

Diplomarbeit  
am Fachbereich Mathematik  
der Technischen Universität Darmstadt



## Counting Points on Elliptic Curves

Angefertigt von  
Lea Poeplau

unter Betreuung von  
Prof. Dr. Johannes Buchmann

20. Dezember 2005



## Acknowledgements

First of all, I would like to thank Prof. Dr. Johannes Buchmann and Ulrich Vollmer, who made this thesis possible.

My special thanks go to my family who supported me throughout my studies in every possible way. I also want to thank Max, Lars, Karsten, Andi, Rafael, W. Nickel, Frauke, Nicole and many others for their help in proofreading, technical support, always being receptive to my thoughts and problems and cheering me up. And, last but not least, thanks, Robert, for all the fish!

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Mathematical Background</b>	<b>3</b>
2.1	$p$ -Adic Rings and Fields . . . . .	3
2.1.1	Extensions of $\mathbb{Q}_p$ . . . . .	5
2.1.2	Representation of $\mathbb{Q}_q$ . . . . .	5
2.1.3	The Little Frobenius of $\mathbb{Z}_q$ . . . . .	6
2.2	Elliptic Curves . . . . .	7
2.2.1	The Group Law . . . . .	10
2.2.2	Isogenies . . . . .	11
2.2.3	Reduction . . . . .	15
2.2.4	The Canonical Lift . . . . .	15
2.2.5	Division Polynomials . . . . .	16
2.3	Modular Polynomials . . . . .	18
2.4	Newton Iteration . . . . .	19
2.4.1	The General Case . . . . .	19
2.4.2	The $p$ -Adic Case . . . . .	20
2.4.3	Hensel's Lemma . . . . .	21
<b>3</b>	<b>Satoh's Algorithm</b>	<b>23</b>
3.1	The Cycle of Curves and their $j$ -Invariants . . . . .	24
3.2	Lifting the $j$ -Invariants . . . . .	25
3.3	Characteristic $p \geq 5$ . . . . .	30
3.3.1	Lifting each Curve . . . . .	30
3.3.2	Lifting each Curve's Torsion Subgroup . . . . .	30
3.3.3	The Formal Group Approach . . . . .	38
3.3.4	Computing the Trace from the Lifted Data . . . . .	39
3.4	Characteristic $p = 2$ . . . . .	42
3.4.1	Lifting each Curve . . . . .	42
3.4.2	Lifting the 2-Torsion . . . . .	43
3.4.3	Computing the Trace from the Lifted Data . . . . .	45
3.5	An Example . . . . .	50
3.6	Extensions of Satoh's Algorithm . . . . .	51
<b>4</b>	<b>Implementation</b>	<b>54</b>
4.1	Representation of $\mathbb{Z}_2$ and $\mathbb{Z}_{2^n}$ . . . . .	54
4.2	The Subsidiary Classes . . . . .	55
4.3	The Main Classes . . . . .	56
4.4	Encountered Difficulties . . . . .	56
4.5	Satoh-FGH vs Skjernaa-VPV . . . . .	59
4.6	Integration in the FlexiECPProvider . . . . .	60

4.7	Numerical Results . . . . .	61
<b>5</b>	<b>Earlier Point Counting Algorithms</b>	<b>63</b>
5.1	Schoof . . . . .	63
5.2	Schoof-Elkies-Atkin . . . . .	64
5.3	Comparison . . . . .	64
5.4	Conclusion . . . . .	65
<b>A</b>	<b>On Time Complexity</b>	<b>66</b>
A.1	Aspects of Complexity Theory . . . . .	66
A.2	Big $\mathcal{O}$ Notation . . . . .	67
<b>B</b>	<b>The JAVA Code</b>	<b>69</b>
B.1	The class <code>ECGF2nParameterGenerator</code> . . . . .	69
B.2	The class <code>SatohFGH</code> . . . . .	74
B.3	The class <code>SkjernaVPV</code> . . . . .	80
B.4	The class <code>Z2nRing</code> . . . . .	85
B.5	The class <code>Z2nRingElement</code> . . . . .	89



# 1 Introduction

The subject of the thesis at hand is the description of the Satoh-FGH algorithm for counting points on elliptic curves over finite fields. Moreover, we provide an implementation of the algorithm for curves over finite fields of characteristic two.

The group of rational points of an elliptic curve over a field  $F$  of characteristic  $p$  is the set of solutions of a so-called *Weierstrass equation*

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

over the field  $F$ , where the  $a_i$  are elements of  $F$ . Formal definitions will be given in Section 2.2.

In recent cryptography, elliptic curves have become a popular and powerful tool. One major application are the several encryption systems based on the *discrete logarithm problem*. The advantage over multiplicative groups lies in the much smaller bitsizes needed for successful encryption.

There are basically two reasons why the number of points on an elliptic curve is of great importance. The first lies in the fact that some cryptographic protocols need the number of points on a curve as an input for their algorithms.

The second reason deals with the usefulness of certain elliptic curves. In fact, not all elliptic curves are useful in cryptosystems. There are certain demands made on the group order of a curve. Therefore, the cardinality of the group of rational points is one of an elliptic curve's most important aspects.

In [Sat00], Takakazu Satoh described a  $p$ -adic algorithm with runtime in  $O(\log^{3+\varepsilon}(p^d))$  for small fixed characteristic  $p \geq 5$  and suggested the extension to characteristics two and three. This suggestion was acted on by Mireille Fouquet, Pierrick Gaudry and Robert Harley in [FGH00] and by Berit Skjernaas in [Skj03]. Considering the strong dependency of Satoh's algorithm on the characteristic, the case of characteristic two is the most important one for the algorithm's use in cryptography.

The basic idea of Satoh's  $p$ -adic algorithm is completely different from  $l$ -adic algorithms such as Schoof and SEA. In  $l$ -adic methods, one computes the trace that leads to the number of points modulo several primes  $l$  and uses the Chinese Remainder Theorem to obtain the actual trace. In  $p$ -adic methods however, the trace is computed modulo powers of one prime  $p$ , the characteristic of the ground field. An overview of the most significant earlier point counting algorithms is given in Section 5.

## Structure and content of this thesis

Chapter 2 will give the necessary mathematical background. It covers an introduction to  $p$ -adic rings and fields, elliptic curves and some general theory needed for the algorithm.

In Chapter 3, we describe the Satoh-FGH algorithm for point counting on elliptic curves over fields of odd characteristic and of characteristic two. We left out characteristic three, since it is usually not applied in cryptography. After the description, we present a small

example. At the end of the chapter, we present the most significant extensions of Satoh's algorithm.

Chapter 4 contains the documentation of our implementation of the Satoh-FGH algorithm in characteristic two. We say a few words on its complexity and present some computational results.

Some earlier important point counting algorithms are shortly presented in Chapter 5. They will be compared, mostly with respect to their complexities. In the end, we can draw a conclusion on the usability in cryptography.

## 2 Mathematical Background

In this section, the mathematical background needed for Satoh's algorithm is presented. Some background in general algebra is assumed and can be found for example in [Lan02]. For further information on the arithmetic of elliptic curves we refer to [Sil92] and [Hus87].

### 2.1 $p$ -Adic Rings and Fields

This section provides some useful facts about  $p$ -adic rings and fields. We use the concept of an *ultrametric norm*.

**Definition 2.1.** Let  $K$  be a commutative ring, with zero element  $0$  and unit element  $1$ . Let  $v: K \rightarrow \mathbb{N}$  be a mapping satisfying the following properties (for  $x, y \in K$ ):

$$(v1) \quad v(x) = 0 \text{ if and only if } x = 0,$$

$$(v2) \quad v(1) = v(-1) = 1,$$

$$(v3) \quad v(xy) \leq v(x)v(y),$$

$$(v4) \quad v(x + y) \leq \max\{v(x), v(y)\}.$$

The map  $v$  is called an *ultrametric norm* on  $K$ .

