 37
3.3.1 Point Addition . . . 38
3.3.2 Point Doubling . . . 39
3.3.3 Efficient Iterated Point Doubling . . . 40
3.3.4 Generation of Random and l-torsion Points on the Curve 40
3.4 Tate Pairing . . . 41

4 Timing Results 45
4.1 Timing Results of the Underlying Arithmetics . . . 47
4.1.1 Timing Results of the Underlying Arithmetics for Ordinary Curves . . . 47
4.1.2 Timing Results of the Underlying Arithmetics for Supersingular Curves . . . 48
4.2 Timing Results for Miller's Algorithm . . . 49
4.2.1 Timing Results for Ordinary Curves . . . 50
4.2.2 Timing Results for Supersingular Curves . . . 51

5 Identity Based Encryption 53

5.1 IBE scheme with Tate Pairings 54

5.1.1 Setup 54

5.1.2 Extract 54

5.1.3 Encrypt 56

5.1.4 Decrypt 56

5.2 Security of the IBE Implementation 56

5.3 Java Implementation of the IBE Scheme 58

6 Future Works 59

A Derivation of the Line Equations 61

A.1 Lines for Addition 61

A.2 Lines for Doubling 63

B Example Computation of the Tate Pairing 67

B.1 Computation of $t([2]P, Q)$ 68

B.2 Computation of $t(P, Q)$ 68

Notation and Abbreviations

$\mathbb{F}$	finite field	8
$\mathbb{F}_q$	finite field with q elements	8
$\mathbb{F}[x]$	polynomial field	8
$\mathbb{F}[x]/(f)$	residue class ring of polynomials	9
$\mathbb{F}_{p^m}$	extension field with degree m	9
$\mathbb{P}^2$	projective plane	10
$E(\mathbb{F})$	elliptic curve over a finite field	11
$\mathcal{O}$	point at infinity on an elliptic curve	11
$[m]P$	m additions of the point P	13
$l(x, y)$	line through points on the curve	13
$E[m]$	m-torsion subgroup of the curve E	14
$\#E(\mathbb{F}_q)$	number of points on the curve E, curve order	14
$\mathcal{A}$, $\mathcal{B}$	divisors	15
$\mathrm{Div}\,(E)$	divisor group of a curve E	15
$\deg\,(\mathcal{A})$	degree of the divisor $\mathcal{A}$	15
$\mathrm{Div}^{\,d}(E)$	divisor group of degree d of a curve E	15
(f)	divisor of a function on a curve	15
$f(\mathcal{A})$	evaluation of a function at a divisor	16
$\mathcal{A} \equiv \mathcal{B}$	equivalence of divisors	16
$divA_P$	divisor equivalent to the divisor $(P) - (\mathcal{O})$	17
(f_P)	principal divisor with $(f_P) = n\mathcal{A}_P$	17
$e_n(P, Q)$	Weil pairing of the n-torsion points P and Q	17
$t(P, Q)$	Tate pairing of the points P and Q	18
TADD	Addition step in Miller's algorithm	22
TDBL	Doubling step in Miller's algorithm	23
M, S	one multiplication or squaring in $\mathbb{F}_q$	23
M_k, S_k	one multiplication or squaring in $\mathbb{F}_{q^k}$	23
l_1^{add}	line through P and Q during TADD	23
l_2^{add}	line through $P + Q$	23
l_1^{dbl}	tangent on P during TDBL	23
l_2^{dbl}	line through $[2P]$	23
DLP	discrete logarithm problem	26
$\mathcal{J}^s$	simplified Chudnovsky-Jacobian coordinate system	29

List of Algorithms

1	Basic outline for Miller's algorithm	22
2	Miller's algorithm	25
3	Schoolbook multiplication	34
4	Reduction of elements	35
5	Extended Euclidean Algorithm	36
6	Fast exponentiation	37
7	Square root extraction	38
8	Point addition	39
9	Point doubling	40
10	Iterated EC double algorithm	40
11	Generating a l-torsion point	42
12	Find a point to a given x	54
13	Map an arbitrary string to a l-torsion point on $E(\mathbb{F}_p)$	55

Introduction

Pairings were first introduced to elliptic curve cryptography for "destructive" methods like the MOV reduction [MOV93]. With the help of the Weil pairing, Menezes, Okamoto and Vanstone showed a way to reduce the discrete logarithm problem on supersingular elliptic curves to the discrete logarithm problem of an extension of the underlying finite field.
Later Frey, Müller and Rück extended this attack to more general elliptic curves with the Tate pairing [FMR99].

But the Weil and Tate pairing can also be used as a constructive tool for cryptography. Boneh and Franklin, for example, proposed an IBE scheme from the Weil pairing in [BF01] and also gave details on how to use the Tate pairing in such a scheme.

Especially the IBE scheme is interesting, since it is an alternative to public key algorithms (like RSA) which simplifies certificate management:
When Alice sends an email to Bob, she just uses Bobs email address (for example bob@firma.de) as a public key string to encrypt the email. When Bob wants to decrypt that email, he contacts a third party called "private key generator" (PKG), authenticates to this PKG just like he would authenticate himself to a CA and obtains his private key to decrypt the email.
The advantages of the IBE scheme over other public key techniques are obvious: Alice does not need a certified public key from Bob to be sure it is really Bob to whom she sends the encrypted message. She only has to trust the PKG. Also, note that Alice can send encrypted emails to Bob even if Bob has not yet received his private key from the PKG. With other public key techniques, Bob would have to generate a key pair, send his public key to a CA, get a certificate and send the public key to Alice before Alice can send encrypted emails to Bob.

Only recently, Okamoto and Pointcheval [OP01] discovered a new class of problems in which the Decision Diffie-Hellman problem is easy, but the Computational Diffie-Hellman problem is hard. With this property, and the even more important bilinearity, the Weil and Tate pairing can be used for

many other cryptographic applications. In his survey [Jou02] Joux proposed a couple of primitives with the use of the Weil and Tate pairing.

As a drawback, these cryptographic primitives and the IBE scheme usually require special curves, for example supersingular curves, which have been proven weak in other cryptographic applications. But with the recent results from Miyaji, Nakabayashi and Takano [MNT01] and Dupont, Enge and Morain [DEM02], it is possible to construct ordinary elliptic curves with given (small) MOV degree, which provides more freedom in cryptographic applications and so more possibilities to react on potential future security problems with these special curves.

Fast implementations of Weil and Tate pairings are as well subject of research. The building stone of all implementations is Miller's algorithm [Mil86], which evaluates a function on an elliptic curve at a divisor. Many of the proposed enhancements, though, concentrate on supersingular curves or curves over characteristic 2 and 3 finite fields. But the discrete logarithm problem over $\mathbb{F}_{2^m}$ can be solved faster than the problem over other finite fields, and so the size of the field must be chosen larger. Furthermore, there exist very efficient implementations of fields $\mathbb{F}_p$, $p > 3$ prime, so it would be a natural processing to choose curves over $\mathbb{F}_p$.
Fortunately, some of the proposed enhancements can be used for ordinary elliptic curves, too. This observation and more enhancements for ordinary curves have been proposed by Izu and Takagi in [IT03].

Works in this Thesis

In this thesis, we summarize the necessary mathematical theory behind the Weil and Tate pairing for two points $P, Q \in E(\mathbb{F}_{p^k})$ and give a detailed description of Miller's algorithm and the enhancements proposed in numerous papers. We also give an efficient implementation of the specified algorithms and methods in Java for the FlexiProvider [Fle] and an implementation of the IBE scheme as an example application of pairing-based cryptosystems.

The basic version of Miller's algorithm evaluates a function f_P on the elliptic curve $E(\mathbb{F}_{p^k})$ at a divisor $\mathcal{A}_Q = (Q) - (\mathcal{O})$, where Q is a point on the elliptic curve.
But since the support of the divisor $\mathcal{A}_Q$ and the principal divisor (f_P) needs to be disjoint, we have to modify the algorithm: instead of $f_P(\mathcal{A}_Q)$ we evaluate $f_P(Q+S)/f_P(S)$ where $S \in E(\mathbb{F}_{p^k})$ is a random point on the curve. This way, the supports of $\mathcal{A}_Q$ and (f_P) are disjoint but in negligible many cases.

For this new formulation of Miller's algorithm we apply the enhancements. A huge computation enhancement is to choose $S \in E(\mathbb{F}_p)$, so we can compute

many steps in the subfield instead of computing in the extension field.

The function f_P on the curve occurring during Miller's algorithm are line equations through points on the curve that appear in the addition and doubling. These line equations can be optimized, too. We carefully implement the evaluations of the line equations and so save some computation time, because we use intermediate results from already finished computations to speed up the evaluation.
Another way to optimize the line equations can be found in the final exponentiation in the Tate pairing. The Tate pairing is only unique up to the l.th power, therefore we need to raise the result of Miller's algorithm to the power of $(p^k - 1)/l$ to eliminate these l.th powers. This final exponentiation has another impact on the computation: factors $a \in \mathbb{F}_p$ occurring in the evaluation are eliminated, too. We use this fact to further reduce the number of computations during Miller's algorithm, especially during the evaluation of the line equations.

As one can see, the mathematical background needed for pairings is broadly based. We need to know about finite (extension) fields, elliptic curves and their properties, divisor theory, bilinear maps and of course the Weil and Tate pairing itself.
But understanding the theory is only one part. Implementing the Tate Pairing is no trivial task, either. The implementation is done in Java within the FlexiProvider toolkit, so we have a platform independent implementation.
Unfortunately, there were no implementations for extension fields or polynomials over finite fields available, so we had to implement those, too. Our first approach was to use Optimal Extension Fields (OEFs), which have computing time advantages over standard implementations, especially during the multiplication and inversion. But as it turned out, finding matching parameters for OEFs and the Tate pairing is no easy task, so we had to discard the use of OEFs and make do with a slower implementation. Implementations for OEFs, however, were done for this thesis and are available now in the FlexiProvider. This way one can benefit from the speed up when matching parameters are found.
There were also problems during the debugging of the Java implementation. The mathematical aspect of pairings is difficult, and the computation of a pairing value uses many steps. To understand where errors are is difficult under these circumstances. During the implementation phase of this thesis, we used different methods to find such errors. The first was to check the theory thoroughly, and as a result we give a derivation of line equations occurring in the algorithm in appendix A. The second technique was to compute the Tate pairing by hand. This way, we could compare the intermediate results from the Java implementation with the ones from the computation by hand and find the according lines in the code that produced the errors. As a bonus, we now have a sample computation of the Tate pairing, which can

be found in appendix B.
The biggest problem, though, is the choice of the system parameters for the Tate pairing, i.e. the choice of an appropriate elliptic curve with a small MOV degree and the value l for an l-torsion subgroup of the elliptic curve. For supersingular curves, there are plenty of choices available. But research on constructing ordinary elliptic curves with small MOV degree has only just begun, so there are only a few curves published so far. Once more results are available, the computing time of the Tate pairing with ordinary elliptic curves can decrease vitally.

Structure of the Thesis

The thesis is organized as follows:
We give an overview over the needed mathematical theory in Chapter 1. Here, we summarize definitions and results for finite fields, extension fields, elliptic curves, point addition on curves, divisors and the Tate and Weil pairing.
Chapter 2 describes Miller's algorithm, which is used for the evaluation of the pairings, security considerations arising from the use of pairing-based cryptosystems and speed enhancements to Miller's algorithm and to the computation of the Weil and Tate pairing in general.
In chapter 3 we give implementation details and algorithms for extension field arithmetics, arithmetic on points on curves and generation of (l-torsion) points.
Then we summarize the complexity of the methods given by the enhancements and give timing results for the Java implementation in chapter 4.
Chapter 5 contains the description of an IBE scheme called "BasicIdent" from [BF01], security considerations for this scheme and implementation details.
Finally, we name possible further speed optimizations for the computation of the Weil and Tate pairing in chapter 6.

Chapter 1

Mathematical Background

In this chapter we present mathematical theory which we need to understand and implement algorithms for pairing-based cryptosystems.
The first section deals with finite fields and extension fields. In the second section, we will give the needed theory on elliptic curves. We introduce divisor theory in the third section, which we use extensively in the fourth section where we describe the Weil and Tate pairing and the respective properties of the pairings.

1.1 Finite Fields

This section deals with finite fields, one of the fundamental elements for pairing-based cryptosystems and elliptic curve cryptography. We will only give a short introduction to finite fields. For more details on this subject, see [LN86].
Throughout this thesis, we will denote arbitrary sets with a normal letter G, F and so on, and finite sets with $\mathbb{F}$.

1.1.1 Groups, Rings and Fields

First, we give the definition of a group, a ring and a field.

Definition 1.1 *Let G be a non-empty set and $\circ$ a binary operation on G. We say $(G, \circ)$ is a* group*, if the following three properties hold:*

a) $\circ$ is associative, i.e. $\forall a, b, c \in G$, $a \circ (b \circ c) = (a \circ b) \circ c$.

b) G has an identity *(or* unity*) $e \in G$, such that $\forall a \in G$, $a \circ e = e \circ a = a$.*

c) *For each $g \in G$ there is an inverse element $g^{-1} \in G$ such that $g \circ g^{-1} = e$.*

We say the group is commutative *or* abelian *if for all $a, b \in G$, $a \circ b = b \circ a$. A group is called* finite *if it contains finitely many elements. The number of elements in a finite group is called* order *and is denoted by $|G|$.*

Definition 1.2 *A* ring *is a triple $(R, +, \cdot)$, where $+$ and $\cdot$ are two binary operations and the following properties hold:*

a) *R is an abelian group with respect to $+$.*

b) *$\cdot$ is associative.*

c) *The distributive law holds, i.e. $\forall a, b, c \in R$ we have $a \cdot (b+c) = a \cdot b + a \cdot c$ and $(b + c) \cdot a = b \cdot a + c \cdot a$.*

The ring is called ring with identity *if the ring has a multiplicative identity and* commutative *if $\cdot$ is commutative.*
Consequently, $a \in R$ is called invertible (or unit) if there is an inverse element of R with respect to the operation $\cdot$.

Definition 1.3 *Let G be a ring with zero element 0. A* field *$\mathbb{F}$ is a commutative ring with identity where every element $g \in G$, $g \neq 0$ is invertible. We say $g \in \mathbb{F}$ has* order *a if $g^a = 1$, where 1 is the multiplicative identity element of $\mathbb{F}$.*

In the following we will denote by $\mathbb{F}_q$ a finite field with q elements.

1.1.2 Polynomial Rings

Let $\mathbb{F}$ be a field. A *polynomial* is an expression of the form $f(x) = \sum_{i=0}^{n} a_i x^i$ with $n \geq 0$. The a_i are called *coefficients*. If $a_i \in \mathbb{F}$, $i = 0, \cdots, n$ we call it a *polynomial over* $\mathbb{F}$. We follow the usual convention that coefficients $a_i = 0$ can be left out.

Definition 1.4 *A polynomial ring $\mathbb{F}[x]$ is formed by the polynomials over a field $\mathbb{F}$ with the operations $+$ and $\cdot$. Let $f(x), g(x) \in \mathbb{F}[x]$, $f(x) = \sum_{i=0}^{n} a_i x^i$, $g(x) = \sum_{i=0}^{m} b_i x^i$, then we define the operations as follows:*

- $f(x) + g(x) = \sum_{i=0}^{\max(n,m)} (a_i + b_i) x^i$
- $f(x) \cdot g(x) = \sum_{k=0}^{n+m} c_k x^k$ *with* $c_k = \sum_{i+j=k,\ 0 \leq i \leq n,\ 0 \leq j \leq m} a_i b_j$

The zero element *of* $\mathbb{F}[x]$ *is the polynomial* $h(x) \in \mathbb{F}[x]$ *with* $h(x) = \sum_{i=0}^{n} 0x^i$.

Let $f(x) = \sum_{i=0}^{n} a_i x^i \neq 0$, so we can assume $a_n \neq 0$. We call a_n the *leading coefficient* and n the *degree* of f. We write $deg(f) = deg(f(x)) = n$. If the leading coefficient of f is $1 \in \mathbb{F}$, where 1 is the identity in $\mathbb{F}$, we say f is a *monic polynomial*.

Next we will introduce divisions of elements $f, g \in \mathbb{F}[x]$. We say g divides the polynomial f if there exists a polynomial $h \in \mathbb{F}[x]$ such that $f = gh$. We also have a division with remainder:

Theorem 1.5 (Division algorithm) *Let* $0 \neq g \in \mathbb{F}[x]$. *Then for any* $f \in \mathbb{F}[x]$ *there exist polynomials* $q, r \in \mathbb{F}[x]$ *such that*

$$f = qg + r\,, \text{ with } deg(r) < deg(g)\,.$$

Now we introduce an important type of polynomial, the irreducible polynomial.

Definition 1.6 *A polynomial* $p \in \mathbb{F}[x]$ *is called* irreducible over $\mathbb{F}$*, if* p *has positive degree and* $p = bc$*, such that* $b, c \in \mathbb{F}[x]$ *implies that either* b *or* c *is a constant polynomial.*

As we see from the definition, the decision whether a polynomial is irreducible depends on the field $\mathbb{F}$ we are working on. This will be important to remember later on.
With this irreducible polynomial, we can build residue classes of polynomials, denoted by $\mathbb{F}[x]/(f)$. One important property of such classes is:

Theorem 1.7 *Let* $f \in \mathbb{F}[x]$. *The residue class ring* $\mathbb{F}[x]/(f)$ *is a field if and only if* f *is irreducible over* $\mathbb{F}$.

1.1.3 Extension Fields

Now we present theory to define extension fields and give a representation of elements from extension fields.

Definition 1.8 *Let* $\mathbb{F}_p$ *be a finite field, with a prime* p. *The field* $\mathbb{F}_{p^m}$ *with an integer* $m > 1$ *is called an* extension field *of the* subfield $\mathbb{F}_p$.

To represent elements of the extension field, we use a residue class ring $\mathbb{F}[x]/(f)$, where f is an irreducible monic polynomial over $\mathbb{F}_p$ with degree m. We call f the *field polynomial*. Elements $A \in \mathbb{F}_{p^m}$ are then represented by polynomials in the residue class ring:

$$A = a_{m-1}x^{m-1} + \cdots + x_1 x^1 + a_0 x^0$$

with $a_i \in \mathbb{F}_p$, $i = m-1, \cdots, 0$. Arithmetic operations in the extension field are operations on the polynomials modulo the field polynomial.
Algorithms for arithmetic operations on extension fields are given in section 3.2.