Fix a prime  $p$  and let  $a \neq 0 \in \mathbb{Z}$ . Then we define

$$\text{ord}_p(a) = \max\{i \in \mathbb{N} \mid a \equiv 0 \pmod{p^i}\}.$$

By definition, we set  $\text{ord}_p(0) = \infty$ , so  $\text{ord}_p: \mathbb{Z} \rightarrow \mathbb{N} \cup \{\infty\}$ . The function  $\text{ord}_p$  can be extended to  $\mathbb{Q}$  by defining

$$\text{ord}_p\left(\frac{a}{b}\right) = \text{ord}_p(a) - \text{ord}_p(b) \text{ for } a, b \in \mathbb{Z}, b \neq 0.$$

The function  $\text{ord}_p$  has some useful properties.

**Proposition 2.2.** *The function  $\text{ord}_p$  satisfies the following properties:*

For  $r, s \in \mathbb{Q}$ ,

$$\begin{aligned} \text{ord}_p(rs) &= \text{ord}_p(r) + \text{ord}_p(s) \\ \text{ord}_p(r + s) &\geq \min\{\text{ord}_p(r), \text{ord}_p(s)\} \end{aligned}$$

These properties imply that the function

$$\begin{aligned} |\cdot|_p: \mathbb{Q} &\rightarrow \mathbb{Q} \\ x &\mapsto p^{-\text{ord}_p(x)} \end{aligned}$$

defines an ultrametric norm on  $\mathbb{Q}$ . With respect to this norm, we can build a completion of  $\mathbb{Q}$ .

**Definition 2.3.** A *valuation* on a field  $K$  is a function  $\nu: K \rightarrow \mathbb{R}$  satisfying the following properties for all  $x, y \in K$ :

1.  $\nu(x) \geq 0$ ,  $\nu(x) = 0 \Leftrightarrow x = 0$ ,
2.  $\nu(xy) = \nu(x) + \nu(y)$ ,
3.  $\nu(x + y) \leq \max(\nu(x), \nu(y))$ .

The function  $\nu$  is called a *discrete valuation* if its image is a discrete subset of  $\mathbb{R}$ .

**Definition 2.4.** The field  $\mathbb{Q}_p$  of *p-adic numbers* is the completion of  $\mathbb{Q}$  with respect to the *p-adic norm*  $|\cdot|_p$ . The ring  $\mathbb{Z}_p$  of *p-adic integers* is defined as

$$\mathbb{Z}_p = \{x \in \mathbb{Q}_p : |x|_p \leq 1\} = \{x \in \mathbb{Q}_p : \text{ord}_p(x) \geq 0\}.$$

Then  $\mathbb{Z}_p$  is a local ring with discrete valuation  $\text{ord}_p$  and unique maximal ideal

$$M = \{x \in \mathbb{Z}_p : \text{ord}_p(x) > 0\}.$$

The residue field  $\mathbb{Z}_p/M$  is isomorphic to  $\mathbb{F}_p$ , the finite field with  $p$  elements. By  $\bar{a} \in \mathbb{F}_p$  we denote the reduction of an element  $a \in \mathbb{Z}_p$  modulo  $M$ .

**Proposition 2.5.** *The units in  $\mathbb{Z}_p$  are precisely the elements with valuation zero:*

$$\mathbb{Z}_p^\times = \mathbb{Z}_p \setminus M,$$

and thus  $\bar{u} \neq 0$  for  $u \in \mathbb{Z}_p^\times$ , since  $\text{ord}_p(0) = \infty$ .

With this characterisation of units, we can see that every element  $a \in \mathbb{Z}_p$  can be written as

$$a = p^k \cdot u,$$

where  $k \geq 0$  and  $u$  is a unit in  $\mathbb{Z}_p$ . The integer  $k$  is called the *valuation* of  $a$  and the inverse of  $a$  in  $\mathbb{Q}_p$  is

$$a^{-1} = \underbrace{p^{-k}}_{\notin \mathbb{Z}_p} \cdot \underbrace{u^{-1}}_{\in \mathbb{Z}_p}.$$

Another, sometimes more useful, representation is the following. An element of  $\mathbb{Q}_p$  can be approximated by a truncated Laurent series as stated in the next Theorem.

**Theorem 2.6.** *Every element  $a \in \mathbb{Q}_p$  can be represented uniquely as a Laurent series*

$$a = \sum_{i=m}^{\infty} a_i p^i,$$

with  $a_i \in \mathbb{F}_p$ ,  $a_m \neq 0$  and  $m = \text{ord}_p(a)$ .

### 2.1.1 Extensions of $\mathbb{Q}_p$

Let  $K$  be a finite extension of  $\mathbb{Q}_p$  of degree  $d$ . The norm  $|\cdot|_p$  can be extended in a unique way from  $\mathbb{Q}_p$  to  $K$ . This norm also defines a valuation on  $K$  given by

$$\text{ord}_p(\alpha) = -\log_p(|\alpha|_p)$$

for  $\alpha \in K$ .

The following definition explains the concept of the valuation ring of  $K$ .

**Definition 2.7.** Let  $R$  be a ring and  $S$  an extension of  $R$ . An element  $s \in S$  is called *integral* over  $R$ , if it is a root of a monic polynomial with coefficients in  $R$ .

The ring  $S' \subset S$  of integral elements over  $R$  is called the *integral closure* of  $R$ .

The integral closure of  $\mathbb{Z}_p$  in  $K$  is given by

$$R = \{x \in K : |x|_p \leq 1\}$$

and  $M = \{x \in K : |x|_p < 1\}$  is the unique maximal ideal of  $R$ . The ring  $R$  is called the *valuation ring* of  $K$  and the residue field of  $K$  is defined as  $R/M$ .

**Definition 2.8.** Let  $K$  be a finite extension of  $\mathbb{Q}_p$  of degree  $d$ , then  $K$  is called *unramified* if its residue field  $R/M$  is isomorphic to  $\mathbb{F}_q$ , where  $q = p^d$ .

An unramified extension of  $\mathbb{Q}_p$  of degree  $d$  is unique up to isomorphism and is denoted by  $\mathbb{Q}_q$ , with  $q = p^d$ . The valuation ring of  $\mathbb{Q}_q$  is denoted by  $\mathbb{Z}_q$ .

### 2.1.2 Representation of $\mathbb{Q}_q$

In the same way that  $\mathbb{F}_q$  can be represented as

$$\mathbb{F}_q \cong \mathbb{F}_p[t]/(f(t)),$$

where  $f(t) \in \mathbb{Z}_p[t]$  is an irreducible monic polynomial of degree  $d$ , we can write  $\mathbb{Q}_q$  as

$$\mathbb{Q}_q \cong \mathbb{Q}_p[t]/(\tilde{f}(t)),$$

where  $\tilde{f}(t) \in \mathbb{Z}_p[t]$  is any lift of  $f(t)$ . The valuation ring  $\mathbb{Z}_q$  can be represented as

$$\mathbb{Z}_q \cong \mathbb{Z}_p[t]/(\tilde{f}(t))$$

and its elements  $a$  are written as

$$a = \sum_{i=0}^{d-1} a_i t^i, \quad a_i \in \mathbb{Z}_p.$$

Figure 1 gives an idea of how  $\mathbb{F}_p, \mathbb{F}_q, \mathbb{Z}_p$  and  $\mathbb{Z}_q$  are related.

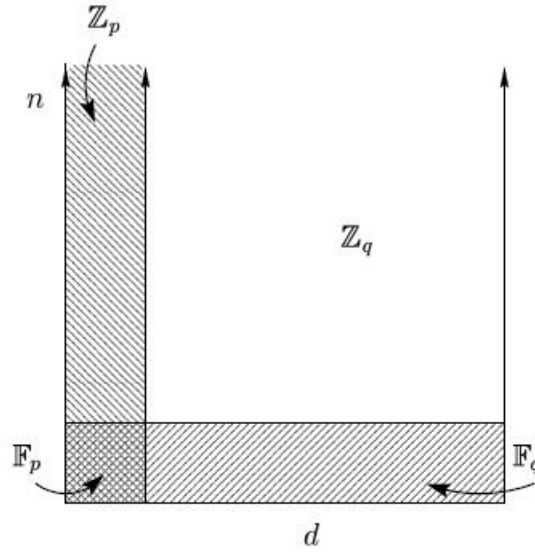


Figure 1: Relation of  $\mathbb{F}_p, \mathbb{F}_q, \mathbb{Z}_p$  and  $\mathbb{Z}_q$  ([FGH00])

### 2.1.3 The Little Frobenius of $\mathbb{Z}_q$

To define the so-called *little Frobenius* of  $\mathbb{Z}_q$ , note that it is *not* a simple  $p$ -powering like the  $p$ -th power Frobenius action of the field  $\mathbb{F}_q$ . Still, it is the lift of the  $p$ -powering map of  $\mathbb{F}_q$ . This leads to the following construction.

**Definition 2.9.** The *little Frobenius* on  $\mathbb{F}_q$  is the field automorphism

$$\sigma: x \mapsto x^p,$$

fixing the prime subfield  $\mathbb{F}_p$ .

The following definitions will lead to the little Frobenius  $\Sigma$  of  $\mathbb{Z}_q$ . They are stated here mainly for completeness, since computing  $\Sigma$  as indicated here is rather slow in practice. The Satoh and FGH algorithms avoid computing  $\Sigma$  explicitly as will be explained in chapter 3.

The first step will be to define a particular lift of scalars from  $\mathbb{F}_q$  to  $\mathbb{Z}_q$  which respects the multiplicative structure.

**Definition 2.10.** The *Teichmüller lift* is the map  $\omega: \mathbb{F}_q \rightarrow \mathbb{Z}_q$  defined by

- $\omega(0) = 0$ ,
- and for  $x$  non-zero,  $\omega(x)$  is the unique  $(q - 1)$ -th root of unity in  $\mathbb{Z}_q$  such that  $\pi(\omega(x)) = x$ , where  $\pi$  denotes the projection to  $\mathbb{F}_p$ .

The following concept is basically a particular decomposition of elements of  $\mathbb{Z}_q$ . It is close to the representation of *Witt vectors* (see Chapter II of [Ser79] for details on Witt vectors).

**Definition 2.11.** The *semi-Witt decomposition* of  $x \in \mathbb{Z}_q$  is a sequence  $(x_i)_{i \geq 0}$  of  $x_i \in \mathbb{F}_q$  such that

$$x = \sum_{i \geq 0} \omega(x_i) p^i.$$

The coordinates  $x_i$  can be computed from  $x$  inductively by

$$\begin{aligned} x_0 &\equiv \pi(x) \\ x_i &\equiv \pi\left(\frac{x - \omega(x_{i-1})}{p}\right), \quad \text{for } i \geq 1. \end{aligned}$$

*Remark 2.12.* In fact, the semi-Witt decomposition defines the unique sequence with the above property.

Now we can define  $\Sigma$ , the little Frobenius lifted from  $\mathbb{F}_q$  to  $\mathbb{Z}_q$ . It satisfies the natural property  $\pi \circ \Sigma = \sigma$  where  $\sigma$  is again the little Frobenius on  $\mathbb{F}_q$ .

**Definition 2.13.** The *little Frobenius*  $\Sigma: \mathbb{Z}_q \rightarrow \mathbb{Z}_q$  is defined as follows.

For any  $x \in \mathbb{Z}_q$ , let  $(x_i)_{i \geq 0}$  be its semi-Witt decomposition. Then  $\Sigma(x)$  is defined to be the element of  $\mathbb{Z}_q$  with decomposition  $(x_i^p)_{i \geq 0}$ . In other words,

$$\Sigma(x) = \sum_{i \geq 0} \omega(x_i^p) p^i$$

which is equivalent to saying

$$\Sigma(x) = \sum_{i \geq 0} \omega(x_i) p^i.$$

## 2.2 Elliptic Curves

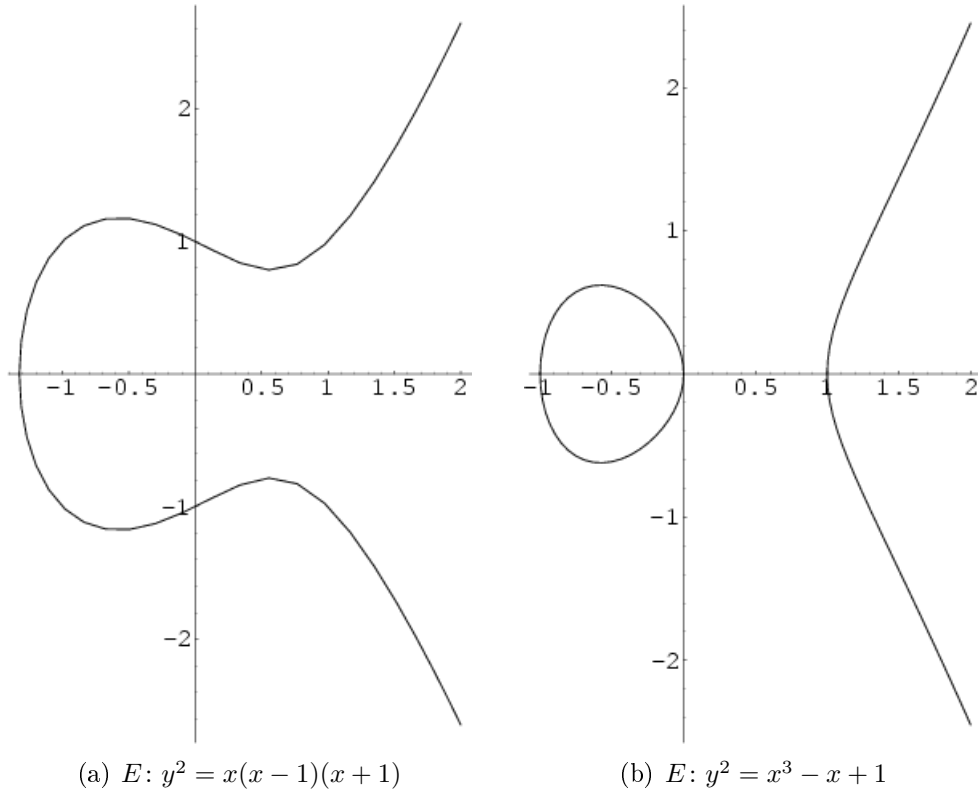
In this section we will give the definition of an elliptic curve over a finite field  $\mathbb{F}_q$ , where  $q = p^d$  for some prime number  $p$ . We will consider the group of rational points and some other important properties of an elliptic curve.

**Definition 2.14.** An *elliptic curve*  $E$  over a field  $K$  is a non-singular (see 2.16) plane curve defined by a *Weierstrass equation* of the form

$$y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6, \quad (2.1)$$

where  $a_1, a_2, a_3, a_4, a_6 \in K$ , together with a distinguished basepoint  $\mathcal{O}$ .