1.2 Elliptic Curve Background

In this section, we will give a short overview on the theory of elliptic curves, as far as we will use them in this thesis.
For a more thorough and comprehensive treatment of elliptic curves theory please refer to one of the many books about elliptic curves, e.g. [ST92], [Sil86] or [Men93].

1.2.1 Elliptic Curves

Let $\mathbb{F}$ be a field and $\bar{\mathbb{F}}$ its algebraic closure. The *projective plane* $\mathbb{P}^2$ consists of the set of triples (a, b, c), with $(a, b, c) \neq (0, 0, 0)$, such that two triples (a, b, c) and (d, e, f) are considered to be the same point if there is a $n \in \mathbb{N}$, $n \neq 0$ with $a = nd$, $b = ne$, $c = nf$. We denote the elements of $\mathbb{P}^2$ with $(x : y : z)$.

Definition 1.9 *An* elliptic curve over $\mathbb{F}$ *is the set of solutions in the projective plane* $\mathbb{P}^2(\bar{\mathbb{F}})$ *of a homogeneous* Weierstrass equation *of the form*

$$\begin{aligned} E &: Y^2Z + a_1XYZ + a_3YZ^2 \\ &= X^3 + a_2X^2Z + a_4XZ^2 + a_6Z^3 \text{ or} \\ F(X,Y,Z) &= Y^2Z + a_1XYZ + a_3YZ^2 - \\ & X^3 - a_2X^2Z - a_4XZ^2 - a_6Z^3 \end{aligned} \tag{1.2.1}$$

with $a_i \in \mathbb{F}$, $i = 1, \cdots, 6$.

This equation is also known as the *long Weierstrass form.*

Throughout the chapters, we may use the *affine* representation of the Weierstrass form, where we set $x = X/Z$ and $y = Y/Z$:

$$\begin{aligned} E &: y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6 \\ \text{or} & \\ f(x,y) &= y^2 + a_1xy + a_3y - x^3 - a_2x^2 - a_4x - a_6 \end{aligned} \tag{1.2.2}$$

There is also a *short Weierstrass form.* For $\mathrm{char}(\mathbb{F}) \neq 2, 3$, this form is

$$E : y^2 = x^3 + ax + b\,. \tag{1.2.3}$$

1.2.2 Representations for Points on Curves

There are different sets of coordinates we can use to represent a point on an elliptic curve. The two most common coordinate sets are affine and projective coordinates. As mentioned above, affine representations have the form (x, y) and projective representations are $(X : Y : Z)$. To convert between these two representations, we set $x = X/Z$ and $y = Y/Z$.

In this thesis, we will also use *Jacobian* and *Chudnovsky Jacobian* coordinates.
Jacobian coordinates are represented, just as projective coordinates, by three parameters (X, Y, Z). But, unlike projective coordinates, we transfer a point from the affine to its Jacobian representation by setting $x = X/Z^2$ and $y = Y/Z^3$.
Chudnovsky Jacobian coordinates are basically Jacobian coordinates (hence we use the same transfer formula to convert them to an affine representation), but the values Z^2 and Z^3 are also present. So the representation is (X, Y, Z, Z^2, Z^3).
Each representation has its advantages and disadvantages. For example, affine coordinates require a division in every addition and doubling, but fewer multiplications than projective coordinates.
Throughout the theory part of this thesis, we will mostly use the affine representation of points. In praxis, we will mostly use Jacobian representations, since they give us faster point doubling, which is useful in Miller's algorithm.

1.2.3 Elliptic Curves Group Law

The set of points of an elliptic curve over a finite field $\mathbb{F}$ together with the point $\mathcal{O}$ at infinity as a neutral element form an abelian group with the point addition. We will now describe what the point $\mathcal{O}$ really is and how the point addition works.

Definition 1.10 *An elliptic curve is called* smooth *or* non-singular *if for all points $P \in E$ on the curve, which satisfy the equation $F(X, Y, Z) = 0$, the partial derivatives $\frac{\partial F}{\partial X}, \frac{\partial F}{\partial Y}, \frac{\partial F}{\partial Z}$ are not 0.*

In the following, we will regard elliptic curves over a finite field $\mathbb{F}$, that is the set of points in $\mathbb{F}$ which satisfy a smooth Weierstrass equation (1.2.1). We will write $E(\mathbb{F})$ for such a curve.

Definition 1.11 *The curve has exactly one rational point with $Z = 0$: $(0 : 1 : 0)$. We will call this the point at infinity and write it as $\mathcal{O}$.*

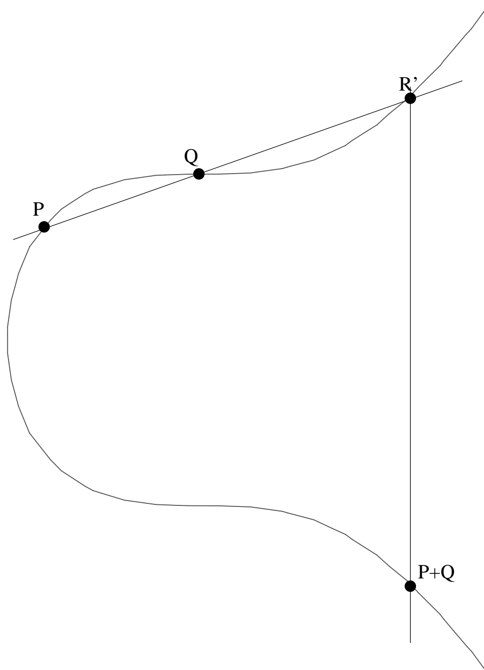

Figure 1.1: Addition on a curve, with $Q \neq \pm P$

Consider the elliptic curve

$$E : y^2 = x^3 + ax + b\,. \tag{1.2.4}$$

Now, we define a group law on $E(\mathbb{F})$: the set of points of an elliptic curve over a finite field with the point at infinity $\mathcal{O}$ as a neutral element and the point addition form an abelian group.

The inverse of a point $P = (x, y)$ is the point mirrored at the x axis, i.e. $-P = (x, -y)$.

Geometrically, we can describe the addition as follows: Let $P, Q \in E(\mathbb{F}) = E$, $Q \neq \pm P$ and $P, Q \neq \mathcal{O}$. Then draw a line l_1 through the points P and Q. This line will intersect the curve E at exactly one more point $R' = P \times Q$. Draw a line l_2 parallel to the y axis through R'. The line l_2 will intersect the curve at exactly one more point $R = P + Q$, the result of the addition. See figure 1.1 for a graphical explanation.

As we have excluded $Q = \pm P$ and $Q = \mathcal{O}$ from the addition so far, there are three special cases.

- Let $Q = P$. Here, the line l_1 through P and Q is not unique anymore.

Instead, we choose the line l_1 as the tangent on the curve through the point P.

- Let $Q = -P$. Here, the line l_1 does not intersect the curve anymore. We call the resulting point R the point at infinity $\mathcal{O}$.
- If $Q = \mathcal{O}$, then we have $P + \mathcal{O} = P$, since $\mathcal{O}$ is the neutral element.

We denote for $m \in \mathbb{Z}$, $P \in E$

$$\begin{aligned} [m]\,P &= P + \cdots + P\,,\ m \text{ terms, for } m > 0 \\ [0]\,P &= \mathcal{O} \\ [m]\,P &= [-m]\,(-P) \ \text{ for } m < 0 \end{aligned}$$

We still use the elliptic curve

$$E:\ y^2 = x^3 + ax + b\,. \tag{1.2.4}$$

We can also give explicit formulas for the addition on this curve. First, for affine coordinates:
Let $P_1 = (x_1, y_1)$, $P_2 = (x_2, y_2)$, $P_1, P_2 \neq \mathcal{O}$. When $P_1 \neq \pm P_2$, set

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

and when $P_1 = P_2$, set

$$\lambda = \frac{3x_1^2 + a}{2y_1}\,,\ v = \frac{-x_1^3 + ax_1 + 2b}{2y_1}\,.$$

Then $P_3 = (x_3, y_3) = P_1 + P_2$ is given by

$$\begin{aligned} x_3 &= \lambda^2 - x_1 - x_2 \\ y_3 &= \lambda(x_1 - x_3) - y_1\,. \end{aligned}$$

The equation for the line through P_1 and P_2, or the tangent to E at P_1 if $P_1 = P_2$, is given by

$$l(x, y) = y - \lambda x - v\,. \tag{1.2.6}$$

For Jacobian coordinates, we recall that setting $x = X/Z^2$, $y = Y/Z^3$ will lead us from the affine representation to the Jacobian representation, so the curve (1.2.4) in Jacobian coordinates is

$$E: Y^2 = X^3 + axZ^4 + bZ^6\,. \tag{1.2.7}$$

We have $P_1 = (X_1, Y_1, Z_1)$, $P_2 = (X_2, Y_2, Z_2)$ and want to compute $P_3 = P_1 + P_2 = (X_3, Y_3, Z_3)$. When $P_1 \neq \pm P_2$, the point P_3 is given by

$$\begin{aligned} X_3 &= -(X_2 Z_1^2 - X_1 Z_2^2)^3 - 2X_1 Z_2^2 (X_2 Z_1^2 - X_1 Z_2^2)^2 + (Y_2 Z_1^3 - Y_1 Z_2^3)^2 \\ Y_3 &= -Y_1 Z_2^3 (X_2 Z_1^2 - X_1 Z_2^2)^3 + \\ &\quad (Y_2 Z_1^3 - Y_1 Z_2^3) \left(X_1 Z_2^2 (X_2 Z_1^2 - X_1 Z_2^2)^2 - X_3 \right) \\ Z_3 &= Z_1 Z_2 (X_2 Z_1^2 - X_1 Z_2^2) \,. \end{aligned} \tag{1.2.8}$$

For the doubling, i.e. if $P_1 = P_2$, the point $P_3 = [2]P_1 = (X_4, Y_4, Z_4)$ is given by

$$\begin{aligned} X_4 &= -8X_1 Y_1^2 + (3X_1^2 + aZ_1^4)^2 \\ Y_4 &= -8Y_1^4 + \\ &\quad (3X_1^2 + aZ_1^4)(4X_1 Y_1^2 + 8X_1 Y_1^2 - (3X_1^2 + aZ_1^4)^2) \\ Z_4 &= 2Y_1 Z_1 \,. \end{aligned} \tag{1.2.9}$$

1.2.4 m-torsion Points and other Curve Properties

Definition 1.12 *The m-*torsion subgroup *of E ($m \in \mathbb{Z}$, E is an elliptic curve), denoted by $E[m]$, is the set of points of order m in E:*

$$E[m] := \left\{ P \in E \middle| \, [m] \, P = \mathcal{O} \right\} .$$

*$P \in E[m]$ is called an m-*torsion point*.*

Take a finite field $\mathbb{F}_q$ with q elements, $q = p^m$ with a prime p and a positive integer m and consider an elliptic curve over this field $E(\mathbb{F}_q)$.
The number of rational points on this curve is finite and hereby denoted by $\#E(\mathbb{F}_q)$. In the equation

$$\#E(\mathbb{F}_q) = q + 1 - t$$

t is called the *trace of Frobenius*.

Definition 1.13 *An elliptic curve $E(\mathbb{F}_q)$ is called* supersingular *if the characteristic p divides the trace of Frobenius t.*

Let $E : y^2 = x^3 + ax + b$ a smooth elliptic curve, $a, b \in \mathbb{F}_q$. $E(\mathbb{F}_q)$ is the curve with point elements in $\mathbb{F}_q$ and $n = \#E(\mathbb{F}_q) = q + 1 - c_1$ is the known order of the curve.
Now consider the curve $E(\mathbb{F}_{q^k})$ with an integer $k > 1$. Since we use the same parameters a, $b \in \mathbb{F}_q$ for both curves, the curve $E(\mathbb{F}_{q^k})$ is called a *subfield curve* and there is an easy way to obtain the group order $\tilde{n} = \#E(\mathbb{F}_{q^k})$ from the group order n. This method is described in [BSS99], we will only give the relevant result here.

Lemma 1.14 *Let E be an elliptic curve over $\mathbb{F}_q$ and $n = \#E(\mathbb{F}_q) = q + 1 - c_1$.*
Then $\tilde{n} = \#E(\mathbb{F}_{q^k}) = q^k + 1 - c_k$ for $k > 1$ with the following series. We set $c_0 = 2$, c_1 as above and

$$c_k = c_1 c_{k-1} - q c_{k-2} \,.$$

In the next section we introduce divisors. One important element of the divisor theory are functions on a curve:

Definition 1.15 *A* function on the curve $E(\mathbb{F})$ *is a rational function $f(x, y) \in \mathbb{F}$. For any point $(x, y) = P \in E(\mathbb{F})$ we define $f(P) = f(x, y)$.*

1.3 Divisors

Divisors are a crucial part of the Weil and Tate pairing. In this section, divisors are defined and some results of divisor theory are given. More details and proofs can be found in [Men93], chapter 2.7, and [Sil86], chapter II.3.

Definition 1.16 *A* divisor *is the formal sum of points on the curve $E = E(\mathbb{F})$. Throughout this thesis, we will write divisors as $\mathcal{A} = \sum_{P \in E} a_p(P)$, with $a_p \in \mathbb{Z}$ and $a_p = 0$ for all but finitely many $P \in E$. We will write $Div(E)$ for the divisor group of a curve E.*
$Div(E)$ forms a group with the addition

$$\sum_{P \in E} n_P(P) + \sum_{P \in E} m_P(P) = \sum_{P \in E} (n_P + m_P)(P) \,.$$

The degree of a divisor *is $deg(\mathcal{A}) = \sum_{P \in E} a_p$. Let $d \in \mathbb{Z}$. We write $Div^d(E)$ for all divisors of degree d of the curve E.*
The support *of a divisor is $supp(\mathcal{A}) = \{P \in E \,|\, a_p \neq 0\}$.*

Mostly we will consider divisors with degree 0 with the according divisor group $\mathrm{Div}^0(E)$.

Example 1.17 *Let $P_i \in E$, $i = 1, 2, 3$. $\mathcal{A} = 3(P_2) - 2(P_1) - 1(P_3)$ is a divisor with degree 0.*

Definition 1.18 *Let f be a function on the smooth curve $E(\mathbb{F}) = E$. The divisor of f, denoted by (f) , is defined as*

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

where $ord_P(f)$ is the order of the zero (or the pole) which f has at the point P. $ord_P(f)$ is positive if P is a zero, and negative if P is a pole of f.

Example 1.19 *Let $ax + by + c = 0$ be the line through the points $P_1, P_2 \in E(\mathbb{F}_q) = E$ with $P_1 \neq \pm P_2$ and let P_3 be the third point on the curve and the line.*
Then, the function $f(x, y) = ax + by + c$ has 3 zeros (namely P_1, P_2 and P_3) of order 1 and a pole of order 3 at $\mathcal{O}$.
So the divisor of $f(x, y)$ is

$$(f) = (P_1) + (P_2) + (P_3) - 3(\mathcal{O}) .$$

Definition 1.20 *Let $\mathcal{A}$ be a divisor and f a function such that $(f) = \mathcal{A}$. Then we call $\mathcal{A}$ a* principal divisor*.*

Let (f) be a principal divisor. For a divisor $\mathcal{A} = \sum_{P \in E} a_P(P)$, where (f) and $\mathcal{A}$ have disjoint supports, we define $f(\mathcal{A})$ as

$$f(\mathcal{A}) = \prod_{P \in E} f(P)^{a_P} . \tag{1.3.1}$$

Remark 1.21 *The restriction above, that the support of $\mathcal{A}$ and the support of (f) has to be disjoint, makes sense out of the following reason: Suppose that $supp((f)) \cap supp(\mathcal{A}) \neq \emptyset$. Then there is a point $\hat{P}$ such that $f(\hat{P}) = 0$ or $f(\hat{P}) = \infty$ and $a_{\hat{P}} \neq 0$. So those points would appear in the multiplication (1.3.1) and so the result is either ∞ or 0.*

Definition 1.22 *Let $\mathcal{A}$, $\mathcal{B}$ be two divisors. We call them* equivalent*, denoted as $\mathcal{A} \sim \mathcal{B}$, if their difference $\mathcal{A} - \mathcal{B}$ is a principal divisor. This condition can also be written as*

$$\mathcal{A} \sim \mathcal{B} \Leftrightarrow \mathcal{A} = \mathcal{B} + (f)$$

for some function f on the curve.

Finally, we will give a couple of useful properties of (principal) divisors which we will use in the following chapters.

Lemma 1.23

a) Any divisor $\mathcal{A} = \sum_{P \in E} a_p(P)$ with degree 0 is equivalent to a divisor of the form $\mathcal{A}' = (Q) - (\mathcal{O})$ for some $Q \in E(\mathbb{F})$. If $\mathcal{A} \in Div^0$, then $Q = \sum [a_p]P$.

b) For every divisor $\mathcal{A} \in Div^0(E)$ exists a unique point $P \in E$ such that $\mathcal{A} \sim (P) - (\mathcal{O})$. We define a map σ given by this association as

$$\sigma : Div^0(E) \rightarrow E .$$

c) *This map is surjective and for two divisors $\mathcal{A}$, $\mathcal{B} \in Div^0(E)$ the following is true:*

$$\sigma(\mathcal{A}) = \sigma(\mathcal{B}) \Leftrightarrow \mathcal{A} \sim \mathcal{B}\,.$$

Lemma 1.24 *A divisor $\mathcal{A}$ is principal if and only if $\sum_{P\in E} a_p = 0$ and $\sum_{P\in E}[a_P]P = \mathcal{O}$.*

A proof for both lemmas can be found in [Sil86], pages 66, 67.

1.4 The Weil/Tate Pairing

With the knowledge above, we are ready to define the Weil and Tate pairing.