In his algorithm, Satoh uses different quantities of elliptic curves to lift the curves and the Frobenius map to an extension of  $\mathbb{F}_q$ .

Figure 2: Two elliptic curves over  $\mathbb{R}$ 

**Definition 2.15.** Let  $E: y^2 = x^3 + ax + b$  be an elliptic curve defined over a finite field  $\mathbb{F}_q$  of characteristic  $p \geq 5$ . Then the *j-invariant* of  $E$  is

$$j(E) = -\frac{2^{12}3^3a^3}{\Delta} = \frac{2^83^3a^3}{4a^3 + 27b^2},$$

where

$$\Delta = -16(4a^3 + 27b^2).$$

The *invariant differential* of  $E$  is

$$\omega = \frac{dx}{2y} = \frac{dy}{3x^2 + a}.$$

In the case of an elliptic curve  $E: y^2 + xy = x^3 + ax^2 + b$  over a finite field of characteristic two, the *j-invariant* is

$$j(E) = \frac{1}{b},$$

In the case of characteristic two, the criterion for supersingularity reduces to the equivalent condition that  $j(E) = 0$ .

**Definition 2.16.** (a) A curve over a field of characteristic  $p > 0$  is called *non-singular* if

$$\Delta \neq 0.$$

If the field  $K$  is of characteristic 0, a curve is called non-singular if

$$\Delta \neq 0.$$

Geometrically, the non-singularity means the curve has no cusps or self-intersections.

(b) An elliptic curve  $E$  over a field of characteristic  $p > 0$  is said to be *supersingular* if there are no points of order  $p$  on  $E$ , that is

$$\{P \in E : p \cdot P = \mathcal{O}\} = \emptyset.$$

Throughout this thesis, we will assume the elliptic curves to be *ordinary*, that is, non-supersingular.

**Transformation of the curve's equation** An appropriate change of variables can in some cases lead to an easier form of the equation of the curve. Consider, for example, the curve given by the equation

$$E: y^2 = x^3 + ax^2 + bx + c$$

over a field of characteristic different from 2 and 3. If now we set

$$x' = x + \frac{a}{3}$$

the equation becomes

$$y^2 = x'^3 + b'x' + c',$$

where

$$b' = b - \frac{a^2}{3} \quad \text{and} \quad c' = c + \frac{2a^3}{27} - \frac{ab}{3}.$$

The following lemma generalizes how the curve's equation can be simplified. Beforehand, we need the following definition.

**Definition 2.17.** The *trace* of an element  $x \in \mathbb{F}_q$ , where  $q = 2^d$ , is defined by

$$\text{Tr}(x) = \sum_{i=0}^{d-1} x^{2^i}.$$

Further information on the trace can be found in [BSS99].

**Lemma 2.18.** (a) In the case where we consider the curve  $E$  over a finite field  $\mathbb{F}_q$ , where  $q = p^d$  for some prime number  $p \geq 5$ , we can simplify the equation to

$$y^2 = x^3 + ax + b, \tag{2.2}$$

where  $a, b \in \mathbb{F}_q$ .

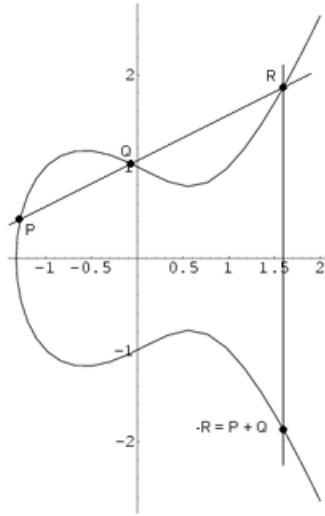


Figure 3: Addition of two points  $P$  and  $Q$  on an elliptic curve

(b) If the characteristic is  $p = 3$ , the curve's equation can be reduced to the form

$$y^2 = x^3 + ax^2 + bx + c,$$

where  $a, b, c \in \mathbb{F}_q$ .

(c) In the case of  $p = 2$ , the curve's equation can be transformed to

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

where  $a, b \in \mathbb{F}_q$ . Note that this equation is only valid for the ordinary case.

In fact,  $b$  is a unit in  $\mathbb{F}_q$  and  $a \in \{0, \gamma\}$ , where  $\gamma$  is a fixed element in  $\mathbb{F}_q$  of trace  $\text{Tr}(\gamma) = 1$ .

In his algorithm, Satoh only considered primes  $p \geq 5$ . Following his work, M. Fouquet, P. Gaudry and R. Harley extended the algorithm to the cases  $p = 2$  and  $p = 3$ . In the following this will be referred to as the Satoh-FGH algorithm.

In this work, we will consider the cases  $p \geq 5$  and  $p = 2$  only, since the case  $p = 3$  is usually not applied in cryptography. Furthermore, the implementation was done only for curves over fields of characteristic two, since this is the most relevant case.

### 2.2.1 The Group Law

Let  $E$  be an elliptic curve given by a Weierstrass equation. Let  $P$  and  $Q$  be two (not necessarily distinct) points on  $E$ . Let  $L$  be the line connecting  $P$  and  $Q$  (the tangent line

to  $E$  if  $P = Q$ ). Then  $L$  intersects  $E$  in a third point  $R$ . Let  $L'$  be the line connecting  $R$  and  $\mathcal{O}$ . Then we define the composition  $P + Q$  to be the intersection point of  $L'$  with  $E$  (compare Figure 3).

**Proposition 2.19.** *An elliptic curve  $E$  defined over a field  $K$  together with the composition  $+$  forms an abelian group with neutral element  $\mathcal{O}$ .*

A proof and the formal properties of the composition  $+$  can be found in [Sil92], III.2.

### 2.2.2 Isogenies

Having studied some important properties of elliptic curves, we now turn to consider the maps between them. Since an elliptic curve has a distinguished zero point  $\mathcal{O}$ , it is natural to focus on maps that respect that property.

**Definition 2.20.** Let  $E_1$  and  $E_2$  be two elliptic curves defined over a field  $K$ . An *isogeny* between  $E_1$  and  $E_2$  is a morphism<sup>1</sup>

$$\sigma: E_1 \rightarrow E_2$$

satisfying  $\sigma(\mathcal{O}) = \mathcal{O}$ . If  $\sigma(E_1) \neq \{\mathcal{O}\}$ , we say that  $E_1$  and  $E_2$  are isogeneous. We let  $[0]: E_1 \rightarrow E_2$  denote the zero isogeny.

A non-trivial isogeny  $\sigma: E_1 \rightarrow E_2$  is actually a surjective group homomorphism and induces an injection of function fields

$$\sigma^*: \bar{K}(E_2) \rightarrow \bar{K}(E_1).$$

The *degree*  $\deg(\sigma)$ , the separable degree  $\deg_s(\sigma)$  and the inseparable degree  $\deg_i(\sigma)$  are defined by the corresponding properties of the finite extension

$$\sigma^*(K(E_2)) \subseteq K(E_1).$$

If  $\deg_s(\sigma) = \deg(\sigma)$ , we say the  $\sigma$  is *separable*. If  $\deg_i(\sigma) = \deg(\sigma)$ , we say the  $\sigma$  is (*purely*) *inseparable*. By convention, we set

$$\deg[0] = 0.$$

Note that an isogeny of degree 1 is an isomorphism.

An isogeny is called *cyclic* if its kernel is a cyclic group.

Actually, the degree of an isogeny plays an important role as can be seen in the following proposition.

**Proposition 2.21.** *If  $\sigma$  is a separable isogeny,*

$$\deg \sigma = \# \ker(\sigma).$$

---

<sup>1</sup>For an exact definition see [Sil92].

The next Theorem shows that being isogenous is in fact a symmetric relation.

**Theorem 2.22.** *Let  $\sigma$  be an isogeny of degree  $d$  between two elliptic curves  $E_1$  and  $E_2$ . Then there exists an isogeny  $\hat{\sigma}: E_2 \rightarrow E_1$  such that*

$$\hat{\sigma} \circ \sigma = [d \deg \sigma],$$

where  $[m]$  denotes the by- $m$ -multiplication on  $E_1$ ,

$$[m](P) = \underbrace{P + P + \cdots + P}_m.$$

The map  $\hat{\sigma}$  is called the dual isogeny to  $\sigma$  and is unique up to isomorphism. An important property of the dual isogeny is the following:

$$\hat{\hat{\sigma}} = \sigma.$$

A proof for this Theorem can be found in [Sil92][III.6].

An isogeny  $\sigma: E_1 \rightarrow E_2$  is either constant or surjective, hence

$$\sigma(E_1) = \{\mathcal{O}\} \quad \text{or} \quad \sigma(E_1) = E_2.$$

The following observations are useful for the handling of elliptic curves.

**Lemma 2.23.** (a) *Let*

$$\phi: E_1 \rightarrow E_2 \quad \text{and} \quad \psi: E_1 \rightarrow E_3$$

*be non-constant isogenies and assume that  $\phi$  is separable. If*

$$\ker \phi \subseteq \ker \psi,$$

*then there exists a unique isogeny*

$$\lambda: E_2 \rightarrow E_3$$

*such that  $\psi = \lambda \circ \phi$ .*

$$\begin{array}{ccc} E_1 & \xrightarrow{\psi} & E_3 \\ & \searrow \phi & \nearrow \lambda \\ & E_2 & \end{array}$$

(b) *Let  $E$  be an elliptic curve and let  $\Phi$  be a finite subgroup of  $E$ . Then there is a unique elliptic curve  $E'$  and a separable isogeny*

$$\phi: E \rightarrow E'$$

*such that*

$$\ker \phi = \Phi.$$

*Remark 2.24.* The elliptic curve  $E'$  from Lemma 2.23 is often denoted by the quotient  $E/\Phi$ . Further information on this can be found in [Sil92], III.4.

**Lemma 2.25.** *Let  $\phi: E_1 \rightarrow E_2$  be a non-constant separable isogeny between two elliptic curves. Then  $\phi$  is unramified and*

$$\#\ker(\phi) = \deg(\phi).$$

**Proof.** A proof can be found in [Sil92], III.4.10. □

Since elliptic curves are groups, the maps between them form groups as well. We set

$$\mathrm{Hom}(E_1, E_2) = \{\text{isogenies } \sigma: E_1 \rightarrow E_2\}.$$

Then  $\mathrm{Hom}(E_1, E_2)$  is a group under the addition law

$$(\sigma + \tau)(P) = \sigma(P) + \tau(P).$$

If we let  $E = E_1 = E_2$ , then we can also compose isogenies. Thus, we denote by

$$\mathrm{End}(E) = \mathrm{Hom}(E, E)$$

the ring with the addition as above and the multiplication given by composition,

$$(\sigma\tau)(P) = \sigma(\tau(P)).$$

The ring  $\mathrm{End}(E)$  is called the *endomorphism ring of  $E$* .

**Definition 2.26.** Let  $E$  be an elliptic curve and  $m \in \mathbb{Z}$ ,  $m \neq 0$ . The  *$m$ -torsion subgroup of  $E$* , denoted  $E[m]$ , is the set of points of order  $m$  in  $E$ ,

$$E[m] = \{P \in E: [m]P = \mathcal{O}\}.$$

The *torsion subgroup of  $E$*  is the set of all points of finite order on  $E$ .

### Elliptic Curves over Finite Fields

Elliptic curves over finite fields have some special properties that are used in Satoh's algorithm. First of all, let us define an important map. In section 2.1.3 we introduced the *Little Frobenius* on  $\mathbb{F}_q$  and  $\mathbb{Z}_q$ . This map extends to an isogeny on an elliptic curve in the following way.

**Definition 2.27.** We denote by  $E^{(p)}$  the curve defined by raising the coefficients of the equation for  $E$  to the  $p$ -th power. We can define the so-called *Little Frobenius* isogeny on  $E$  by

$$\begin{aligned} \varphi_p: E &\rightarrow E^{(p)} \\ (x, y) &\mapsto (x^p, y^p). \end{aligned}$$

In the context of elliptic curves, the name *Frobenius* is also used for another map.

**Definition 2.28.** Let  $K$  be a field of characteristic  $p > 0$  with  $q = p^d$  elements. Let  $E$  be an elliptic curve defined over  $K$ . The *Frobenius endomorphism* (in short *Frobenius*) on  $E$  is given by

$$\begin{aligned}\varphi_q: E &\rightarrow E \\ (x, y) &\mapsto (x^q, y^q).\end{aligned}$$

Note that the difference in the names arises from the fact that the *Little Frobenius* on  $E$  only takes the point coordinates to the power  $p$ , whereas the Frobenius endomorphism takes them to the power  $q = p^d$ .

**Definition 2.29.** Over the finite field  $K = \mathbb{F}_q$ , the number of rational points on  $E$  is finite and will be denoted by  $\#E(\mathbb{F}_q)$ . The quantity  $t$  defined by

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

is called the *trace of Frobenius* at  $q$ .

The trace  $t$  of the Frobenius endomorphism  $\varphi_q$  plays a crucial role, when it comes to point counting. Note that it sometimes is defined as the unique integer  $t$  satisfying the equation

$$\varphi_q^2 - t\varphi_q + q = 0.$$

**Theorem 2.30** (Hasse). *Let  $\mathbb{F}_q$  be the finite field with  $q = p^d$  elements. Then the trace  $t$  of Frobenius satisfies*

$$|t| \leq 2\sqrt{q}.$$

The next Theorem shows that the values of  $E[p]$  and  $\text{End}(E)$  are closely related.

**Theorem 2.31.** *Let  $K$  be a finite field of characteristic  $p$  and  $E/K$  an elliptic curve. For each integer  $r \geq 1$ , let*

$$\varphi_d: E \rightarrow E^{(p^d)} \quad \text{and} \quad \hat{\varphi}_d: E^{(p^d)} \rightarrow E$$

be the  $p^r$ -power Frobenius isogeny and its dual. Then the following properties are equivalent:

- (i)  $E[p^d] = 0$  for one (all)  $r \geq 1$ , that is, the curve is supersingular.
- (ii) The map  $\hat{\varphi}_d$  is purely inseparable for one (all)  $d \geq 1$ .
- (iii) The map  $[p]: E \rightarrow E$  is purely inseparable and  $j(E) \in \mathbb{F}_{p^2}$ .

If none of these equivalent conditions hold, the curve is ordinary and

$$E[p^d] = \mathbb{Z}/p^d\mathbb{Z} \quad \text{for all } d \geq 1.$$

### 2.2.3 Reduction

We next look at the operation of reduction modulo  $p$ . For example, the natural reduction map

$$\mathbb{Z}_q \rightarrow \mathbb{F}_q = \mathbb{Z}_q/p\mathbb{Z}_q$$

is denoted  $t \rightarrow \tilde{t}$ . We represent a curve  $\mathcal{E}$  over  $\mathbb{Q}_q$  as

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

Now we reduce its coefficients modulo  $p$  to obtain a (possibly singular) curve over  $\mathbb{F}_q$ , namely

$$E: y^2 = x^3 + \tilde{a}x + \tilde{b}.$$

**Definition 2.32.** The curve  $E$  over  $\mathbb{F}_q$  is called *reduction of  $\mathcal{E}$  modulo  $p$* . The equation for  $E$  is unique up to the standard change of coordinates ([Sil92], III.3.1b) for Weierstrass equations over  $\mathbb{F}_q$ .