1.4.1 The Weil Pairing

Let $E = E(\mathbb{F})$ be an elliptic curve over a finite field $\mathbb{F}$. Let $P, Q \in E[n]$ be points, $\mathcal{A}_P$ a divisor equivalent to the divisor $(P) - (\mathcal{O})$.
Note that $n\mathcal{A}_P$ is a principal divisor[1], so there exists a function f_P such that $(f_P) = n\mathcal{A}_P$. $\mathcal{A}_Q$ and f_Q are defined in the same way.

Definition 1.25 *The* Weil pairing *of P and Q is*

$$\begin{aligned} e_n : E[n] \times E[n] &\rightarrow \mathbb{F} \\ e_n(P,Q) &= \frac{f_P(\mathcal{A}_Q)}{f_Q(\mathcal{A}_P)} \end{aligned} \tag{1.4.1}$$

Theorem 1.26 *Let $P, P_1, P_2, Q, Q_1, Q_2 \in E[n]$ if not defined otherwise. The Weil pairing e_n satisfies the following properties:*

- *bilinear:*

$$\begin{aligned} e_n(P_1 + P_2, Q) &= e_n(P_1, Q)e_n(P_2, Q) \\ &\textit{and} \\ e_n(P, Q_1 + Q_2) &= e_n(P, Q_1)e_n(P, Q_2)\,. \end{aligned}$$

- *alternating:*

$$e_n(P,Q) = e_n(Q,P)^{-1}\,.$$

- *non-degenerate: If $e_n(P,Q) = 1$ for all $P \in E[n]$, then $Q = \mathcal{O}$.*
- *compatible: If $P \in E[nn']$, $Q \in E[n]$, then $e_{nn'}(P,Q) = e_n([m']P, Q)$.*

For a proof of this theorem, see [Sil86], pages 96-98.

[1] $n\mathcal{A}_P \sim n(P) - n(\mathcal{O})$, i.e. $\sum_{P\in E} a_P = n - n = 0$ and $\sum_{P\in E}[a_P]P = [n]P - [n]\mathcal{O} = \mathcal{O} - \mathcal{O} = \mathcal{O}$, so according to lemma 1.24, $n\mathcal{A}_P$ is principal.

1.4.2 The Tate Pairing

In the following, we write $\mathbb{F}_q^* = \mathbb{F}_q \setminus \{0\}$. For the Tate pairing, we consider elliptic curves $E(\mathbb{F}_q)$ over the finite field $\mathbb{F}_q$, $q = p^m$ with a prime p and a positive integer m.
$E(\mathbb{F}_q)[l]$ are the points on the curve with order l, with l coprime to q. We usually choose $l|\#E(\mathbb{F}_q)$.
Let k be another positive integer with

$$l|(q^k - 1) \text{ and } l \not| (q^s - 1) \text{ for } 0 < s < k \tag{1.4.2}$$

the so called *security multiplier* or *MOV degree* (see [MOV93]).
Let $P \in E(\mathbb{F}_{q^k})[l]$, $Q \in E(\mathbb{F}_{q^k})/lE(\mathbb{F}_{q^k})$. The divisor $\mathcal{A}_Q$ is, as defined above, a divisor equivalent to the divisor $(Q) - (\mathcal{O})$, $\mathcal{A}_P$ a divisor equivalent to $(P) - (\mathcal{O})$ and $(f_P) = n\mathcal{A}_P$.

Definition 1.27 *The* Tate pairing *of P and Q is defined as follows:*

$$\begin{aligned} t : E(\mathbb{F}_{q^k})[l] \times E(\mathbb{F}_{q^k})/lE(\mathbb{F}_{q^k}) &\rightarrow \mathbb{F}_{q^k}^*/(\mathbb{F}_{q^k}^*)^l \\ t(P, Q) = f_P(\mathcal{A}_Q) \,. \end{aligned} \tag{1.4.3}$$

If we want to stress out which l we use, we may write $t_l(P, Q)$ for a specific l.

The group $\mathbb{F}_{q^k}^*/(\mathbb{F}_{q^k}^*)^l$ can be thought of as the set of equivalence classes of $F_{q^k}^*$ with the equivalence relation

$$a \equiv b \;\Leftrightarrow\; \exists c \in \mathbb{F}_{q^k}^* \text{ with } a = bc^l \,.$$

We call this *equivalence modulo l.th powers.*
Thus, the Tate pairing is only defined up to a multiple by an l.th power in $\mathbb{F}_{q^k}^*$. To obtain a unique value, we have to raise the value of the Tate pairing to the power $(q^k - 1)/l$ and will so eliminate all multiples of order l.

Theorem 1.28 *The Tate pairing satisfies the following properties:*

- *Bilinearity:*

$$\begin{aligned} t_l(P_1 + P_2, Q) &= t_l(P_1, Q)t_l(P_2, Q) \\ &\text{and} \\ t_l(P, Q_1 + Q_2) &= t_l(P, Q_1)t_l(P, Q_2) \,. \end{aligned}$$

for all $P, P_1, P_2 \in E(\mathbb{F}_q)[l]$ and all $Q, Q_1, Q_2 \in E(\mathbb{F}_{q^k})[l]$. So for any $a \in \mathbb{Z}$ we have

$$t_l(aP, Q) = t_l(P, aQ) = t_l(P, Q)^a \,.$$

- *Non-degeneracy: If* $t_l(P,Q) = 1\ \forall Q \in E(\mathbb{F}_{q^k})[l]$*, then* $P = \mathcal{O}$*. Conversely, for each* $P \neq \mathcal{O}\ \exists Q \in E(\mathbb{F}_{q^k})[l]$ *so that* $t_l(P,Q) \neq 1$*.*

- *Compatibility: Let* $l = hl'$*. If* $P \in E(\mathbb{F}_q)[l]$ *and* $Q \in E(\mathbb{F}_{q^k})[l']$*, then* $t_{l'}(hP,Q) = t_l(P,Q)^h$*.*

With the above notations, we recite the following result from [BKLS02], where a proof of this lemma can be found.

Lemma 1.29 *Let* $n = \#E(\mathbb{F}_q)$ *and* q*,* k *defined as above. The value* $q-1$ *is a factor of* $(q^k-1)/r$ *for any factor* r *of* n *for a supersingular elliptic curve over* $\mathbb{F}_q$ *with security multiplier* $k > 1$*.*

As stated in [IT03], this result can be used for ordinary elliptic curves, too.

Chapter 2

Computation of the Weil/Tate Pairing

Computing the Weil/Tate pairing is a costly process. When the pairings were used first, the best known algorithm was exponential in the size of the input. In [Mil86], Miller gives an algorithm for this computation which is linear in the size of the input. In the first section of this chapter we will explain how this algorithm works and present additional technique which is needed during the algorithm to compute the values of the pairings.
The second part of this chapter deals with security aspects of the Weil and Tate pairing. A crucial part of this is the size of the groups, so no known methods can be used to compute the Tate pairing without knowing all the parameters.
Since 1986, when Miller published his algorithm, numerous enhancements have been emerged. Many of them only work on specific types of elliptic curves, e.g. on supersingular elliptic curves. In the third part of this chapter, however, we will only describe and explain enhancements which apply to ordinary elliptic curves.

2.1 Miller's Algorithm

In this section, we will explain Miller's algorithm.
As we have seen in the definitions 1.25 and 1.27, for computing the Tate and Weil pairing we need to find the function f_P and then evaluate its value at $\mathcal{A}_Q$ (for the Weil pairing, of course, we also need the function f_Q and its value at $\mathcal{A}_P$).
By doing these steps separately, we need to evaluate and compute functions of large degrees.
Instead, Miller's algorithm uses a "double and add" algorithm (also called

binary method) for elliptic curve point multiplication with an evaluation of points on the curve on lines which appear during the addition of points. In this algorithm, we denote by TDBL an algorithm for point doubling and updating the value f , and by TADD an algorithm for point addition and updating f. TDBL and TADD will be discussed later in this section. More on Miller's algorithm can be found in [Mil86], [IT03] and [GHS02]. We show a high level description of Miller's algorithm in algorithm 1.
The input for the algorithm are two points $P \in E(\mathbb{F}_p)[l]$ and $Q \in E(\mathbb{F}_{p^k})[l]$, where p is prime, l is coprime to p, k is a positive integer with $l|(q^k - 1)$ and $l \not|(q^s - 1)$ for $0 < s < k$. Usually we choose l as a prime divisor of $\#E(\mathbb{F}_p)$. We also know the binary representation of l: $l = \sum_{i=0}^{m} b_i 2^i$.
The output of the algorithm is the value $f = f_P(\mathcal{A}_Q) \in \mathbb{F}_{q^k}$. To get a unique value of the Tate pairing of the points P and Q, we need to raise f to the power of $(q^k - 1)/l$, thus eliminating all l.th powers.

Algorithm 1 Basic outline for Miller's algorithm

Input: The points $P \in E(\mathbb{F}_p)[l]$, $Q \in E(F_{p^k})[l]$ and an integer l with its binary representation $l = \sum_{i=0}^{m} b_i 2^i$
Output: $f_P(\mathcal{A}_Q) \in \mathbb{F}_{p^k}$.
 for $i := m - 1, \cdots, 1, 0$ **do**
  compute TDBL
  if $b_i = 1$ **then**
   compute TADD
  end if
 end for
 return $f_P(\mathcal{A}_Q)$

2.1.1 TADD

TADD is an algorithm for computing the addition of two (non-equal) points on elliptic curves, determining the lines which occur during this addition and evaluating the lines at certain points.

For this algorithm, recall the addition of the points P,Q on the elliptic curve as described in section 1.2. Now let l_1 be the line through P and Q, and let R_1 be the third point on the curve and the line l_1.
Let l_2 be the line between R_1 and $\mathcal{O}$ (i.e. a vertical line, if $R_1 \neq \mathcal{O}$) and $R_2 = P + Q$ the third point on the curve and the line l_2.

We can think of l_1 and l_2 as functions on the curve with the corresponding divisors

$$
\begin{aligned}
(l_1) &= (P) + (Q) + (R_1) - 3(\mathcal{O}) \\
(l_2) &= (R_1) + (R_2) - 2(\mathcal{O})
\end{aligned}
$$

One can then verify[1], that

$$((P) - (\mathcal{O})) + ((Q) - (\mathcal{O})) = (R_2) - (\mathcal{O}) + (l_1/l_2)\,. \tag{2.1.1}$$

We can also give equations for these lines. We use the Jacobian coordinate system and use $T = (X_1, Y_1, Z_1)$, $P = (X_2, Y_2, 1)$.

For the addition $T + P = (X_3, Y_3, Z_3)$ with $T \neq \pm P$, those lines are

$$l_1^{add}(x, y) = Z_3(y - Y_2) - (Y_2 Z_1^3 - Y_1)(x - X_2) \tag{2.1.2}$$

$$l_2^{add}(x, y) = Z_3^2 x - X_3\,. \tag{2.1.3}$$

For an explicit derivation of the line equations from the affine line equations see appendix A.

To estimate the computing time for the coefficients of the lines, we count the amount of multiplications and additions taking place, in compliance with [IT03], and refer to multiplications and squarings in $\mathbb{F}_q$ with M and S, to multiplications and squarings in $\mathbb{F}_{q^k}$ with M_k and S_k.
The addition $T + P$ uses $8M + 3S$, and we get the coefficient $R = Y_2 Z_1^3 - Y_1$ as an intermediate result during the addition. So for the coefficients of the line equations, we only need to compute $Z_3 Y_2$, RX_2 and Z_3^2, for which we need $2M + 1S$.
The evaluation of the line equations at a point $Q \in E(\mathbb{F}_{p^k})$ takes $3kM$, since we multiply three elements from $\mathbb{F}_{p^k}$ (x and y in l_1^{add} and x in l_2^{add}) with an element from $\mathbb{F}_p$.

2.1.2 TDBL

TDBL is an algorithm for computing the point doubling of a point on an elliptic curve, determining the lines which occur during this doubling and evaluating these lines at certain points.

For the doubling $T + T = (X_4, Y_4, Z_4)$, the first line is the tangent to the curve at T. The line equations are

$$l_1^{dbl}(x, y) = (Z_4 Z_1^2 y - 2Y_1^2) - (3X_1^2 + aZ_1^4)(Z_1^2 x - X_1) \tag{2.1.4}$$

$$l_2^{dbl}(x, y) = Z_4^2 x - X_4\,. \tag{2.1.5}$$

[1] Keep in mind that $(l_1/l_2) = (l_1) - (l_2)$.

For an explicit derivation of the line equations from the affine line equations see appendix A.

Here, the doubling $T+T$ uses $4M+6S$, and as an intermediate result we get $M=(3X_1^2+aZ_1^4)$. So for the coefficients of the line equations, we need to compute $Z_4Z_1^2$, $2Y_1^2$, MZ_1^2, MX_1 and Z_4^2, i.e. we need $3M+3S$.
The evaluation of the line equations uses, as for TADD, $3kM$.

2.1.3 Choice of the Divisor $\mathcal{A}_Q$

In the complete algorithm, we will compute $f_P(Q+S)/f_P(S)$ instead of, as defined in 1.27, $f_P(\mathcal{A}_Q)$, where $\mathcal{A}_Q$ is a divisor equivalent to the divisor $(Q)-(\mathcal{O})$, $\mathcal{A}_P$ a divisor equivalent to $(P)-(\mathcal{O})$ and $(f_P)=n\mathcal{A}_P$.
To see why we do this, let E be an elliptic curve, $P\in E(\mathbb{F}_q)[l]$, $Q\in E(\mathbb{F}_{q^k})[l]$ where l is a positive integer coprime to q (usually we choose $l\big|\#E(\mathbb{F}_q)$).
As we have seen in remark 1.21, for the evaluation of $f_P(\mathcal{A}_Q)$, (f_P) and $\mathcal{A}_Q$ need to have a disjoint support.
To guarantee this property, we introduce a random point $S\in E(\mathbb{F}_{q^k})[l]$ on the curve. Then, $(Q+S)-(S)\sim(Q)-(\mathcal{O})\sim\mathcal{A}_Q$. Hence we can write the Tate pairing as

$$t(P,Q)=\frac{f_P(Q+S)}{f_P(S)}\,. \tag{2.1.6}$$

Now for the supports of (f_P) and $\mathcal{A}_Q$ to be disjoint, $\pm S$ and $\pm(Q+S)$ must not be in the support of (f_P). This only happens with negligible probability.

Remark 2.1 *To prove that* $(Q+S)-(S)\sim(Q)-(\mathcal{O})$, *we use lemma 1.23. Since* $1(Q+S)-1(S)$ *has degree* 0, *it follows from the lemma that* $\sum[a_P]P=[1](Q+S)-[1]S=Q+S-S=Q$.

2.1.4 The Complete Algorithm

We now give the algorithm described in the previous sections in algorithm 2. We have three points $P\in E(\mathbb{F}_q)[l]$, $Q,S\in E(\mathbb{F}_{q^k})[l]$ and the binary representation of $l=\sum_{i=0}^{m}b_i2^i$ as the input (see beginning of this section or chapter 1.4.2 for more details on the choice of the parameters). The output is the same as in algorithm 1: a value f which has to be raised to the power of $(q^k-1)/l$ to get a unique value.
An example computation of Miller's algorithm for the Tate pairing can be found in appendix B.

To see that the algorithm really returns the desired result, note that at each iteration (stage i) of the algorithm, the point T is the result of the

Algorithm 2 Miller's algorithm

Input: The points $P \in E(\mathbb{F}_q)[l]$, $Q, S \in E(\mathbb{F}_{q^k})[l]$ and the binary representation of $l = \sum_{i=0}^{m} b_i 2^i$
Output: $f_P(Q+S)/f_P(S) \in \mathbb{F}_{q^k}$
set $Q' := Q+S$, $T := P$, $f := 1$
for $i := m-1, \cdots, 1, 0$ **do**
compute $T := [2]T$ and the lines l_1, l_2 from the doubling $[2]T$ as described above
compute $f := f^2 \frac{l_1(Q')l_2(S)}{l_1(S)l_2(Q')}$
if $b_i = 1$ **then**
compute $T := T+P$ and the lines l_1, l_2 from the addition $T+P$.
compute $f := f \frac{l_1(Q')l_2(S)}{l_1(S)l_2(Q')}$
end if
end for
return f

point addition $[k]P$, where k is composed from the top i bits of the binary representation of l.
Let g be a function such that $k\left((P)-(\mathcal{O})\right) = (T)-(\mathcal{O})+(g)$. Then, the value f from the algorithm is the evaluation of the function g at the divisor $\mathcal{A}_{\tilde{Q}} = (Q+S)-(S)$: $f = g\left(\mathcal{A}_{\tilde{Q}}\right)$.
At the end of the algorithm, we have $T = [l]P = \mathcal{O}$ and

$$l\left((P)-(\mathcal{O})\right) = (T)-(\mathcal{O})+(f_P) = (f_P)$$

where f_P is the function from (2.1.6), which is the function of the definition of the Tate pairing in 1.27.

Instead of calculating f as above, we can also set $f := a/b$. This way we only have multiplications during Miller's algorithm and only one division in $\mathbb{F}_{q^k}$ at the end.
We write for the addition $T+P$:

$$a = a \times l_1^{add}(Q+S) \times l_2^{add}(S) \quad (2.1.7)$$
$$b = b \times l_1^{add}(S) \times l_2^{add}(Q+S) \quad (2.1.8)$$

We need $4M_k$ for this update, so the overall computation complexity of TADD is

$$(8M+3S)+(2M+1S)+2(3kM)+4M_k = (10+6k)M+4S+4M_k$$

For the doubling $T+T$:

$$a = a^2 \times l_1^{dbl}(Q+S) \times l_2^{dbl}(S) \quad (2.1.9)$$
$$b = b^2 \times l_1^{dbl}(S) \times l_2^{dbl}(Q+S) \quad (2.1.10)$$

Here we need $4M_k$ and $2S_k$ for the update, so the overall complexity of TDBL is

$$(4M+6S)+(3M+3S)+2(3kM)+4M_k+2S_k = (7+6k)M+9S+4M_k+2S_k$$

2.2 Security Aspects for Cryptographic Applications

In this section we will emphasize general security aspects for cryptographic applications using the Weil/Tate pairing. For these applications, the security depends highly on the discrete logarithm problem:

Definition 2.2

a) Let $\mathbb{K}$ be a finite field of order q and $g \in \mathbb{K}$ be a generator of this field. For every $a \in \mathbb{K}$ there exists an integer x such that $a = g^x$. The smallest x satisfying this equation is then called the discrete logarithm on finite fields.
The discrete logarithm problem on finite fields *(DLP) is to find the discrete logarithm x for given $a, b \in \mathbb{K}$, i.e. to find x such that $b = a^x$.*

b) Let $E = E(\mathbb{K})$ be an elliptic curve over a finite field with order $\#E = q$ and with a generator point G. The discrete logarithm on elliptic curves *is $l \in [1, \cdots, q]$ such that $P = [l]G$ for $P\ inE$.*
The discrete logarithm problem on elliptic curves *(EC-DLP), consequently, is for two points $P, Q \in E(\mathbb{K})$ to find an l such that $Q = [l]P$*

With the MOV reduction [MOV93] we can reduce the discrete logarithm problem on elliptic curves to a discrete logarithm problem on a finite field. To see that, we first look at the EC-DLP $Q = [l]P$ as described above. $t(P, Q)$ is the Tate pairing (see definition 1.27) with the properties mentioned in theorem 1.28. Let $g = t(P, P)$ and $h = t(Q, P)$. Then $h = t([l]P, P) = t(P, P)^l = g^l$ because of the bilinearity. Since the Tate pairing is also non-degenerate, g and h have both order q in $\mathbb{K}$, so we have the DLP $h = g^l$.

As a result of the MOV reduction, we have to choose our groups carefully, so that both the DLP and the EC-DLP are hard to solve.
For the DLP, the field size q^k needs to be at least 1024 bits, to avoid attacks with advanced discrete logarithm solving techniques like Pohlig-Hellman or Index Calculus. On the other side it should not be too big to avoid costly computations in large extension fields.
For the EC-DLP, the best known attack is Pollard parallelizable rho method with an expected running time of $0.88\sqrt{q}$. To avoid this attack, the minimum size of l has to be 160 bits.

More information on the sizes of q^k, q, l and DL solving algorithms can be found in [LV01].

Further security considerations for pairing-based cryptosystems, e.g. relations to other hard problems of cryptography, see [Jou02].

2.3 Enhancements to Miller's Algorithm

There are numerous enhancements to Miller's algorithm. Unfortunately, most of them apply only to small subsets of elliptic curves, for example to supersingular curves. The aim of this thesis, though, is to have an algorithm which works on ordinary elliptic curves. So in this section, we only treat those enhancements that can be applied to ordinary elliptic curves. The following results are taken from [IT03], [BKLS02] and [GHS02] and we summarize the complexity of the distinct enhancement in table 4.5 in chapter 4.

2.3.1 Choice of Groups

An important observation is, that when applying Miller's algorithm for computing $t(P, Q)$ with $P \in E(\mathbb{F}_q)$ and $Q \in E(\mathbb{F}_{q^k})$, the lines l_1 and l_2 are actually elements of the smaller field $\mathbb{F}_q$. The large field $\mathbb{F}_{q^k}$ is only used for computing the value f.
To benefit from this observation, one should use efficient representations of the field $\mathbb{F}_q$ for all operations on the elliptic curve.
On this matter, we can also choose the random point $S \in E(\mathbb{F}_q)$ instead of $E(\mathbb{F}_{q^k})$ and thus reduce the number of operations in $\mathbb{F}_{q^k}$ as follows:
For TADD we still have $(8M + 3S)$ for the addition itself and $2M + 1S$ for the computation of the coefficients, but we save time during the evaluation of the lines and the update. The evaluation of $l_1^{add}(Q + S)$ still uses $2kM$, but the evaluation of $l_2^{add}(S)$ only needs $1M$. The update now has $1M_k$ each for $\tilde{a} = a \times l_1^{add}(Q + S)$ and $\tilde{b} = b \times l_2^{add}(Q + S)$ and $1M$ each for $\tilde{a} \times l_2^{add}(S)$ and $\tilde{b} \times l_1^{add}(S)$.
The same goes for $l_1^{add}(S)$ and $l_2^{add}(Q + S)$, so we have a total complexity of

$$(8M + 3S) + (2M + 1S) + (5k + 3)M + 2M_k \text{ for TADD and}$$
$$(4M + 6S) + (3M + 3S) + (5k + 3)M + 2M_k + 2S_k \text{ for TDBL.}$$

As we have seen above, we use l-torsion points for the pairings. As noted in the previous chapter, the size of l has to be at least 160 bits to ensure the security of the elliptic curve discrete logarithm problem. On the other hand, choosing a large l has computation disadvantages, so one should choose l with a size of 160 bits.

Another natural proceeding is to choose l such that its binary representation has a low Hamming weight. With such an l, the if-statement will be executed very rarely, thus reducing the total running time of the algorithm. As stated in [GHS02], these cases exist: Consider, for example, the elliptic curve E: $y^2 + y = x^3 + x + 1$ over $\mathbb{F}_{2^{283}}$. Then $\#E(\mathbb{F}_{2^{283}}) = l = 2^{283} + 2^{142} + 2^0$, and so l has Hamming weight 3.

2.3.2 Eliminating Factors in the Update

We have seen in chapter 1 that $E(\mathbb{F}_q)[l]$ are the points on the curve with order l. We choose $l \big| \#E(\mathbb{F}_q)$ and have the conditions $l \big| (q^k - 1)$ and $l \not| (q^s - 1)$ for $0 < s < k$ with the security multiplier k.
For the Tate pairing, we compute $\alpha^{(q^k-1)/l}$ with $\alpha \in \mathbb{F}_{q^k}$.
If we use the above mentioned idea that $S \in E(\mathbb{F}_q)$, we have a couple of terms where $a^{(q^k-1)/l}$ with $a \in E(\mathbb{F}_q)$ is computed.
Finally, note that $l \not| (q-1)$. Then, using Fermat's Little Theorem and lemma 1.29, we get $a^{(q^k-1)/l} = \left(a^{q-1}\right)^{\frac{q^k-1}{l(q-1)}} = 1^{\frac{q^k-1}{l(q-1)}} = 1$ for $a \in E(\mathbb{F}_q)$. This means we can discard any factor where the base is in $E(\mathbb{F}_q)$ and the exponent includes a factor $(q-1)$ during Miller's algorithm.

For the addition $T + P$ in Miller's algorithm, this would mean that we need to compute

$$\begin{aligned} a &= a \times l_1(Q+S) \\ b &= b \times l_2(Q+S)\,. \end{aligned}$$

Compared with (2.1.7) the terms $l_2(S)$ and $l_1(S)$ within the multiplication can be dropped now and we have the overall complexity

$$2M_k + (3k+10)M + 4S\,.$$

The same applies to the doubling $T + T$. Here, we only need to update

$$\begin{aligned} a &= a^2 \times l_1(Q+S) \\ b &= b^2 \times l_2(Q+S)\,. \end{aligned}$$

thus saving the terms $l_2(S)$ and $l_1(S)$ and the according multiplication. The complexity here is

$$2M_k + 2S_k + (3k+7)M + 9S\,.$$

2.3.3 Coordinate System for Points

As we have seen in (2.1.2) and (2.1.4), we need squares of the Z coordinates. So a new coordinate system for the points, which already includes this

squares would spare us some time in the computation.
This new representation (X, Y, Z, Z^2) is called the simplified Chudnovsky-Jacobian coordinate system and is referred to by $\mathcal{J}^s$.

With this system, we still need $8M + 3S$ for the addition $T + P$, $T \neq \pm P$, but we only need $2M$ for the computations of the coefficients of the lines.
For the doubling $T + T$, we still get $4M + 6S$ for the doubling itself, but we only need $3M + 1S$ for the computation of the lines.
All together, the complexity with the simplified Chudnovsky-Jacobian coordinate system is

$$(8M + 3S) + (2M) + 2(3k)M + 4M_k \text{ for TADD and}$$
$$(4M + 6S) + (3M + 1S) + 2(3k)M + 4M_k + 2S_k \text{ for TDBL.}$$

2.3.4 Evaluation of the Line Functions

During the addition $T + P$, $T \neq \pm P$, we get the lines

$$l_1(x, y) = Z_3(y - Y_2) - (Y_2 Z_1^3 - Y_1)(x - X_2) = ax + by + c \quad (2.3.1)$$
$$l_2(x, y) = Z_3^2 x - X_3 = dx + e \quad (2.3.2)$$

with $2M + 1S$ for the computation of the coefficients without the above optimization. In Miller's algorithm, we evaluate the lines at the points $Q + S$ and S.

We write $S = (x_S, y_S, 1)$ and $(Q+S) = (x_{Q+S}, y_{Q+S}, 1)$ as above and assume that $x_{Q+S}x_S$ and $y_{Q+S}y_S$ are already computed. In Miller's algorithm, the fraction $\frac{l_1(Q+S) \times l_2(S)}{l_1(Q) \times l_2(Q+S)}$ needs to be evaluated.
For the numerator of this fraction, we compute

$$l_1(Q+S) \times l_2(S) = ad(x_{Q+S}x_S) + bd(y_{Q+S}x_s) + aex_{Q+S} + bey_{Q+S} + cdx_S + ce$$

We need $2M + 1S$ to compute the coefficients a, b, c, d and e, $6M$ for the coefficients ad, bd, ae, be, cd and ce and $5kM$ for the evaluation itself.
For the denominator, we evaluate

$$l_1(S) \times l_2(Q+S) = ad(x_S x_{Q+S}) + bd(y_S x_{Q+S}) + aex_S + bey_S + cdx_{Q+S} + ce$$

Since we already know all coefficients from the numerator and $ad(x_S x_{Q+S})$, we only need $4kM$ for the evaluation.
At the end, we need $2M_k$ to combine the numerator and the denominator and update the f, so we have a total computation estimate of

$$2M_k + (9k + 16)M + 4S \text{ for the addition.}$$

For the doubling, we can apply the same technique and get a total computation estimate of

$$2M_k + 2S_k + (9k + 13)M + 9S\,.$$

2.3.5 Addition Formula for $[2^w]P$

When computing $[2^w]P$, one way to do it is to apply the doubling algorithm w times. It may be faster, though, to share intermediate results during the calculations. As proposed in [IT03], there is a way to apply such an algorithm to our current situation. The algorithm itself is listed as algorithm 10 in chapter 3.

Let $[2^i]P = (X_i, Y_i, Z_i)$ be computed by this algorithm. The lines $l_1^{(i)}$ and $l_2^{(i)}$ are recursively defined by those equations:

$$
\begin{aligned}
l_1^{(i)}(x,y) &= (Z_i Z_{i-1}^2 y - 2Y_{i-1}^2) - M_{i-1}(Z_{i-1}^2 x - X_{i-1}) \\
l_2^{(i)}(x,y) &= Z_i^2 - X_i \quad \text{with} \\
Z_i &= 2Y_{i-1}Z_{i-1} \quad \text{and} \\
M_{i-1} &= 3X_{i-1}^2 + aZ_{i-1}^4
\end{aligned}
$$

We only need $3M + 2S$ for the coefficients $Z_i Z_{i-1}^2$, $M_{i-1}Z_{i-1}^2$, $M_{i-1}X_{i-1}$ and Z_i^2, because we already computed Z_{i-1}^2.
With this algorithm we can compute the doubling step in Miller's algorithm with the update for f in $2wM_k + 2wS_k + (9k + 13)wM + (5w + 1)S$.

Chapter 3

Implementation of the Tate Pairing

For the implementation of Miller's algorithm and the Tate pairing we use Java and the FlexiProvider toolkit (see [Fle] for more information). With the ECProvider of the FlexiProvider, we already have efficient implementations of arithmetics in the field $\mathbb{F}_p$ and of elliptic curves arithmetics over $\mathbb{F}_p$ where p is prime. In this chapter, we describe the additions to the FlexiProvider made for this thesis.
In the first section, we give a general overview over the class structure we use to implement the Tate pairing and Miller's algorithm.
The necessary algorithms for arithmetics over the field $\mathbb{F}_{p^m}$ (p prime and m a positive integer) and implementation details are described in the second section.
The third section deals with elliptic curve arithmetic over $\mathbb{F}_{p^m}$ and describes the algorithms used there.
In the fourth section the implementation of the Tate pairing itself is described, and we finish with the fifth section where other classes are mentioned.

Most of the used algorithms are standardized by the IEEE Working Group P1363 and described in [IEE02].

3.1 Overview over the Class Structure

We have implemented four different subjects: Storage of and arithmetic on polynomials over $\mathbb{F}_p$, storage of and arithmetic on elements from the field $\mathbb{F}_{p^m}$, elliptic curve point arithmetic and storage for elliptic curves over $\mathbb{F}_{p^m}$ and points on this curve, and the Tate pairing itself.

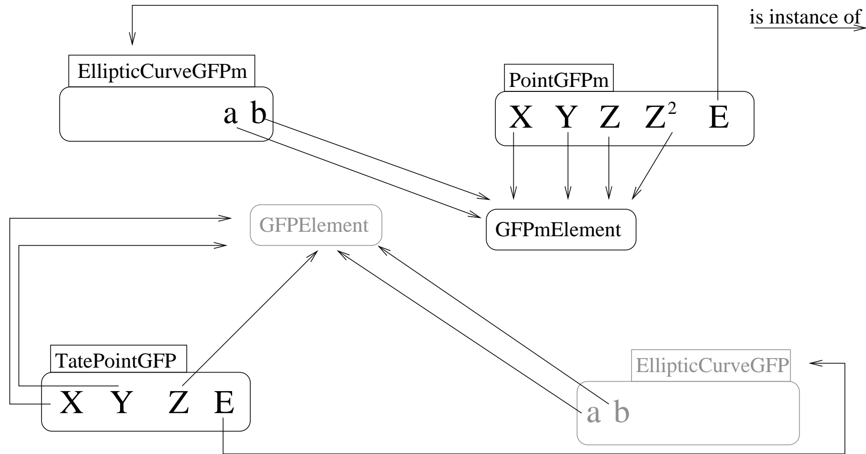

Figure 3.1: Overview of the $\mathbb{F}_{p^m}$ classes

Arithmetic over $\mathbb{F}_{p^m}$ is implemented in the class `GFPmElement`. All algorithms described in section 3.2 are implemented. Since an element $A(x) \in \mathbb{F}_{p^m}$ is represented by a polynomial (more on that topic in section 3.2), the coefficients of this polynomial are in $\mathbb{F}_p$ and hence instances of `GFPElement`. For elliptic curves over $\mathbb{F}_{p^m}$ there are two classes: `EllipticCurveGFPm` holds the curve parameters $a \in \mathbb{F}_{p^m}$ and $b \in \mathbb{F}_{p^m}$ for the elliptic curve $y^2 = x^3 + ax + b$, `PointGFPm` contains a point on the elliptic curve and the necessary arithmetics which we describe in section 3.3.

Although there already is an implementation of arithmetics on points $P \in E(\mathbb{F}_p)$ we wrote our own. This way we can benefit from the simplified Chudnovsky-Jacobian coordinates and the sharing of intermediate results from the addition and doubling during the evaluation of the line equations. For the implementation of `TatePointGFP` we use the same algorithms as for `PointGFPm`, so we do not describe this implementation in greater detail.

The Tate pairing is implemented in the class `TatePairing`, which gets its parameters from the class `TPParams`. Those classes are described in section 3.4.

For the relations between `GFPmElement`, `GFPElement`, `PointGFPm`, `TatePointGFP`, `EllipticCurveGFPm` and `EllipticCurveGFP` see figure 3.1. For the relations between the underlying classes `GFPmElement`, `GFPPolynomial` and `GFPElement` see figure 3.2. The grey areas are classes which were already present in the FlexiProvider. The black areas are classes which were added for this thesis.

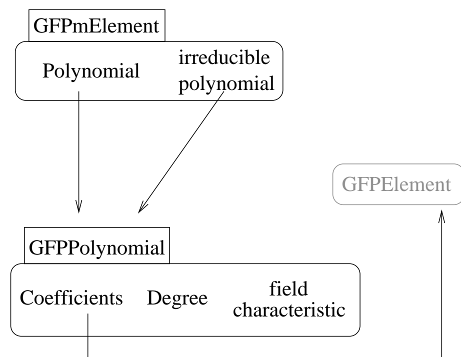

Figure 3.2: Relations between the arithmetic classes

3.2 $\mathbb{F}_{p^k}$ Fields and Arithmetic

The implementation of the fields $\mathbb{F}_{p^k}$, where $p > 3$ is prime and k is a positive integer, is straight forward. The theory we use is summarized in section 1.1 and can be found more detailed in [LN86].
In this section we will describe algorithms for addition, multiplication and inversion of elements in the field $\mathbb{F}_{p^k}$ which we need in Miller's algorithm.
For creating points on elliptic curves, we need to test whether an element $A(x) \in \mathbb{F}_{p^k}$ is a square in the field, and extract the square root, i.e. an element $B(x) \in \mathbb{F}_{p^k}$ with $B^2(x) = A(x)$. We will also give algorithms for those techniques.

3.2.1 Extension Fields

As we recall from section 1.1, finite fields $\mathbb{F}_{p^k}$ with $k > 1$ and $p > 3$ prime are called *odd characteristic extension field*. The field $\mathbb{F}_p$ is called the *subfield* and k is the *extension degree*.
Elements of $\mathbb{F}_{p^k}$ are represented by a polynomial modulo an irreducible field polynomial. Those extension fields are isomorphic to $\mathbb{F}_p[t]/(f(t))$, where $f(t)$ is a monic irreducible polynomial of degree m over $\mathbb{F}_p$

$$f(t) = t^m + \sum_{i=0}^{m-1} p_i t^i \,, \; p_i \in \mathbb{F}_p \,.$$

We use a residue class with the polynomial of the least degree in this class, so an element $A(x) \in \mathbb{F}_{p^m}$ will be represented by the *standard* (or *polyno-*

mial/canonical) basis representation

$$A(x) = a_{m-1}x^{m-1} + \cdots + a_1x^1 + a_0x^0$$

with $a_i \in \mathbb{F}_p$, $i = m-1, \cdots, 0$. All arithmetic operations are performed modulo the field polynomial.