We assume the curves  $\mathcal{E}$  to have *good* reduction modulo  $p$ , that is, the resulting curves  $E$  to be non-singular.

### 2.2.4 The Canonical Lift

For an ordinary elliptic curve  $E$  over the finite field  $\mathbb{F}_q$  we denote its canonical lift by  $\mathcal{E}$  which is an elliptic curve over some unramified extension of  $\mathbb{Q}_p$ .

**Definition 2.33.** Let  $E$  over  $\mathbb{F}_q$  be an elliptic curve. The *canonical lift*,  $\mathcal{E}$ , of  $E$  is characterized by the following two properties:

1. The reduction of  $\mathcal{E}$  modulo  $p$  of  $\mathcal{E}$  is  $E$ ,
2.  $\text{End}(E) \cong \text{End}(\mathcal{E})$  as a ring.

A classical result of Deuring [Deu41] shows that  $\mathcal{E}$  exists and is unique up to isomorphism.

The following result, which can be found in [Mes72], Cor. V.3.4., is relevant when one wants to lift a dual isogeny.

**Theorem 2.34.** *Let  $E, E'$  be two ordinary elliptic curves over  $\mathbb{F}_q$  and  $\mathcal{E}, \mathcal{E}'$  their respective canonical liftings. Then*

$$\text{Hom}(E, E') \cong \text{Hom}(\mathcal{E}, \mathcal{E}').$$

The following Proposition provides more details on the structure of the homomorphism ring of an elliptic curve.

**Proposition 2.35.** *As in Theorem 2.34, let  $E, E'$  be two ordinary elliptic curves over  $\mathbb{F}_q$  and  $\mathcal{E}, \mathcal{E}'$  their respective canonical liftings. Denote the isomorphism between  $\text{End}(E)$  and  $\text{End}(\mathcal{E})$  by*

$$\varphi: \text{End}(E) \rightarrow \text{End}(\mathcal{E}).$$

*Then there exist*

$$\begin{aligned} \psi: \text{Hom}(E, E') &\rightarrow \text{Hom}(\mathcal{E}, \mathcal{E}') \\ \hat{\psi}: \text{Hom}(\mathcal{E}, \mathcal{E}') &\rightarrow \text{Hom}(E, E') \end{aligned}$$

*with the following properties. Let  $\Lambda \in \text{Hom}(\mathcal{E}, \mathcal{E}')$  and  $\lambda \in \text{Hom}(E, E')$  such that*

$$\psi(\Lambda) = \lambda.$$

(i) *Then*

$$\hat{\psi}(\hat{\Lambda}) = \hat{\lambda}.$$

(ii) *Furthermore,  $\psi$  and  $\hat{\psi}$  satisfy the following equation:*

$$\varphi(\hat{\Lambda} \circ \Lambda) = \hat{\psi}(\hat{\Lambda}) \circ \psi(\Lambda) = \hat{\lambda} \circ \lambda \tag{2.3}$$

$$\begin{array}{ccccc} \mathcal{E} & \xrightarrow{\Lambda} & \mathcal{E}' & \xrightarrow{\hat{\Lambda}} & \mathcal{E} \\ \pi \downarrow & & \pi \downarrow & & \pi \downarrow \\ E & \xrightarrow{\lambda} & E' & \xrightarrow{\hat{\lambda}} & E \end{array}$$

### 2.2.5 Division Polynomials

The significance of the division polynomials can be seen in Corollary 2.38. It tells us that, for odd  $m$ , the division polynomial  $\psi_m(x)$  has as roots exactly the points of order  $m$  on  $E$ .

**Definition 2.36.** Let  $E: y^2 = x^3 + ax + b$ . Then we define the *division polynomials*

$$\psi_m \in \mathbb{Z}[a, b, x, y]$$

inductively as follows:

$$\begin{aligned} \psi_1 &= 1, \\ \psi_2 &= 2y, \\ \psi_3 &= 3x^4 + 6ax^2 + 12bx - a^2, \\ \psi_4 &= 4y(x^6 + 5ax^4 + 20bx^3 - 5a^2x^2 - 4abx - 8b^2 - a^3), \\ \psi_{(2m+1)} &= \psi_{(m+2)}\psi_m^3 - \psi_{(m-1)}\psi_{(m+1)}^3 \quad (m \geq 2), \\ \psi_{(2m)} &= \psi_m(\psi_{(m+2)}\psi_{(m-1)}^2 - \psi_{(m-2)}\psi_{(m+1)}^2) \quad (m \geq 3). \end{aligned}$$

One can show by induction that for even  $m$  the polynomial  $\psi_m(x, y)$  is divisible by  $\psi_2 = 2y$ . We define for  $m > 2$

$$f_m = \begin{cases} \psi_m & \text{if } m \text{ odd} \\ \frac{\psi_m}{2y} & \text{if } m \text{ even} \end{cases}$$

The polynomial  $f_m(x, y)$  defines a function on  $E$ .

**Proposition 2.37.** *Denote by  $\bar{K}$  the algebraic closure of a field  $K$ . Consider an elliptic curve*

$$E(\bar{K}): y^2 = x^3 + ax + b$$

over  $\bar{K}$ . Then the function

$$f_m: (x, y) \mapsto f_m(x, y)$$

has a factor

$$\sum_{P \in E[m](\bar{K})} \deg_i[m](P) - m^2(\mathcal{O}),$$

where  $\deg_i[m]$  denotes the inseparable degree of the isogeny  $[m]: E \rightarrow E$ .

Assuming in the following that  $m$  is odd, one can prove by induction that the variable  $y$  in  $f_m(x, y)$  occurs only with even exponent. Successively substituting  $y^2$  by  $x^3 + ax + b$  we get a univariate polynomial in the variable  $x$ . We denote the resulting polynomial by  $\psi_m(x)$ . Let again  $\bar{K}$  denote the algebraic closure of  $K$  and choose  $S \subset E[m](\bar{K})$  such that

$$S \cap (-S) = \emptyset \quad \text{and} \quad E[m](\bar{K}) = S \cup (-S) \cup \{\mathcal{O}\}.$$

**Corollary 2.38** (Skjernaa). *There exists a constant  $c \in K$  such that*

$$\psi_m(x) = c \prod_{P \in S} (x - x_P)^{\deg_i[m]},$$

where  $x_P$  denotes the  $x$ -coordinate of  $P$ . If  $\deg_i[m] = 1$ , then

$$\deg(\psi_m) = \frac{m^2 - 1}{2} \quad \text{and} \quad c = m.$$

A proof can be found in [Car04].

**Characteristic Two** Considering elliptic curves over finite fields of characteristic two, they are of the form

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

Thus in (2.1), we have  $a_1 = 1$ ,  $a_3 = a_4 = 0$  and consequently  $b_2 = 1$ ,  $b_4 = b_6 = 0$ ,  $b_8 = a_6$ . The recursive formulas for the polynomials  $\psi_m$  are

$$\begin{aligned}\psi_1 &= 1, \\ \psi_2 &= x, \\ \psi_3 &= x^4 + x^3 + a_6, \\ \psi_4 &= x^6 + a_6x^2, \\ \psi_{(2m+1)} &= \psi_{(m+2)}\psi_m^3 + \psi_{(m-1)}\psi_{(m+1)}^3, \quad m \geq 2, \\ \psi_{(2m)} &= \psi_m(\psi_{(m+2)}\psi_{(m-1)}^2 + \psi_{(m-2)}\psi_{(m+1)}^2), \quad m \geq 3.\end{aligned}$$

Again, one can show by induction that for even  $m$  the polynomial  $\psi_m(x)$  is divisible by  $\psi_2 = x$ . We define for  $m > 2$

$$f_m = \begin{cases} \psi_m & \text{if } m \text{ odd} \\ \frac{\psi_m}{x} & \text{if } m \text{ even} \end{cases}$$

### 2.3 Modular Polynomials

One of the main steps in Satoh's algorithm uses a system of modular polynomial equations. In this section, we will briefly explain their important properties.