3.2.2 Addition and Subtraction

Given the polynomial basis representation of a field element $A(x) \in \mathbb{F}_{p^m}$

$$A(x) = a_{m-1}x^{m-1} + \cdots + a_1x^1 + a_0x^0$$

with $a_i \in \mathbb{F}_p$, $i = m-1, \cdots, 0$, addition and subtraction is straightforward: simply add or subtract coefficients of the same degree. With $A(x), B(x) \in \mathbb{F}_{q^k}$, this means:

$$A(x) \pm B(x) = (a_{m-1} \pm b_{m-1})x^{m-1} + \cdots + (a_1 \pm b_1)x + (a_0 \pm b_0)\,.$$

Additions and subtractions of the coefficients are operations on the field $\mathbb{F}_p$, i.e. they are performed modulo p.

3.2.3 Multiplication

Multiplication of two elements $A(x), B(x) \in \mathbb{F}_{p^m}$, $C(x) = A(x) \times B(x)$ is performed in two steps. The first step is an ordinary polynomial multiplication with the schoolbook method (see algorithm 3). After this step, we have an intermediate polynomial $C'(x)$ of degree less or equal to $2m-2$

$$C'(x) = A(x) \times B(x) = c'_{2m-2}x^{2m-2} + \cdots + c'_1x + c'_0\,,\ c'_i \in \mathbb{F}_p\,.$$

Algorithm 3 Schoolbook multiplication

Input: $A(x), B(x) \in \mathbb{F}_{q^m}$ with $A(x) = \sum a_i x^i$, $B(x) = \sum b_j x^j$
Output: $C'(x) = \sum c_k x^k = A(x) \times B(x)$

```
for i := 0, 1, ..., m-1 do
    for j := 0, 1, ..., m-1 do
        c_{i+j} := c_{i+j} + a_i b_j mod p
    end for
end for
return C'(x) := Σ c_k x^k
```

The second step is to compute the residue $C(x) \equiv C'(x) \bmod f(x)$, where $f(x)$ is the field polynomial. Of course, $C(x)$ is again in $\mathbb{F}_{p^m}$. The reduction is done with the help of the division algorithm (see theorem 1.5) and is

given in algorithm 4. We simply do a polynomial division with remainder of $C'(x)$ and the field polynomial $f(x)$. In the algorithm, we assume that the coefficients of the according polynomials f and r are f_i and r_i.
Now we esteem the complexity of the algorithm. We use one multiplication of elements in $\mathbb{F}_{p^k}$. Since there is no reduction after the multiplication, we have complexity k^2M for this step. The other operations are negligible. According to [Buc99], the while loop will be traversed $\deg(q)+1$ times, where q is a polynomial with $C(x) = q(x)f(x)+r(x)$. To estimate the maximum degree of q, we note that in the first iteration $\deg(q(x)) = \deg(C'(x)) - \deg(f(x))$. Since $\deg(C'(x)) \leq 2k-2$ and $\deg(f(x)) = k$, $\deg(q(x)) \leq k-2$.
Combining these results, we have a complexity result of $(k-1)k^2M$ for the reduction.

Algorithm 4 Reduction of elements

Input: A polynomial $C'(x)$ and the field polynomial $f(x)$
Output: $C(x) \in \mathbb{F}_{p^m}$ modulo $f(x)$
  $r(x) := C'(x)$, $q(x) := 0$
  while $r(x) \neq 0$ and $deg(r) > deg(f)$ **do**
    $t := \frac{r_{\deg(r)}}{f_{\deg(f)}}$
    $h(x) := tx^{\deg(r)-\deg(f)}$
    $r(x) := r(x) - h(x)g(x)$
    $q(x) := q(x) + h(x)$
  end while
  return $r(x)$

3.2.4 Inversion

For computing the multiplicative inverse of a field element $A(x) \in \mathbb{F}_{p^m}$ we use the Extended Euclidean Algorithm for computing the greatest common divisor. This algorithm computes for given $A(x), B(x) \in \mathbb{F}_{p^m}$ the result $g = gcd(A(x), B(x)) = w(x)A(x) + v(x)B(x)$, $g \in \mathbb{F}_p$.
To invert an element $A(x)$ in the field $\mathbb{F}_p[x]/(f)$, where $f(x)$ is an irreducible polynomial over $\mathbb{F}_p$, we compute

$$g = gcd(A(x), f(x)) = w(x)A(x) + v(x)f(x)\,. \tag{3.2.1}$$

Since $f(x)$ is irreducible, the value g is a unit in $\mathbb{F}_p$, so to obtain the inverse element $w(x)$, we have to multiply (3.2.1) with g^{-1}

$$gg^{-1} = 1 = gcd(A(x), f(x)) = \overbrace{w(x)g^{-1}}^{=A^{-1}(x)} A(x) + g^{-1}v(x)f(x)\,.$$

So $A^{-1}(x) = g^{-1}w(x)$, our desired result. The Extended Euclidean Algorithm is given in algorithm 5.

Algorithm 5 Extended Euclidean Algorithm

Input: $A(x) \in \mathbb{F}_{p^m}$ and the field polynomial $f(x)$
Output: $A^{-1}(x)$ such that $A(x)A^{-1}(x) = 1$
$w_0(x) := 1$, $w_1(x) := 0$
$v_0(x) := 0$, $v_1(x) := 1$
sign:= 1
while $b \neq 0$ **do**
$r(x) := A(x) \bmod f(x)$
$q(x) := A(x)/f(x)$
$A(x) := f(x)$
$f(x) := r(x)$
$w_{\text{temp}} := w_1(x)$, $v_{\text{temp}} := v_1(x)$
$w_1(x) := q(x)w_1(x) + w_0(x)$
$v_1(x) := q(x)v_1(x) + v_0(x)$
$w_0(x) := w_{\text{temp}}$, $v_0(x) := v_{\text{temp}}$
sign:= −sign
end while
Let $a \in \mathbb{F}_p$ be the constant coefficient of $A(x)$. All other coefficients of $A(x)$ should be 0 by now.
$w_0(x) := \text{sign} \times w_0(x)$, $v_0(x) := -\text{sign} \times v_0(x)$
return $a^{-1} \times w_0(x)$

3.2.5 Exponentiation

For exponentiation of field elements, we use the fast exponentiation algorithm given in algorithm 6. It takes a field element $A(x) \in \mathbb{F}_{p^m}$ and an exponent $e \in \mathbb{N}$ as a bit string as the input and outputs $A^e(x) \in \mathbb{F}_{p^m}$.

3.2.6 Square Test

For the square test, consider Fermat's little theorem

Theorem 3.1 *Let $a \in \mathbb{F}_q$ and $gcd(a, q) = 1$. Then $a^{(q-1)} \equiv 1 \bmod q$.*

To test whether an element $A(x) \in \mathbb{F}_{p^m}$ is a square, we simply compute

$$B(x) = A(x)^{\frac{p^m-1}{2}} .$$

The result is either ± 1 or 0, since $A(x)^{p^m-1} \equiv 1$ in $\mathbb{F}_{p^m}$ and so $A(x)^{\frac{p^m-1}{2}} \equiv \pm 1$ in $\mathbb{F}_{p^m}$. If the result is 1, $A(x)$ is a square since the equivalence relation from Fermat's little theorem is satisfied. If the result is -1, $A(x)$ is no square. The result is 0 if and only if $A(x) = 0$; in this case $A(x)$ is a square, too.

Algorithm 6 Fast exponentiation

Input: $A(x) \in \mathbb{F}_{p^m}$ and exponent $e \in \mathbb{N}$ as a bit string $e = \sum_{i=0}^{m-1} e_i 2^i$ with $e_i \in \{0, 1\}$
Output: $B(x) = A^e(x)$

$B(x) := 1$
for $i := 0, 1, \cdots, m-1$ **do**
  if $e_i = 1$ **then**
    $B(x) := B(x) \times A(x)$
  end if
  $A(x) := A(x) \times A(x)$
end for
return $A(x)$

3.2.7 Square Root Extraction

To compute the square root of an element $A(x) \in \mathbb{F}_{p^m}$, we use algorithm 7, taken from [IEE02]. This algorithm is based on the Tonelli algorithm, which is explained in greater detail in [Buc]. Here, we only want to mention the following

Lemma 3.2 *Let G be a finite cyclic group of known order $|G| = 2^e k$, k odd. Let g be a square in G and $y \in G$. Let i be an even integer such that the order of (gy^{-e}) is odd. Then*

$$\sigma = \left(gy^{-e}\right)^{(k+1)/2} y^{e/2}$$

is a square root of g.

3.3 Elliptic Curve Arithmetic over $\mathbb{F}_{p^k}$

In this implementation, we are using the elliptic curve

$$E: \; y^2 = x^3 + ax + b \,, \; a, b \in \mathbb{F}_{p^k} \,. \tag{3.3.1}$$

Since we are using more point doublings than point additions in Miller's algorithm, we are choosing the simplified Chudnovsky-Jacobian coordinate system for point representation, where a point on a curve is (X, Y, Z, Z^2). As stated in 1.2.2 based on [IT03] and [CMO98], point doubling is less expensive in the Jacobian coordinate system than in the affine coordinate system. Since we use point doubling in every iteration of Miller's algorithm, and we can optimize the number of point additions, the simplified Chudnovsky-Jacobian coordinate system is the best suited for this task.

Algorithm 7 Square root extraction

Input: $A(x) \in \mathbb{F}_{p^m}$
Output: $B(x) \in \mathbb{F}_{p^m}$ with $B^2(x) = A(x)$ or the message "no square roots exist"
  Choose a random $Y(x) \in \mathbb{F}_{p^m}$ until $Y(x)$ is no square
  Write $p^m - 1 = k2^e$. k is odd
  $Y(x) := Y^k(x)$
  $C(x) := A^k(x)$
  $Z(x) := A^{\frac{k+1}{2}}(x)$
  if $C(x)^{2^{e-1}}(x) \neq 1$ **then**
    return "no square roots exist"
  end if
  while $C(x) \neq 1$ **do**
    Let $i \in \mathbb{N}$ be the smallest number such that $C^{2^i}(x) = 1$
    $C(x) := C(x) \times Y^{2^{e-i}}$
    $Z(x) := Z(x) \times Y^{2^{e-i-1}}$
  end while
  return $Z(x)$

Above curve equation in Jacobian coordinates is

$$E_J: \; Y^2 = X^3 + aXZ^4 + bZ^6\,. \tag{3.3.2}$$

In the following subsections, we will describe the point addition and doubling on the curve as well as the generation of points and l-torsion points on the curve.

3.3.1 Point Addition

For computing the point addition $P + Q = R$, $P = (X_1, Y_1, Z_1, Z_1^2)$, $Q = (X_2, Y_2, Z_2, Z_2^2)$, $R = (X_3, Y_3, Z_3, Z_3^2)$, we are using the addition formula given in (1.2.8):

$$\begin{aligned}
X_3 &= -8X_1Y_1^2 + (3X_1^2 + aZ_1^4)^2 \\
Y_3 &= -8Y_1^4 + (3X_1^2 + aZ_1^4)(4X_1Y_1^2 + 8X_1Y_1^2 - (3X_1^2 + aZ_1^4)^2) \\
Z_3 &= 2Y_1Z_1Z_2
\end{aligned}$$

and rewrite them so that intermediate results are shared between the computations of X_3, Y_3 and Z_3. The result is shown in algorithm 8, which gives an efficient implementation of the point addition with 8 multiplications and 6 squarings in $\mathbb{F}_{p^k}$.
The two if clauses in this algorithm catch the case that the result is the point at infinity and that the points P and Q are the same. The first case

is true if $X_1 = X_2$ and $Y_1 \neq Y_2$, or using the variables we introduced in the algorithm: $H = 0$ and $R \neq 0$. The second case is true if $X_1 = X_2$ and $Y_1 = Y_2$, or $H = 0$ and $R = 0$.
We are using two temporary variables T_1 and T_2 during the computation for storing results we need in the other steps.
Otherwise, the algorithm is a straight forward implementation of above mentioned addition formula.

Algorithm 8 Point addition

Input: Two points $P, Q \in E(\mathbb{F}_{p^k})$, $P \neq Q$
Output: $R = P + Q$ or "P and Q are the same points"
  $U_1 := X_1 Z_2^2$, $S_1 := Y_1 Z_2^3$
  $U_2 := X_2 Z_1^2$, $S_2 := Y_2 Z_1^3$
  $H := U_2 - U_1$, $R := S_2 - S_1$
  if $H = 0$ **then**
    if $R = 0$ **then**
      return "P and Q are the same points"
    end if
  else
    return $\mathcal{O}$
  end if
  $T_1 := H^2$, $T_2 := U_1 T_1$
  $T_1 := T_1 H$
  $X_3 := -T_1 - 2T_2 + R^2$
  $Y_3 := -S_1 T_1 + R(T_2 - X_3)$
  $Z_3 := Z_1 Z_2 H$
  return (X_3, Y_3, Z_3, Z_3^2)

3.3.2 Point Doubling

For point doubling, i.e. $P + P = (X_1, Y_1, Z_1, Z_1^2) + (X_1, Y_1, Z_1, Z_1^2) = (X_4, Y_4, Z_4, Z_4^2)$, we use the formulas (1.2.9):

$$
\begin{aligned}
X_4 &= -8X_1Y_1^2 + (3X_1^2 + aZ_1^4)^2 \\
Y_4 &= -8Y_1^4 + (3X_1^2 + aZ_1^4)(4X_1Y_1^2 + 8X_1Y_1^2 - (3X_1^2 + aZ_1^4)^2) \\
Z_4 &= 2Y_1Z_1 .
\end{aligned}
$$

Algorithm 9 gives an efficient implementation of the point doubling with 4 multiplications and 6 squarings.

Algorithm 9 Point doubling

Input: A point $P \in E(\mathbb{F}_{p^k})$
Output: $R = P + P$
  $M := 3X_1^2 + aZ_1^4$
  $S := 4X_1Y_1^2$
  $X_4 := M^2 - 2S$
  $Y_4 := M(S - X_4) - 8Y_1^4$
  $Z_4 := 2Y_1Z_1$
  return (X_4, Y_4, Z_4, Z_4^2)

3.3.3 Efficient Iterated Point Doubling

As already mentioned in 2.3.5, when computing $[2^w]P$ it may be faster to share intermediate results during the calculations.
Algorithm 10 gives such a method for our case. The input is a point $P = (X, Y, Z)$ and a positive integer w, and the algorithm outputs the point $P_w = [2^w]P$.
This algorithm requires $4w$ multiplications and $(4w + 2)$ additions.

Algorithm 10 Iterated EC double algorithm

Input: $P_0 = (X_0, Y_0, Z_0)$, w
Output: $P_w = [2^w]P_0 = (X_w, Y_w, Z_w)$
  $W_0 := aZ_0^4$, $M_0 := 3X_0^2 + W_0$ $S_0 := 4X_0Y_0^2$
  $X_1 := M_0^2 - 2S_0$, $Y_1 := M_0(S_0 - X_1) - 8Y_0^4$, $Z_1 := 2Y_0Z_0$
  for $i = 1, 2, \cdots, w - 1$ **do**
    $W_i := 2(8Y_{i-1}^4)W_{i-1}$
    $M_i := 3X_i^2 + W_i$
    $S_i := 4X_iY_i^2$
    $X_{i+1} := M_i^2 - 2S_i$
    $Y_{i+1} := M_i(S_i - X_{i+1}) - 8Y_i^4$
    $Z_{i+1} := 2Y_iZ_i$
  end for
  return $P_w = (X_w, Y_w, Z_w)$

3.3.4 Generation of Random and l-torsion Points on the Curve

For generating random points on the curve we make use of the curve equation

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

We choose a random value $x \in \mathbb{F}_{p^m}$, compute the right side of the equation and then use the square test from section 3.2.6 to determine if the right side

is a square in $\mathbb{F}_{p^m}$. If the answer is no, we choose a new x until we have found a square. If the right side is a square, we compute this square root via the method in section 3.2.7 and use the result as the y coordinate. The affine representation of the point is then (x, y) and we obtain the Jacobian representation by setting $z = 1$, thus resulting in the point $(x, y, 1)$.

To generate l-torsion points, we need the order of the elliptic curve, i.e. the number $\#E(F_q) = n$, and $l|n$.
The basic algorithm is quite simple:

a) Let $\#E(F_q) = lr = n$.

b) Generate, with above method, a random point Q on the curve.

c) If $P = [r]Q = \mathcal{O}$, go back to step b).

d) Now, the point P should be l-torsion. To verify this, we test, if $[l]P = \mathcal{O}$ and print out an error message if the condition does not hold.

e) Otherwise, P is the l-torsion point.

This proceeding has one drawback. If $\#E(\mathbb{F}_q) = n = lr$ is large, and l is only a relatively small fraction of n, the algorithm will have a large running time.
This is usually the case if we choose l-torsion points from $E(\mathbb{F}_{p^k})$, since l is about 160 bits in size and $\tilde{n}$ is much larger than l.
So if we want to find l-torsion points on $E(\mathbb{F}_{p^k})$ we modify our algorithm described above:
Let $\#E(\mathbb{F}_q) = n$, $l|n$. Then set v to be the largest integer such that $l^v|n$. If $v = 1$, we have the same algorithm as above If $v > 1$, however, we only have to compute $P = [n/l^v]Q \neq \mathcal{O}$, which is easier and faster.
The algorithmic description of this procedure is given in algorithm 11. Both above algorithm and the modification can be found in [IEE00].
An easy way to compute the group order $\#E(\mathbb{F}_{p^k})$ is given in section 1.2.4.

3.4 Tate Pairing

In this section we describe the implementation of the Tate pairing itself.

Since the necessary algorithms are already given in chapter 2, we only describe the way the functions are divided in the different classes, how to use the Tate pairing and how to create the necessary points for the pairing.

First, there is the class `TPParams`, which holds all the necessary parameters for the pairing, such as the elliptic curves over $\mathbb{F}_p$ and $\mathbb{F}_{p^m}$, the points P,

Algorithm 11 Generating a l-torsion point

Input: An elliptic curve E and integers l, r such that $\#E = lr$.
Output: a l-torsion point P or the message "wrong group order, no l-torsion point found".