Let  $E_1$  and  $E_2$  be two elliptic curves over a finite field  $K$ . Then we can easily test if there exists a cyclic isogeny of degree  $n$  between  $E_1$  and  $E_2$ .

**Theorem 2.39.** *Let  $n \in \mathbb{N}$  and define  $d$  by*

$$d = n \prod_{p|n} \left(1 + \frac{1}{p}\right).$$

*Then there exists a symmetric polynomial  $\Phi_n(X, Y) \in \mathbb{Z}[X, Y]$  of degree  $d$  such that the following properties are equivalent:*

- (i)  $E_1$  and  $E_2$  are related via a cyclic isogeny of degree  $n$
- (ii)  $\Phi_n(j(E_1), j(E_2)) = 0$ .

If  $n$  is a prime  $l$ , the degree of  $\Phi_l$  equals  $d = l + 1$ , as one can easily see by considering the definition of the degree stated in the above Theorem. Furthermore,  $\Phi_l$  satisfies the so-called *Kronecker relation*

$$\Phi_l(X, Y) \equiv (X^l - Y)(X - Y^l) \pmod{l}.$$

The polynomial  $\Phi_n(X, Y)$  is called the  *$n$ -th modular polynomial* and can be computed using the Fourier expansion of the modular function  $j(\tau)$ . Further information regarding this aspect can be found in [BSS99], [Sch95] and [Sil94].

The symmetry of  $\Phi_n$  will be important in its later application in the algorithm.

**Examples** The coefficients become rather large as  $p$  increases, as one can see in the modular equations for  $\Phi_2$  and  $\Phi_3$  presented here.

$$\begin{aligned}\Phi_2(X, Y) &= X^3 + Y^3 - X^2Y^2 + c_1(XY^2 + X^2Y) - c_2(X^2 + Y^2) \\ &+ c_3XY + c_4(X + Y) - c_5,\end{aligned}$$

where

$$\begin{aligned}c_1 &= 1488, & c_2 &= 162000, & c_3 &= 40773375, \\ c_4 &= 8748000000, & c_5 &= 157464000000000.\end{aligned}$$

$$\begin{aligned}\Phi_3(X, Y) &= X^4 + Y^4 - X^3Y^3 + d_1(X^3Y^2 + X^2Y^3) - d_2(X^3Y + XY^3) \\ &+ d_3(X^3 + Y^3) + d_4X^2Y^2 + d_5(X^2Y + XY^2) + d_6(X^2 + Y^2) \\ &- d_7XY + d_8(X + Y),\end{aligned}$$

where

$$\begin{aligned}d_1 &= 2232, & d_2 &= 1069956, & d_3 &= 36864000, \\ d_4 &= 2587918086, & d_5 &= 8900222976000, & d_6 &= 452984832000000, \\ d_7 &= 770845966336000000, & d_8 &= 1855425871872000000000.\end{aligned}$$

## 2.4 Newton Iteration

Some of the inner computations in Satoh's algorithm are based on Newton iteration. Therefore, we will present it here.

Newton iteration is a root-finding algorithm that uses the first two terms of the Taylor series of a function  $f(x)$  in the vicinity of a suspected root. It is related to *Hensel's lemma* in such a way that it forms an algorithm out of the results from the lemma.

### 2.4.1 The General Case

The Taylor series of  $f$  about the point  $x_0 + \varepsilon$  is given by

$$\begin{aligned}f(x_0 + \varepsilon) &= \sum_{n=0}^{\infty} \frac{1}{n!} \varepsilon^n f^{(n)}(x_0) \\ &= f(x_0) + \varepsilon f'(x_0) + \frac{1}{2} \varepsilon^2 f''(x_0) + \dots\end{aligned}$$

Keeping terms only to first order,

$$f(x_0 + \varepsilon) \approx f(x_0) + \varepsilon f'(x_0). \quad (2.4)$$

This expression can be used to estimate the amount of the offset  $\varepsilon$  needed to find a better approximation to the root starting from an initial guess  $x_0$ . Setting  $f(x) = f(x_0 + \varepsilon) = 0$  and solving (2.4) for  $\varepsilon \equiv \varepsilon_0$  gives

$$\varepsilon_0 = -\frac{f(x_0)}{f'(x_0)},$$

which is the first-order adjustment to the root's position. By setting  $x_1 = x_0 + \varepsilon_0$ , calculating a new  $\varepsilon_1$ , and so on, the process can be repeated until it converges to a root.

In this way, we find a new  $x_{n+1}$  in each iteration step, by calculating

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)},$$

for  $n = 1, 2, 3, \dots$ . An initial point that provides safe convergence of Newton's method is called an *approximate zero*.

If the iteration converges to a zero, it does so quadratically, that is, the precision approximately doubles after each iteration.

### 2.4.2 The $p$ -Adic Case

We will now consider the  $p$ -adic case, since this is the specific iteration we will need in the algorithm. Let  $R$  be a  $p$ -adic ring. The following Newton iteration allows us to improve an approximate root of a polynomial. Given an already good approximation to a root, we can efficiently compute a better approximation with approximately twice the precision.

Let us first take a look at the meaning of the word *precision*. We write

$$p^k | x \quad \text{if} \quad x \equiv 0 \pmod{p^k}.$$

In this case,  $p^{-k}x$  is in  $R$ . We write

$$p^k \parallel x \quad \text{if} \quad p^k | x \quad \text{but} \quad p^{k+1} \nmid x.$$

In such a case,  $p^{-k}x$  is invertible in  $R$ .

Let  $x \in R$  and  $f \in R[t]$ . Let  $k \in \mathbb{N}$  be such that  $p^k \parallel f'(x)$  and assume  $p^{n+k} | f(x)$  for some  $n > k$ . Under these conditions, we will say that  $x$  is an *approximate root of  $f$  known to precision  $n$* . This is meant in the sense that there exists an exact root  $y$  of  $f$  in  $R$  with  $x \equiv y \pmod{p^n}$ .

We will first state a lemma showing how a Newton iteration improves a root known to precision  $n$  into one known to precision  $2n - k$ .

**Lemma 2.40.** *Let  $x \in R$  and  $f \in R[t]$ . Let  $k \in \mathbb{N}$  be such that  $p^k \parallel f'(x)$  and assume that  $p^{n+k} | f(x)$  for some natural number  $n > k$ . Define*

$$\Delta := \frac{f(x)}{f'(x)}$$

and  $y = x - \Delta$ .

*Then  $y \equiv x \pmod{p^n}$ ,  $p^{2n} | f(y)$  and  $p^k \parallel f'(y)$ .*

The definition of  $\Delta$  corresponds to the summand  $\varepsilon$  in the standard Newton iteration. The difference in the  $p$ -adic case is the division by  $f'(x)$ , which in general is not possible in a ring. We therefore make the assumption  $p^k \parallel f'(x)$ , so that  $p^{-k}f'(x)$  becomes invertible in  $R$ , as we remarked earlier.

The conclusions  $p^{2n} \mid f(y)$  and  $p^k \parallel f'(y)$  ensure, that, by our above notation,  $y$  is a root known to precision  $2n - k$ .