Determine v such that $l^v | \#E$.
Set $h := \frac{\#E}{l^v}$ and choose a random point $Q \in E$
$P := [h]Q$
while $P = \mathcal{O}$ **do**
  Choose a random point $Q \in E$
  $P := [h]Q$
end while
Set $\alpha := 0$
while $\alpha < v$ **do**
  $\alpha := \alpha + 1$, $T := P$, $P := [l]P$
  if $P = \mathcal{O}$ **then**
    return T
  end if
end while
return "wrong group order, no l-torsion point found".

Q and S used during the pairing, the field characteristic p, the extension degree m, the number of points on the curve $\#E(\mathbb{F}_p) = n$ and a prime divisor l of n which satisfies $l|(q^m - 1)$ and $l \not| (q^s - 1)$ for $0 < s < m$. After creating an instance of `TPParams`, the function `init()` generates the points P, Q and S. Alternatively, one can also set the points manually by using the functions `setP, setQ` and `setS`.

With these parameters, the class `TatePairing` can be initialized. This class implements Miller's algorithm with the various enhancements described in chapter 2. For the creation of an instance of `TatePairing`, the constructor functions needs an instance of `TPParams` as input. To compute the Tate pairing of the points P and Q generated (or set) in the instance of `TPParams`, one can simply call the function `compute()`. The result is the Tate pairing value.

Example 3.3 *The following is an example on how to use the above described functions and classes to compute the Tate pairing for the parameters described in appendix B:*
$E : y^2 = x^3 + x$, $p = 43$, $l = 11$, $k = 2$, $P = (23, 8, 1)$ *and* $Q = (20, 8t, 1)$.

```
BigInteger p = new BigInteger("43");
BigInteger l = new BigInteger("11");
BigInteger order = p.add(BigInteger.ONE);
BigInteger a = BigInteger.ONE; BigInteger b = BigInteger.ZERO;
```

```
GFPPolynomial irreducible = new GFPPolynomial(extdegree, p);
GFPElement[] irredValues = {
  toGFP(new BigInteger("1")),
  toGFP(new BigInteger("0")),
  toGFP(new BigInteger("1"))
};
irreducible.assign(irredValues);
irreducible.setIrredPolynomial(irreducible);
TPParams params = new TPParams(a, b, p, extdegree, irreducible,
                               l, order, true);

GFPmElement Qx = new GFPmElement(params.getEPm().getA());
GFPmElement Qy = new GFPmElement(params.getEPm().getA());
Qx.assign(new GFPElement(BigInteger.valueOf(20L)), 0);
Qy.assign(new GFPElement(BigInteger.valueOf(8L)),  1);

PointGFPm Q = new PointGFPm(Qx, Qy, params.getEPm());
TatePointGFP P = new TatePointGFP(BigInteger.valueOf(23L),
                                  BigInteger.valueOf(8L),
                                  params.getEP());
TatePairing pairing = new TatePairing(params, P, Q);
GFPmElement result = pairing.compute();
```

Chapter 4

Timing Results

In this chapter, we summarize the complexity results from chapter 2 in a table and give timing results for our implementations of arithmetics over finite extension fields, point arithmetics and the Tate pairing.
All tests were running on a notebook with a Pentium M 1400 MHz processor and 512 MB RAM using Windows XP and JDK 1.4.1.

For our tests with ordinary curves, we use the following curve parameters from [DEM02]:

$$\begin{aligned} k &= 7 \\ p &= 2228021501991753969207603720194256465687 7 \text{ (135 bit)} \\ l &= 20994281098551570014 9 \text{ (68 bit)} \\ order &= 2228021501991753969177750603090732726827 2 \\ a &= 2008148572763713778628194731374451917319 3 \\ b &= 1934857596354367048435058401767850401196 5 \end{aligned} \tag{4.0.1}$$

order is the order of the curve $E : y^2 = x^3 + ax + b$ over $\mathbb{F}_p$, l is the largest prime divisor of *order*.
These parameters are not secure enough for the usage in a cryptosystem. p is 135 bit long, therefore p^k only has 939 bits. According to section 2.2, however, we need at least 1024 bits for p^k to avoid known DLP solving techniques. Even worse is the size of l: We need at least 160 bits to ensure that the EC-DLP is hard to solve, the given l here only has 68 bits.
Here we see one of the biggest problems when using ordinary curves for pairing-based cryptosystems: the choice of the parameters. The other curves in [DEM02] are too large for a comparison with other computation results. For example, the next larger curve has extension degree 10 and a 428 bit prime p, which would result in a 4280 bit field $\mathbb{F}_{p^{10}}$. In such a large field, computations take very long and are not comparable with other timing

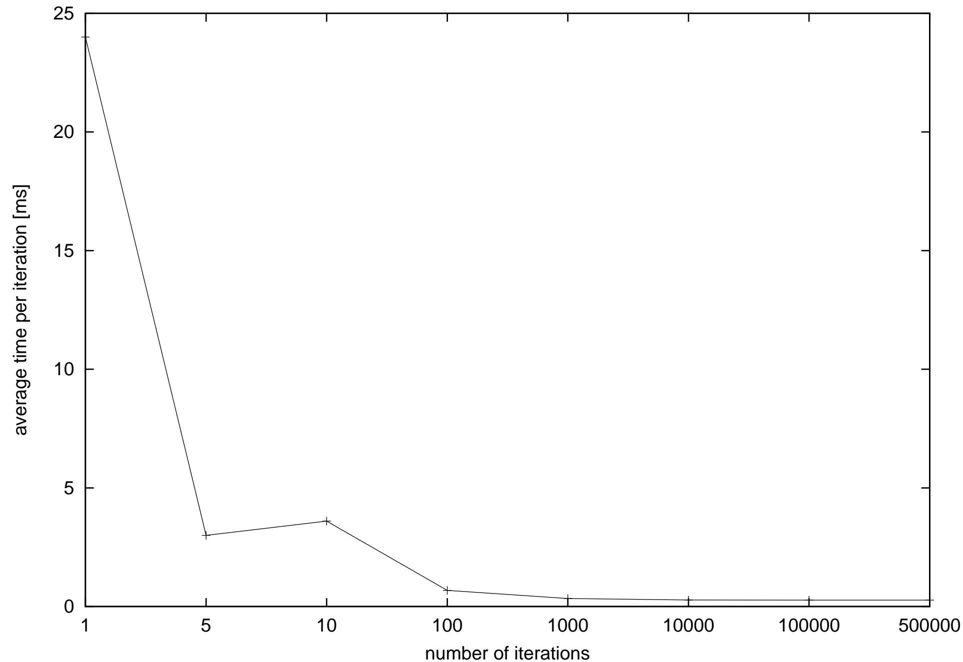

Figure 4.1: Java timings for calling the same function repeatedly

results. But since finding parameters exceed the scope of this thesis, we restrict to the above given parameters.

Before we give our timing results, we need to say a few words to the speed of Java itself. Java was designed to be platform independent, not necessarily fast. When comparing the timing results in this chapter, one has to keep that in mind. We can never reach the timings from programs written in C/C++ and compiled on a specific platform.
As it turns out we cannot reach them even close and it's hard to predict how long a computation may take. For example, consider the following situation: We write a timing test where we call EC doubling repeatedly, first only once, then five times, then 10 times and so on. The timing results for this test are shown in figure 4.1. As we can see, the first call of EC doubling takes 24 ms, and as the numbers of iterations increase, the time per iteration converge at 0.28 ms. This behavior makes it hard to predict how much computing time an operation may take exactly.
The reason for this behavior could be a technique Java calls "adaptive compiling". With this technique, the Java compiler tries to discover performance bottlenecks, i.e. code that is used a lot of times, and optimizes the compilation of those parts, while seldom-used code is only treated with the normal

compiler.
These techniques make it very hard to predict the timing results of a fixed operation, as figure 4.1 shows.

4.1 Timing Results of the Underlying Arithmetics

In this section we show the timing results of arithmetics in $\mathbb{F}_{p^k}$ and on points in $E(\mathbb{F}_p)$ and $E(\mathbb{F}_{p^k})$. In order to compare our results with the ones of other implementations, we also compile timing results for supersingular curves.

4.1.1 Timing Results of the Underlying Arithmetics for Ordinary Curves

For computations in extension fields we need an irreducible polynomial. The size of this polynomial is crucial to the computation time needed for arithmetics in the extension field.
To our knowledge, there is no easy way to find such a polynomial, so we wrote a little program that tests some combinations of trinomials $f(x) = x^7 + i_0\, x^m + i_1$ with $i_0, i_1 \in \mathbb{F}_p$ and $7 < m < 0$. As a result from [LN86], we know for our field $\mathbb{F}_p$ and for extension degree 7 there are $\frac{1}{7}(p^7 - p)$ irreducible polynomials, so chances are good we find a small polynomial pretty soon.
As it turned out, after a very short running time the irreducible polynomial

$$f(x) = x^7 + x + 14$$

was found.

For all timings results we use the parameters given in (4.0.1) at the beginning of this chapter.
In table 4.1 are the timing results for arithmetics in $\mathbb{F}_{p^7}$. We compute 50,000 operations each and show the average time per operation in the table. As one can see, additions only take negligible time, while divisions are about 5 times more expensive than multiplications. To take account of this fact, we substituted the divisions in Miller's algorithm with multiplications and one division at the end.
Efficient multiplications in the extension field are important because we use them frequently during Miller's algorithm.
For the exponentiation result, we used the same exponent $(p^7 - 1)/l$ as we use at the final exponentiation of the Tate pairing. The exponent is 894 bit long and has a Hamming weight of 430, so we have to compute 894 squarings and 430 multiplications in $\mathbb{F}_{p^7}$ for the exponentiation.

In table 4.2 we show the timing results for the average computing time of point additions, doublings and generations of points over the fields $\mathbb{F}_p$ and

Operation	Average time
Addition	0.0201 ms
Multiplication	0.569 ms
Division	3.619 ms
Exponentiation	755.88 ms

Table 4.1: Timing results for arithmetics in $\mathbb{F}_{p^7}$

Operation	Timings $E(\mathbb{F}_p)$	Timings $E(\mathbb{F}_{p^7})$
Point generation	2.100 ms	6406.2 ms ms
Addition	0.107 ms	4.306 ms
Doubling	0.103 ms	4.172 ms

Table 4.2: Timing results for arithmetics with ordinary curves

$\mathbb{F}_{p^7}$. We compute 100,000 respective 50,000 operations each with the same points and give the average time per operation in the table.
Since point doublings are used in every step of Miller's algorithm, it is vital to have an efficient implementation for an efficient evaluation of the Tate pairing. With an average of 0.103 ms, this has been achieved.
The timing result for point generation is not that important for our use of the pairings, since points are only computed once during the setup. But there may be other usages where efficient point creation is needed, so we give the time for point generations, too.
Arithmetics over $\mathbb{F}_{p^7}$ are only used in the setup phase of the Tate pairing computation. Once we enter the main loop, no arithmetics on curves over $\mathbb{F}_{p^7}$ will be done anymore, so the timings for those operations have no influence on the computing time of the Tate pairing.

4.1.2 Timing Results of the Underlying Arithmetics for Supersingular Curves

We use the supersingular curve

$$E : y^2 = x^3 + x$$

over a field $\mathbb{F}_{p_S}$ with a prime $p_S > 3$, $p_S \equiv 3 \bmod 4$. The curve order then is $\#E(\mathbb{F}_{p_S}) = p_S + 1$ and we can obtain an l_S-torsion point on $E(\mathbb{F}_{p_S^2})$ from an l_S-torsion point on $(x, y) = P \in E(\mathbb{F}_{p_S})$ with the distortion map $\phi(x, y) = (-x, iy)$, where $i \in \mathbb{F}_{p_S^2}$ and $i^2 = -1$. These parameters are given as an example in [BKLS02] and [Jou02].
The only problem left is to find a prime p_S with at least 512 bits in size such

Operation	Average time
Addition	0.009 ms
Multiplication	0.163 ms
Division	2.844 ms
Exponentiation	195.08 ms

Table 4.3: Timing results for arithmetics in $\mathbb{F}_{p_S^2}$

Operation	Timings $E(\mathbb{F}_{p_S})$	Timings $E(\mathbb{F}_{p_S^2})$
Point generation	56.491 ms	4094.9 ms
Addition	0.661 ms	1.234 ms
Doubling	0.590 ms	1.203 ms

Table 4.4: Timing results for arithmetics with supersingular curves

that $l_S | p_S + 1$. A good approach is to rewrite this condition to $p_S = l_S k - 1$, find a prime l_S with about 160 bits in size and then repeatedly try values k with about 352 bits in size until we find a prime p which also satisfies $p_S \equiv 3 \bmod 4$.
With this method, we found the following parameters:

$$\begin{aligned} l_S &= 1332798160418053653818450729169417606839252697287 \text{ (160 bits)} \\ k &= 21417346072854556656144864201530338610971045588546641641555 \\ &\quad 39536909022889796653498272774200458078090198661 2 \\ p_S &= l_S k - 1 = 285449994469373788397767217592479043842225040909307 \\ &\quad 326588684768129551045984771731043699990459546899486271499948 \\ &\quad 5875296623107450777270913076391289776272164 3 \text{ (514 bits)} \end{aligned} \tag{4.1.1}$$

Those parameters were used to compile the timing results given in table 4.3 for arithmetics in $\mathbb{F}_{p_S^2}$ and in table 4.4 for elliptic curve operations.

4.2 Timing Results for Miller's Algorithm

In table 4.5 we first give a summary of the complexity for the different enhancements given in chapter 2.
For a better comparison of these enhancements, we estimate the complexity time for a multiplication in the extension field and convert it to the number of multiplications in the underlying field.

No.	Enhancement	TADD	TDBL
1	no enhancement (see chapter 2.1.4)	$(10+6k)M+4S$ $+4M_k$	$(7+6k)M+9S$ $+4M_k+2S_k$
2	$S \in E(\mathbb{F}_p)$ (see chapter 2.3.1)	$(13+5k)M+4S$ $+2M_k$	$(10+5k)M+9S+2$ M_k+2S_k
3	2 and elim factors in $\mathbb{F}_p$ (see chapter 2.3.2)	$(10+3k)M+4S$ $+2M_k$	$(7+3k)M+9S$ $+2M_k+2S_k$
4	Coordinate system (see chapter 2.3.3)	$(10+6k)M+3S$ $+4M_k$	$(7+6k)M+7S$ $+4M_k+2S_k$
5	3 and 4	$(10+3k)M+3S$ $+2M_k$	$(7+3k)M+7S$ $+2M_k+2S_k$

Table 4.5: Comparison of the enhancements for the Tate pairing computation

No.	TADD	%	TDBL	%
1	1428M	0	2116M	0
2	728M	51.0	1426M	67.4
3	721M	50.4	1409M	66.6
4	1427M	99.9	2114M	99.9
5	769M	53.8	1457M	68.9

Table 4.6: Comparison of the enhancements for the Tate pairing computation for $k = 7$

We assume that squarings and multiplications use the same amount of time, so we set $M = S$.
For multiplications in $\mathbb{F}_{p^k}$ we have k^2M for the polynomial multiplication with the schoolbook method and $(k-1)k^2M$ for the reduction, so we have $M_k = k^2M + (k-1)k^2M = k^3M$ all together. We also set $S_k = M_k$. Then we set $k = 7$ for the ordinary curve and $k = 2$ for the supersingular curve and give explicit computation times in table 4.6 for the ordinary curve and in 4.7 for the supersingular curve. The "%" column contains the percentage of computation time we the given enhancement uses in comparison to the first, non-optimized version of the Tate pairing.

4.2.1 Timing Results for Ordinary Curves

We use the parameters from (4.0.1) from the beginning of this chapter and give the timing results for our Tate pairing implementation in the table 4.8. All results are computed with simplified Chudnovsky-Jacobian coordinates.

No.	TADD	%	TDBL	%
1	58M	0	76M	0
2	43M	74.1	61M	80.3
3	36M	62.1	54M	71.1
4	57M	98.3	74M	97.4
5	35M	60.3	52M	68.4

Table 4.7: Comparison of the enhancements for the Tate pairing computation for $k = 2$

No.	Method	Average time per computation
4	No enhancement	1129 ms
2+4	$S \in E(\mathbb{F}_p)$	1016 ms
3+4	$S \in E(\mathbb{F}_p)$ and eliminating factors	1003 ms

Table 4.8: Timing results for the Tate pairing with different enhancements and a ordinary curve, $|p^k| = 939$ bits, $|l| = 68$ bits

We want to stress again that the used parameters in this example are not secure enough to use in an actual cryptosystem. The bit length of the given l is only 68 bits, which makes it very easy to solve the EC-DLP on the curve. But the short l has another impact: It reduces the numbers of additions in the main loop of Miller's algorithm. With our l, we only have 31 additions. With cryptographic secure l of at least 160 bits, we would expect, if randomly chosen, about 80 additions and as a result an even longer running time for the Tate pairing.

The main key to further speed enhancements of the algorithm are certainly optimized parameters. The use of a special prime number like a Solinas prime reduces the number of additions and therefore the computation time vitally.
Another way to save computation time are different parameters, so we can use Optimal Extension Fields (OEFs) for arithmetics on the extension field, instead of the slow and straight-forward implementation we have at the moment. Especially the multiplications and inversions will benefit from the use of OEFs.

4.2.2 Timing Results for Supersingular Curves

Now we show timing results for a supersingular curve. This way, we can compare our results to the results in [BKLS02]. Galbraith et al did not

No.	Method	Average time per computation
4	No enhancement	782 ms
2+4	$S \in E(\mathbb{F}_p)$	736 ms
3+4	$S \in E(\mathbb{F}_p)$ and eliminating factors	624 ms

Table 4.9: Timing results for the Tate pairing with different enhancements and a supersingular curve, $|p^k| = 1028$ bits, $|l| = 160$ bits

publish timing results for curves over a prime field with $p > 3$, so we cannot compare our results with the ones given in [GHS02].

We use the same parameters as in (4.1.1) and give the timing results of the Tate pairing in table 4.9, where we use the simplified Chudnovsky-Jacobian coordinates in all methods.

There is a huge difference in the timings between our implementation and the implementation of Barreto et al. For the case of a prime field $p > 3$, Barreto et al only needed 20 ms for the computation of the Tate pairing.
One explanation is the use of optimized parameters like a Solinas prime (i.e. a prime of the form $q = 2^\alpha \pm 2^\beta \pm 1$). With such a prime, only three additions take place. Compared to our prime l, where we have 83 additions, this saves a lot of time. An even greater impact has the fact that Barreto et al used C/C++ for their implementation. As already said at the beginning of this chapter, Javas main goal was to be platform independent. This goal was achieved at the expense of speed. For example, garbage collection takes place when Java wants, not when the programmer tells the programming language to do it.
According to the timing results of the respective operations, the timings of the Tate pairing should be a lot better. But here, the way Java handles repetitive code with its "adaptive compiling" technique makes it hard to predict which part of the code is being optimized and which part is not.
Of course, the advantage of Java and the reason why we chose it, is the platform independence. The give implementation is a good deal slower than implementations in C/C++, but we do not need to compile, and probably alter the code, again if we want to use it on other operating systems or processor architectures.

Chapter 5

Identity Based Encryption

Identity Based Encryption (IBE) is one example for the use of Tate pairings in cryptography.
Shamir first proposed in [Sha84] a public key encryption scheme called IBE. There are four algorithms in such a scheme:

a) *setup* generates global parameters and a *master key*.

b) *extract* uses the master key to generate a private key for a specific ID$\in \{0,1\}^*$.

c) *encrypt* encrypts a given message with a public ID key corresponding to the private key in algorithm 5.0.b.

d) *decrypt* finally decrypts a ciphertext with the appropriate private key.

As one can see, the fundamental idea behind IBE is, that there is no need for a certified public key for sending an encrypted message. If Alice wants to send Bob a message, she simply chooses an arbitrary string, for example Bobs email address, as the public key and encrypts the message. Bob then can obtain a private key for the used ID and decrypt the ciphertext.

In [BF01], Boneh and Franklin introduced an IBE implementation with bilinear maps which we will describe in this chapter.
The first section contains the description of the four algorithms with the use of Tate pairings, in the second section we briefly discuss the security considerations of the given IBE method and the third section gives details on the Java implementation.

5.1 IBE scheme with Tate Pairings

We describe the IBE scheme "BasicIdent" with Tate pairings from [BF01]. There is also an IBE scheme called "FullIdent", which is provably secure against chosen ciphertext attacks. More security considerations can be found in the next section.

5.1.1 Setup

During the *setup*, we choose the necessary parameters for the Tate pairing, i.e. a prime p, the elliptic curve $E(\mathbb{F}_p)$ with order $n = \#E(\mathbb{F}_p)$ and the curve $E(\mathbb{F}_{p^k})$, the MOV degree k and a prime $l|n$.
Next we choose a random *master key* $0 < x < l$, a point $P \in E(\mathbb{F}_{p^k})$ and compute $Q = [x]P$.
With the exception of the master key x, all parameters are public.

5.1.2 Extract

Here, we *extract* the private key $S_{ID} \in E(\mathbb{F}_p)$ for a given ID$\in \{0,1\}^*$.
We use a hash function $H_1 : \{0,1\}^* \to \mathbb{F}_p$ to map the ID to an element on $x \in \mathbb{F}_p$. Then we use a slightly modified version of the algorithm for finding l-torsion points as given in section 3.3.4, to find a point for this x. This algorithm is given in 13. As a output it gives a point $Q_{ID} \in E(\mathbb{F}_p)[l]$ acting as the users public key.
We get the private key by multiplying the master key to the users public key:

$$S_{ID} = [x]Q_{ID} \,.$$

Algorithm 12 Find a point to a given x

Input: An integer x and a curve $E(\mathbb{F}_p)$
Output: A point $P \in E(\mathbb{F}_p)$

```
Set y := −1
while y = −1 do
  y2 := x^3 + ax + b mod p
  if y2 is a square in F_p then
    Set y := y2
  else
    Set x := x + 1 mod p
  end if
end while
return the point P = (x, y, 1)
```

Algorithm 13 Map an arbitrary string to a l-torsion point on $E(\mathbb{F}_p)$

Input: A string $s \in \{0,1\}^*$, a curve $E(\mathbb{F}_p)$ and the group order $\#E(\mathbb{F}_p) = n = kl$

Output: A point $Q_s \in E(\mathbb{F}_p)[l]$ corresponding to the string s or the message "Wrong group order, no l-torsion point found"

Use a hash function $H_1 : \{0,1\}^* \to \mathbb{F}_p$ to get an integer $\tilde{x}_s \in \mathbb{F}_p$ corresponding to the string s
Set $k := n/l \bmod p$
Set $x := \tilde{x}_s$
Let P be the result from algorithm 12 with arguments x and $E(\mathbb{F}_p)$.
Set $G := [k]P$
while $G = \mathcal{O}$ **do**
  Set x to the x coordinate of P
  $x := x + 1 \bmod p$
  Let P be the result from algorithm 12 with arguments x and $E(\mathbb{F}_p)$.
  $G := [k]P$
end while
if $[l]G \neq \mathcal{O}$ **then**
  return "Wrong group order, no l-torsion point found"
else
  return G
end if

5.1.3 Encrypt

To *encrypt* a message $m \in \mathbb{F}_{p^k}$, we choose a random integer $0 < r < l$ and compute

$$C_1 = [r]P \text{ and } C_2 = m \cdot t(Q_{ID}, Q)^r$$

where $t(P, Q)$ is the Tate pairing. The ciphertext is $< C_1, C_2 >$.

Now we need a way to map a an arbitrary string $s \in \{0,1\}^*$ to the polynomial representation of an element in $\mathbb{F}_{p^k}$.
We show how this can be done for a bit string of length $m = \lfloor \log_2 (p^k - 1) \rfloor$. Then we can build a block cipher with block length m for longer strings.
To map an arbitrary bit string $s = s_m s_{m-1} \cdots s_1 s_0$ to an element of $\mathbb{F}_{p^k}$, we first interpret this bit string as an integer value $\tilde{s} = \sum_{i=0}^{m} s_i 2^i$.
Note that we can represent elements of the extension field by an integer value $\{0, 1, \cdots, p^k - 1\}$. For an $A(x) \in \mathbb{F}_{p^k}$, $A(x) = \sum_{i=0}^{k-1} a_i x^i$, the corresponding integer would be $A = A(p) = a_{k-1}p^{k-1} + \cdots + a_1 p + a_0$, i.e. we have a p-adic representation.
Since we need the polynomial representation for our arithmetics, we can now simply convert the integer $\tilde{s}$ to the p-adic representation and so obtain the polynomial.

5.1.4 Decrypt

To *decrypt* a given ciphertext $< C_1, C_2 >$, we compute

$$\tilde{m} = C_2 \cdot t(S_{ID}, C_1)^{-1}.$$

We now show that $\tilde{m} = m$ the plaintext:

$$\begin{aligned}
\tilde{m} &= C_2 \cdot t(S_{ID}, C_1)^{-1} \\
&= m \cdot t(Q_{ID}, Q)^r \cdot t(S_{ID}, C_1)^{-1} \\
&= m \cdot t(Q_{ID}, [x]P)^r \cdot t([x]Q_{ID}, [r]P)^{-1} \\
&= m \cdot t(Q_{ID}, P)^{rx} \cdot t(Q_{ID}, P)^{-rx} \\
&= m.
\end{aligned}$$

5.2 Security of the IBE Implementation

As we have seen in section 2.2, we have to make sure that the discrete logarithm problem is hard on both $E(\mathbb{F}_p)$ and on $\mathbb{F}_{p^k}$, otherwise an attacker could reduce the discrete logarithm problem on $\mathbb{F}_{p^k}$ to the discrete logarithm problem on $E(\mathbb{F}_p)$ and solve it.

Another implication, stated in [BF01], is that the *Decision Diffie-Hellman* problem (DDH) is easy in $E(\mathbb{F}_p)$.

Definition 5.1 *The* Decision Diffie-Hellman problem *in* $E(\mathbb{F}_p)$ *is to distinguish between* $< P, [a]P, [b]P, [c]P >$ *and* $< P, [a]P, [b]P, [ab]P >$ *with* a, b, $c \in \mathbb{F}_p$ *and* $P \in E(\mathbb{F}_p)$ *all random elements.*

To see that the DDH is easy, let P, $[a]P$, $[b]P$, $[c]P \in E(\mathbb{F}_p)$ be given. Then we have the following relation

$$c = ab \bmod q \;\Leftrightarrow\; t(P, [c]P) = t([a]P, [b]P)$$

Therefore we cannot use the DDH for building a cryptosystem in the group $E(\mathbb{F}_p)$. Instead, we use the *Bilinear Diffie-Hellman (BDH)* problem:

Definition 5.2 *Let* G_1, G_2 *be two groups of prime order* q *and* $\hat{e} : G_1 \times G_1 \to G_2$ *be a non-degenerate, bilinear map and* $P \in G_1$ *a generator.*
The Bilinear Diffie-Hellman (BDH) problem *is, for given* $< P, [a]P, [b]P, [c]P >$, a, b, $c \in \mathbb{F}_p$, *to compute* $W = \hat{e}(P, P)^{abc} \in G_2$.

According to [BF01], the BDH in $< G_1, G_2, \hat{e} >$ is thought to be no harder than the Computational Diffie-Hellman problem. The converse is still an open problem.

What we can say is, that the isomorphism induced by the Tate pairing from $E(\mathbb{F}_p)$ to $\mathbb{F}_{p^k}$ is assumed to be a one-way function:
For a point $P \in E(\mathbb{F}_p)$ we define the isomorphism $h_Q : E(\mathbb{F}_p) \to \mathbb{F}_{p^k}$ by $h_Q(P) = t(P, Q)$. If any of these isomorphisms are invertible, the Bilinear Diffie-Hellman problem is easy in $< E(\mathbb{F}_p), \mathbb{F}_{p^k}, t(P, Q) >$, so no hard problems would be left to build a cryptosystem on.
Luckily, there is no known efficient algorithm for inverting h_Q for fixed Q. If there were one such algorithm, that would imply that the DDH in $\mathbb{F}_{p^k}$ is easy. But since we especially chose $\mathbb{F}_{p^k}$ such that the DDH is hard in $\mathbb{F}_{p^k}$, we can assume that all above mentioned isomorphisms are one-way functions.

At last we want to mention that the above given version of the IBE, called "BasicIdent" by Boneh and Franklin, is only semantically secure against a chosen plaintext attack, not provable secure against a chosen ciphertext attack. But "BasicIdent" can be modified so that it is secure against chosen ciphertext attacks. The resulting version is called "FullIdent".
The security proofs and details how to modify "BasicIdent" to obtain "FullIdent" can be found in [BF01].

5.3 Java Implementation of the IBE Scheme

The main part of "BasicIdent" is implemented in the class `IBE`. It holds the functions for *setup, extract, encrypt* and *decrypt*. The public values $P, Q \in E(\mathbb{F}_{p^m})$ and $Q = [x]P$ are stored in the class `IBEPublicKey`. The class `IBE` has to be initialized with a pair of `TPParams` (see section 3.4) before the first usage.
After that, one can encrypt any `String` with a public ID, for example the email address of the recipient. The result of the encrypt is an array of `IBECipherTexts`, where each `IBECipherText` contains an instance of `GFPmElement` and an instance of `PointGFPm`, according to C_1 and C_2 in section 5.1.3.
Finally, the decrypt function takes an array of `IBECipherTexts` and an `IBEPrivateKey` as input and outputs a `String` containing the decrypted ciphertext. The `IBEPrivateKey` contains the point $S_{ID} = [x]Q_{ID}$, as stated in section 5.1.2.
Below we are giving a code example for the usage of our IBE implementation.

Example 5.3 *We assume that* `TPParams params` *contains valid parameters for the Tate pairing.*
For the encryption, one would do:

```
IBE ibealg = new IBE(params);
ibealg.init();
String s = "This text is about to be encrypted";
IBECipherText[] ciphertext = ibealg.encrypt(s,
                "user@domain.invalid");
```

Now, `IBECipherText[]` *contains the ciphertext. The code to decrypt this ciphertext is the following, where we use the same variables as during the encryption:*

```
IBEPrivateKey privkey = ibealg.extract("user@domain.invalid");
String plaintext = ibealg.decrypt(ciphertext, privkey);
```

After this step, the string `plaintext` *contains the value* `This text is about to be encrypted`*.*

Chapter 6

Future Works

In this thesis we gave a description on how to compute the Tate pairing of two points on elliptic curves, an implementation of the Tate pairing and underlying classed in Java for the FlexiProvider framework and an example implementation of an Identity Based Encryption scheme called "BasicIdent". Contrary to existing implementations like [BKLS02] and [GHS02], we concentrated on an implementation for ordinary elliptic curves. Unfortunately, this topic has not been subject to many research projects, yet, so we are missing good parameters for the implementation.

For more special curves, e.g. supersingular ones, there are parameters that allow the computation of the Tate pairing in approximately 60-20 ms when implemented in C/C++. In our case, the most efficient method takes up to 624 ms to finish.
The main reason for this is, as stated in chapter 4, the speed of Java. So an efficient implementation is basically a decision between computation speed and platform independence. In this thesis, we chose the latter.

Another way to improve the performance of the Java implementation is to find parameters for the use with *optimal extension fields (OEFs)*. OEFs allow a fast and efficient implementation of the field arithmetic, especially the field inversion and multiplication benefits from the use of OEFs. Detailed theory and algorithms for the implementation of OEFs can be found in [BP01].
In the beginning, we planned on using OEFs instead of a standard implementation of extension fields. Unfortunately, there are no known curves with small MOV degree which also fulfill the necessary properties for OEFs at the moment, so we had to discard this approach.
In the following, we give a short description of OEFs and then give a brief recapitulation of the properties parameters need to fulfill for the use with

Tate pairings and OEFs.
For the definition of an OEF, we need a new class of prime numbers:

Definition 6.1 *Let $c \in \mathbb{N}$. We say p is a* pseudo-Mersenne prime *if it is a prime number of the form $p = 2^n \pm c$, $log_2 c \leq \lfloor \frac{1}{2} n \rfloor$.*

Then, we can define OEFs as follows.

Definition 6.2 *An* Optimal Extension Field *is a finite field $\mathbb{F}_{p^k}$ such that p is a pseudo-Mersenne prime and an irreducible binomial $P(x) = x^k - \omega$ exists over $\mathbb{F}_p$.*

To find such an irreducible binomial, the following lemma may be used.

Lemma 6.3 *Let $\omega \in \mathbb{F}_p$ be a primitive element and k a divisor of $(p-1)$. Then $x^k - \omega$ is an irreducible binomial over $\mathbb{F}_p$.*

Collecting the properties of parameters for the Tate pairing and for OEFs, we need a prime $p = 2^n \pm c$, $log_2(c) \leq \lfloor \frac{1}{2} n \rfloor$ to have a pseudo-Mersenne prime. We need a prime $l | \# E(\mathbb{F}_p)$ and an integer k such that $l | (p^k - 1)$ and $l \nmid (p^s - 1)$ for $0 < s < k$. Finally k must be a divisor of $(p-1)$, so that the irreducible (field) polynomial $f(t) = t^k - \omega$ exists.

Additionally, one could try and find a value l with low hamming weight. With such an l, the number of TADD calls during Miller's algorithm can be minimized and we can save further computation steps.

Another way is to find more efficient exponentiation techniques. In our implementation, we are using fast exponentiation, which uses up to 750 ms (or 74%) for ordinary curves and 200 ms (or 32%) for supersingular curves for the final exponentiation in the Tate pairing. For the case of supersingular curves, there is such a technique given in [BKLS02]. With a similar technique for the case of ordinary curves, the Tate pairing could be computed faster.

To sum this chapter up, with more research on the parameters for ordinary curves, one can dramatically decrease the computation time of the Tate pairing for ordinary curves.

Appendix A

Derivation of the Line Equations

We use an elliptic curve in short Weierstrass form

$$\begin{aligned} E &: y^2 = x^3 + ax + b \text{ for affine coordinates or} \\ E &: Y^2 = X^3 + aXZ^4 + bZ^6 \text{ for Jacobian coordinates.} \end{aligned}$$

In this chapter we show how the line equations in TADD (from (2.1.2) and (2.1.3)) and TDBL (from (2.1.4) and (2.1.5)) are derived from the affine line equation

$$l(x, y) = y - \lambda x - v \tag{1.2.6}$$

For the addition $(x_1, y_1) + (x_2, y_2) = (x_3, y_3)$ of points where none equals $\mathcal{O}$, the values λ and v are defined as in section 1.2.3:
For $(x_1, y_1) \neq (x_2, y_2)$ and $(x_1, y_1) \neq (x_2, -y_2)$

$$\lambda = \frac{y_2 - y_1}{x_2 - x_1}, \; v = \frac{y_1 x_2 - y_2 x_1}{x_2 - x_1}$$

and when $(x_2, y_2) = (x_1, y_1)$

$$\lambda = \frac{3x_1^2 + a}{2y_1}, \; v = \frac{-x_1^3 + ax_1 + 2b}{2y_1}.$$

We use a Jacobian representation, i.e. we convert points from the affine to the Jacobian representation with setting $x = X/Z^2$ and $y = Y/Z^3$.

A.1 Lines for Addition

In Miller's algorithm, we need two lines during the addition. l_1^{add} is the line through the points $T = (X_1, Y_1, Z_1)$ and $P = (X_2, Y_2, Z_2)$ and l_2^{add} is the

line through the point $T + P = (X_3, Y_3, Z_3)$ parallel to the y axis.
In our special case, $P = (X_2, Y_2, 1)$, i.e. $Z_2 = 1$.

According to the affine line equation, we have

$$l_1^{add}(x,y) = y - y_2 - \frac{y_2 - y_1}{x_2 - x_1}(x - x_2)$$

where (x_1, y_1) are the affine coordinates for T and (x_2, y_2) are the affine coordinates for P.
Replacing the affine with Jacobian coordinates, we get

$$\begin{aligned} l_1^{add}(x,y) &= y - \frac{Y_2}{Z_2^3} - \frac{\frac{Y_2}{Z_2^3} - \frac{Y_1}{Z_1^3}}{\frac{X_2}{Z_2^2} - \frac{X_1}{Z_1^2}}\left(x - \frac{X_2}{Z_2^2}\right) \\ &= \frac{1}{Z_2^3}\left[Z_2^3 y - Y_2 - \frac{\frac{Y_2}{Z_2^3} - \frac{Y_1}{Z_1^3}}{\frac{X_2}{Z_2^2} - \frac{X_1}{Z_1^2}}(Z_2^3 x - Z_2 X_2)\right] . \end{aligned} \tag{A.1.1}$$

Since

$$\frac{Y_2}{Z_2^3} - \frac{Y_1}{Z_1^3} = \frac{Y_2 Z_1^3 - Y_1 Z_2^3}{Z_1^3 Z_2^3}$$

and

$$\frac{X_2}{Z_2^2} - \frac{X_1}{Z_1^2} = \frac{X_2 Z_1^2 - X_1 Z_2^2}{Z_1^2 Z_2^2} = \frac{H}{Z_1^2 Z_2^2}$$

we can write (A.1.1) as

$$\begin{aligned} l_1^{add}(x,y) &= \frac{1}{Z_2^3}\left[Z_2^3 y - Y_2 - \frac{(Y_2 Z_1^3 - Y_1 Z_2^3) Z_1^2 Z_2^2}{H Z_1^3 Z_2^3}(Z_2^3 x - Z_2 X_2)\right] \\ &= \frac{1}{Z_2^3}\left[Z_2^3 y - Y_2 - \frac{Y_2 Z_1^3 - Y_1 Z_2^3}{H Z_1 Z_2}(Z_2^3 x - Z_2 X_2)\right] \\ &= \frac{1}{Z_2^3 Z_1}\left[Z_1 Z_2^3 y - Z_1 Y_2 - \frac{Y_2 Z_1^3 - Y_1 Z_2^3}{H}(Z_2^2 x - X_2)\right] \\ &= \frac{1}{Z_2^3 Z_1 H}\left[H Z_1 Z_2^3 y - H Z_1 Y_2 - (Y_2 Z_1^3 - Y_1 Z_2^3)(Z_2^2 x - X_2)\right] . \end{aligned}$$

As we know from the addition formula for points on elliptic curves (equation 1.2.8), $Z_3 = Z_1 Z_2 H$. In this case $Z_2 = 1$, and so we obtain the final formula for the line l_1^{add}.

$$l_1^{add}(x,y) = \frac{1}{Z_1 H}\left[Z_3(y - Y_2) - (Y_2 Z_1^3 - Y_1)(x - X_2)\right] .$$

At last, we note that according to section 2.3.2, we can eliminate all factors from $\mathbb{F}_p$ from our equation if we use the line equation in Miller's algorithm

with the final exponentiation. That means we can eliminate the fraction $\frac{1}{Z_1 H}$ and obtain the line equation $l_1^{add}(x,y) = Z_3(y - Y_2) - (Y_2 Z_1^3 - Y_1)(x - X_2)$ as used in section 2.1.1.

Next we give the line equation $l_2^{add}(x,y)$ for the line through the point $T + P = (X_3, Y_3, Z_3) \neq \mathcal{O}$ parallel to the y axis.
The affine equation is

$$l_2^{add}(x,y) = x - x_3$$

Replacing the affine with Jacobian coordinates leads to the equation

$$\begin{aligned} l_2^{add}(x,y) &= x - \frac{X_3}{Z_3^2} \\ &= \frac{1}{Z_3^2}\left(Z_3^2 x - X_3\right) \end{aligned}$$

Again, we can eliminate the fraction $\frac{1}{Z_3^2}$ when using the line equation in Miller's algorithm with the final exponentiation, so the line equation for Miller's algorithm is $l_2^{add}(x,y) = Z_3^2 x - X_3$.

In the case that $T + P = \mathcal{O}$, we simply have $l_2^{add}(x,y) = 1$.

A.2 Lines for Doubling

We also need two lines for doubling. The first line $l_1^{dbl}(x,y)$ is the tangent at the point $T = (X_1, Y_1, Z_1)$, the second line $l_2^{dbl}(x,y)$ is the line through the point $[2]T = (X_4, Y_4, Z_4)$ parallel to the y axis.
Same as above, we begin with the affine equation for this case and use (x_1, y_1) and (x_4, y_4) as affine coordinates which we replace against Jacobian coordinates by setting $x = X/Z^2$ and $y = Y/Z^3$. a and b are the coefficients

of the elliptic curve.

$$\begin{aligned}
l_1^{dbl}(x,y) &= y - \frac{3x_1^2+a}{2y_1}x - \frac{2b - x_1^3 + ax_1}{2y_1}x \\
&= y - \frac{3\frac{X_1^2}{Z_1^4}+a}{2\frac{Y_1}{Z_1^3}}x - \frac{2b - \frac{X_1^3}{Z_1^6} + a\frac{X_1}{Z_1^2}}{2\frac{Y_1}{Z_1^3}} \\
&= \frac{1}{2\frac{Y_1}{Z_1^3}}\left[2\frac{Y_1}{Z_1^3}y - \left(3\frac{X_1^2}{Z_1^4}+a\right)x - \left(2b - \frac{X_1^3}{Z_1^6} + a\frac{X_1}{Z_1^2}\right)\right] \\
&= \frac{1}{2\frac{Y_1}{Z_1^3}}\left[2\frac{Y_1}{Z_1^3}y - \frac{1}{Z_1^4}\left(3X_1^2 + Z_1^4 a\right)x - \frac{1}{Z_1^6}\left(2Z_1^6 b - X_1^3 + aZ_1^4 X_1\right)\right] \\
&= \frac{1}{2\frac{Y_1}{Z_1^3}Z_1^6}\left[2Z_1^3 Y_1 y - Z_1^2\left(3X_1^2 + Z_1^4 a\right)x\right. \\
&\quad \left. - \left(2Z_1^6 b - X_1^3 + aZ_1^4 X_1\right)\right]. \qquad \text{(A.2.1)}
\end{aligned}$$

Instead of computing the term $2Z_1^6 b$, we make use of the curve equation and rewrite it as

$$\begin{aligned}
& Y_1^2 = X_1^3 + aX_1Z_1^4 + bZ_1^6 \\
\Leftrightarrow\quad & bZ_1^6 = Y_1^2 - X_1^3 - aX_1Z_1^4\,.
\end{aligned}$$

Then we use this bZ_1^6 in the last parenthesis in equation (A.2.1):

$$\begin{aligned}
& 2Z_1^6 b - X_1^3 + aZ_1^4X_1 \\
=\ & 2Y_1^2 - 2X_1^3 - 2aX_1Z_1^4 - X_1^3 + aZ_1^4X_1 \\
=\ & 2Y_1^2 - 3X_1^3 - aX_1Z_1^4\,.
\end{aligned}$$

With this result, we go back to equation (A.2.1) and remember from equation (1.2.9) that $Z_4 = 2Y_1Z_1$, and so

$$\begin{aligned}
& \frac{1}{2\frac{Y_1}{Z_1^3}Z_1^6}\left[2Z_1^3Y_1y - Z_1^2\left(3X_1^2 + Z_1^4a\right)x - 2Y_1^2 + 3X_1^3 + aX_1Z_1^4\right] \\
=\ & \frac{1}{2\frac{Y_1}{Z_1^3}Z_1^6}\left[Z_4Z_1^2y - 2Y_1^2 - (Z_1^2x - X_1)(3X_1^2 - aX_1Z_1^4)\right]\,.
\end{aligned}$$

When eliminating the fraction $\frac{1}{2\frac{Y_1}{Z_1^3}Z_1^6}$, we obtain the equation $l_1^{dbl}(x,y) = (Z_4Z_1^2y - 2Y_1^2) - (Z_1^2x - X_1)(3X_1^2 - aX_1Z_1^4)$ which we use during the doubling in Miller's algorithm.

The equation for $l_2^{dbl}(x,y)$ can be obtained just like the line for addition, this time through the point $[2]T = (X_4, Y_4, Z_4)$, so we only give the result

$$l_2^{dbl}(x,y) = \frac{1}{Z_4^2}\left(Z_4^2x - X_4\right)$$

and note that the equation we use in Miller's algorithm is consequently $l_2^{dbl}(x,y) = Z_4^2 x - X_4$.

Appendix B

Example Computation of the Tate Pairing

In this chapter, we give an example computation of the Tate pairing with all intermediate results.

We use affine coordinates and affine line equations during Miller's algorithm. $p = 43$, so the subfield is $\mathbb{F}_{43}$, we use the supersingular elliptic curve $E : y^2 = x^3 + x$ which has order $\#E(\mathbb{F}_p) = p + 1 = 44$ as stated in [Men93], $k = 2$ and $l = 11$.
As noted in [Jou02] and [BKLS02], for this special kind of curve we can use the distortion map $\phi(x, y) = (-x, iy)$ with $i \in \mathbb{F}_{p^2}$, $i^2 = -1$ to send l-torsion points on $E(\mathbb{F}_p)$ to l-torsion points on $E(\mathbb{F}_{p^2})$. The irreducible field polynomial is $f(t) = t^2 + 1$.

We use the 11-torsion points $P = (x_2, y_2) = (23, 8)$ and $Q = (x_Q, y_Q) = (20, 8t)$, compute two Tate pairings $t([2]P, Q)$ and $t(P, Q)$ and then decide whether $t([2]P, Q) = t(P, Q)^2$.

Before we give the computations itself, we note that $l = 11$ is 1011 in binary representation, so during Miller's algorithm, we execute the steps TDBL, TDBL, TADD, TDBL, TADD.
We also point out that $l_2^{add}(x_Q, y_Q) = x_Q - x_3 \in \mathbb{F}_p$ and $l_2^{dbl}(x_Q, y_q) = x_Q - x_4 \in \mathbb{F}_p$. Since the final exponentiation with $(p^2 - 1)/l$ eliminates factors in $\mathbb{F}_p$ (see section 2.3.2), we do not need to compute l_2^{add} and l_2^{dbl} in this example.

Step	Op.	T	params	$l_1(x_Q, y_Q)$	f	f^2
init	-	$R = (14, 36)$	-	-	1	1
2	TDBL	$[2]R = (31, 25)$	$(14, 36)$	$17t + 23$	$17t + 23$	$8t + 25$
1	TDBL	$[4]R = (4, 5)$	$(31, 25)$	$13t + 30$	$6t + 1$	-
1	TADD	$[5]R = (23, 35)$	$(4, 5)$ $(14, 36)$	$37t + 13$	$29t + 6$	$4t + 12$
0	TDBL	$[10]R = (14, 7)$	$(23, 35)$	$t + 35$	$23t + 29$	-
0	TADD	$[11]R = (0, 0)$	$(14, 7)$ $(14, 36)$	$0t + 41$	$40t + 28$	-

Table B.1: Computation of $t([2]P, Q)$

B.1 Computation of $t([2]P, Q)$

For convenience, we computed the points occurring during Miller's algorithm in advance and give them here. We set $R = [2]P = (14, 36)$ and obtain the following points:
$[2]R = (31, 25)$, $[4]R = (4, 5)$, $[5]R = (23, 35)$, $[10]R = (14, 7)$, $[11]R = (0, 0)$.

We use the points $T := R = (14, 36)$ and $Q := (20, 8t)$ and summarize the computation in table B.1. The params column contains the points used for the line equation. For TDBL, this is (x_1, y_1), for TADD it is (x_1, y_1) and (x_2, y_2).

As a result of the final exponentiation, we get

$$t([2]P, Q)^{(p^2+1)/l} = (40t + 28)^{168} = 23t + 26\,. \tag{B.1.1}$$

Now we compute $t(P, Q)^2$ and compare the result with the one above.

B.2 Computation of $t(P, Q)$

Again, we computed the points occurring during Miller's algorithm in advance and give them here. We use $P = (23, 8)$ and obtain the following points:
$[2]P = (14, 36)$, $[4]P = (31, 25)$, $[5]P = (13, 24)$, $[10]P = (23, 35)$, $[11]P = (0, 0)$.

We use the points $T = P = (23, 8)$ and $Q = (20, 8t)$ and summarize the computation in table B.2. The params column contains the points used for the line equation. For TDBL, this is (x_1, y_1), for TADD it is (x_1, y_1) and (x_2, y_2).

Step	Op.	T	params	$l_1(x_Q, y_Q)$	f	f^2
init	-	$P = (23, 8)$	-	-	1	1
2	TDBL	$[2]P = (14, 36)$	$(23, 8)$	$42t + 35$	$42t + 35$	$16t + 20$
1	TDBL	$[4]P = (31, 25)$	$(14, 36)$	$17t + 23$	$20t + 16$	-
1	TADD	$[5]P = (13, 24)$	$(31, 25)$ $(23, 8)$	$22t + 13$	$10t + 26$	$4t + 17$
0	TDBL	$[10]P = (23, 35)$	$(13, 24)$	$40t + 22$	$37t + 22$	-
0	TDBL	$[11]P = (0, 0)$	$(23, 35)$ $(23, 8)$	$0t + 5$	$13t + 38$	-

Table B.2: Computation of $t(P,Q)$

The result of the final exponentiation is

$$t(P,Q)^{(p^2+1)/l} = (13t + 38)^{168} = 3t + 11\,.$$

Because of the bilinearity of the Tate pairing, we should have $t(P,Q)^2 = t([2]P,Q)$. This is true, since

$$t(P,Q)^2 = (3t + 11)^2 = 23t + 26\,,$$

the same value as in B.1.1.

Bibliography

[BF01] D. Boneh and M. Franklin. Identity-Based Encryption from the Weil Pairing. In *CRYPTO 2001*, volume 2139 of *Lecture Notes in Computer Science*, pages 213–229. Springer Verlag, 2001.

[BKLS02] P. Barreto, H. Kim, B. Lynn, and M. Scott. Efficient Algorithms for Pairing-Based Cryptosystems. In *CRYPTO 2002*, volume 2442 of *Lecture Notes in Computer Science*, pages 47–53. Springer Verlag, 2002.

[BP01] Daniel V. Bailey and Christof Paar. Efficient Arithmetic in Finite Field Extensions with Application in Elliptic Curve Cryptography. volume 14 of *Journal of Cryptology*, pages 153–176. Springer Verlag, 2001.

[BSS99] Ian F. Blake, Gadiel Seroussi, and Nigel P. Smart. *Elliptic Curves in Cryptography*, volume 265 of *LMS Lecture Notes Series*. Cambridge University Press, 1999.

[Buc] Johannes Buchmann. Algorithms for Binary Quadratic Forms. yet unpublished book.

[Buc99] Johannes Buchmann. *Einführung in die Kryptographie*. Springer Verlag, 1999.

[CMO98] H. Cohen, A. Miyaji, and T. Ono. Efficient Elliptic Curve Exponentiation Using Mixed Coordinates. In *ASIACRYPT '98*, volume 1514 of *Lecture Notes in Computer Science*, pages 51–65. Springer Verlag, 1998.

[DEM02] R. Dupont, A. Enge, and F. Morain. Building curves with arbitrary small MOV degree over finite prime fields. *Cryptology ePrint Archive, Report 2002/094*, 2002.

[Fle] Flexiprovider toolkit for JCA/JCE. http://www.fleixprovider.de.

[FMR99] G. Frey, M. Müller, and H.G. Rück. The Tate Pairing and the Discrete Logarithm Applied to Elliptic Curve Cryptography. *IEEE Trans. on Information Theory, volume 45*, pages 1717–1718, 1999.

[GHS02] S.D. Galbraith, K. Harrison, and D. Soldera. Implementing the Tate Pairing. In *ANTS-V*, volume 2369 of *Lecture Notes in Computer Science*, pages 324–337. Springer Verlag, 2002.

[IEE00] P1363 Working Group IEEE. IEEE P1363, standard specifications for public key cryptography, 2000.

[IEE02] P1363 Working Group IEEE. IEEE P1363a, standard specifications for public key cryptography, amendment 1, 2002.

[IT03] Tetsuya Izu and Tsuyoshi Takagi. Efficient Computations of the Tate Pairing for the Large MOV Degrees. In *ICISC 2002*, volume 2587 of *Lecture Notes in Computer Science*, pages 283–297. Springer Verlag, 2003.

[Jou02] Antoine Joux. The Weil and Tate Pairings as Building Blocks for Public Key Cryptosystems. In *ANTS-V*, volume 2369 of *Lecture Notes in Computer Science*, pages 20–32. Springer Verlag, 2002.

[LN86] Rudolf Lidl and Harald Niederreiter. *Introduction to Finite Fields and their Applications.* Press Syndicate of the University of Cambridge, 1986.

[LV01] Arjen K. Lenstra and Eric R. Verheul. Selecting Cryptographic Key Sizes. volume 14 of *Journal of Cryptology*, pages 255–293. Springer Verlag, 2001.

[Men93] Alfred Menezes. *Elliptic Curve Public Key Cryptosystems.* SECS 234. Kluwer Academic Publishers, 1993.

[Mil86] Victor S. Miller. Short Programs for Functions on Curves. Unpublished manuscript, 1986.

[MNT01] A. Miyaji, M. Nakabayashi, and S. Takano. New Explicit Conditions of Elliptic Curve Traces for FR-reduction. In *IEICE Trans. Fundamentals*, volume E84 A. Oxford University Press, May 2001.

[MOV93] A. Menezes, T. Okamoto, and S. Vanstone. Reducing Elliptic Curve Logarithms to Logarithms in a Finite Field. *IEEE Transaction on Information Theory volume 39, pages 1639-1646*, 1993.

[OP01] T. Okamoto and D. Pointcheval. The Gap-Problems: a New Class of Problems for the Security of Cryptographic Schemes. In *PKC 2001*, volume 1992 of *Lecture Notes in Computer Science*, pages 104–118. Springer Verlag, 2001.

[Sha84] A. Shamir. Identity-based cryptosystems and signature schemes. In *CRYPTO '84*, volume 196 of *Lecture Notes in Computer Science*, pages 47–53. Springer Verlag, 1984.

[Sil86] Joseph H. Silverman. *The Arithmetic of Elliptic Curves*. Graduate Texts in Mathematics. Springer Verlag, 1986.

[ST92] Joseph H. Silverman and John Tate. *Rational Points on Elliptic Curves*. Undergraduate Texts in Mathematics. Springer Verlag, 1992.