Repeated application of this lemma gives rise to an algorithm to compute a root of a polynomial to any desired (but finite) precision, from a sufficiently precise initial root.

The input values for the algorithm `NEWTONITERATIONS` ([FGH00]) are a desired precision  $n$ , a polynomial  $f \in R[t]$ , a starting solution  $x_0 \in R$  and an integer  $k$  with  $p^k \parallel f'(x_0)$  and  $p^{2k+1} \mid f(x_0)$ . It returns an approximate root  $y \in R$  with  $y \equiv x_0 \pmod{p^{k+1}}$  and  $p^{n+k} \mid f(y)$ .

---

**Algorithm 1** `NewtonIterations` [FGH00]

---

```

1: procedure NEWTONITERATIONS( $n, f, x_0, k$ )
2:   if  $n \leq k + 1$  then
3:     return  $x_0$ 
4:   end if
5:    $n' \leftarrow \lceil \frac{n+k}{2} \rceil$ 
6:    $x \leftarrow \text{NEWTONITERATIONS}(n', f, x_0, k)$ 
7:    $y \leftarrow x - \frac{f(x)}{f'(x)}$ 
8:   return  $y$ 
9: end procedure

```

---

For our implementation, we will include some strategies to improve the efficiency of the algorithm. These will be described in Chapter 4.

### 2.4.3 Hensel's Lemma

In the process of the algorithm, Satoh uses a modification of Hensel's Lemma. To understand the problems which arise in the use of the original Lemma, we state it here.

Let  $K$  be a complete field with non-Archimedean norm  $|\cdot|$  and valuation ring  $R$ . Given an initial approximation  $r_0 \in R$  to a root of a polynomial  $f \in R[x]$ , then Hensel's lemma provides the existence of a unique root  $r \in R$  of  $f$  close to  $r_0$ .

**Lemma 2.41** (Hensel's Lemma). *Let  $K$  be a complete field with non-Archimedean norm  $|\cdot|$  and valuation ring  $R$ . Let  $f \in R[x]$  be a polynomial and assume that  $r_0$  satisfies*

$$|f(r_0)| < |f'(r_0)|^2.$$

*Then there exists a unique element  $r \in R$  such that*

$$f(r) = 0 \text{ and } |r - r_0| \leq \left| \frac{f(r_0)}{f'(r_0)} \right| < |f'(r_0)|.$$

The last inequality is just an application of the assumption made on  $|f(r_0)|$ .

The proof of Hensel's lemma is constructive in the sense that it provides an algorithm to compute  $r$  starting from  $r_0$ . We first define a sequence  $\{r_k\}_{k \in \mathbb{N}}$  by

$$r_{k+1} := r_k - \frac{f(r_k)}{f'(r_k)}.$$

One can prove by induction that

$$|f(r_k)| \leq c^{2^k - 1} |f(r_0)| \text{ with } c = \left| \frac{f(r_0)}{f'(r_0)^2} \right| < 1.$$

This also shows that the convergence of the sequence  $\{r_k\}_{k \in \mathbb{N}}$  is quadratic.

### 3 Satoh's Algorithm

The starting point Satoh's algorithm shares with its predecessors is the equation

$$\#E(\mathbb{F}_q) = q + 1 - t, \quad (3.1)$$

where  $|t| < 2\sqrt{q}$  denotes the trace of the  $q$ -th power Frobenius endomorphism  $F$  on  $\mathbb{F}_q$ . Since  $q$  is given, the problem of determining the number of points on  $E$  reduces to determining the trace  $t$ .

The condition  $|t| < 2\sqrt{q}$  allows us to work only up to a certain precision. This plays an important role in the implementation.

Apart from that, Satoh's point counting algorithm is of quite a different nature than the former methods. Algorithms like Schoof or SEA determine the trace modulo many primes  $l$  and apply the Chinese Remainder Theorem in order to compute the group order. Satoh computes it directly, *not* modulo different primes but modulo powers of one fixed prime  $p$ .

The basic idea of Satoh's algorithm is to use the lift of both the curve  $E$  and the Frobenius endomorphism  $F$  to a local ring in a canonical way. Lifting  $F$  to  $\mathcal{F}$  in a canonically preserves the trace and we can compute  $\text{Tr}(\mathcal{F})$  instead. This turns out to be easier than computing the trace of  $F$ .

We start with a curve  $E$  given over  $\mathbb{F}_q$ , where  $q = p^d$ . We want to lift this curve to a curve  $\mathcal{E}$  defined over the  $p$ -adic ring  $\mathbb{Z}_q$  with residue class field  $\mathbb{F}_q$ . Intuitively, one can think of  $\mathbb{Z}_q$  as an extension of  $\mathbb{F}_q$  in the same way as the  $p$ -adic integers  $\mathbb{Z}_p$  relate to the prime field  $\mathbb{F}_p$ . In a more precise way,  $\mathbb{Z}_q$  is the unramified valuation ring of  $\mathbb{Q}_q$ , where  $\mathbb{Q}_q$  is the unique unramified extension of degree  $d$  over  $\mathbb{Q}_p$ , with residue field  $\mathbb{F}_q$ .

A Theorem by Lubin, Serre and Tate proves the existence of a unique lifted  $j$ -invariant  $J \in \mathbb{Z}_q$  corresponding to the canonical lift  $\mathcal{E}$  of  $E/\mathbb{F}_q$ . Moreover, they obtain the equation

$$\Phi_p(J, \Sigma(J)) = 0 \quad , \text{ where } J \equiv j \pmod{p}, \quad (3.2)$$

where  $\Phi_p$  is the  $p$ -th modular polynomial as defined in Section 2.3. Note that  $\Sigma$  denotes the lifted Frobenius isogeny of  $\mathcal{E}$ . This leads to the idea of computing the lifted  $j$ -invariant by solving (3.2).

However, it has turned out that using this approach directly would be slow due to the complicated structure of the Frobenius of  $\mathbb{Z}_q$  (cp. 2.1.3).

Satoh points out a much more efficient algorithm using the following approach: rather than lifting  $j$  to  $J$  in isolation, it is much faster to lift  $j$  simultaneously along with its conjugates  $j_i (= j^{p^{d-i}})$ . For this purpose, we build a cycle of curves  $(E_i)_{0 \leq i < d}$  from  $E$ , where  $E_i = \sigma^{d-i}(E)$ .

$$E_0 \xrightarrow{\sigma_{d-1}} E_{d-1} \xrightarrow{\sigma_{d-2}} \cdots \xrightarrow{\sigma_1} E_1 \xrightarrow{\sigma_0} E_0$$

Similarly, we get a graph for the cycle of  $j$ -invariants:

$$j_0 \xrightarrow{\sigma_{d-1}} j_{d-1} \xrightarrow{\sigma_{d-2}} \cdots \xrightarrow{\sigma_1} j_1 \xrightarrow{\sigma_0} j_0$$

The equations  $\Phi_p(J, \Sigma(J)) = \Phi_p(J_{i+1}, J_i)$  for  $0 \leq i < d$ , given by the Lubin-Serre-Tate Theorem, now yield an algebraic system over  $\mathbb{Z}_q$  that does not involve Frobenius. This system can be solved using multivariate Newton iteration.

The iteration yields all the  $J_i$ 's to  $p$ -adic precision  $O(p^n)$  where  $n = \lceil \frac{d}{2} \rceil + 1$ . Next, for each  $i$ , we find a corresponding curve  $\mathcal{E}_i$  defined over  $\mathbb{Z}_q$  having  $J_i$  as its  $j$ -invariant. This time an ordinary Newton iteration suffices to compute the curves' coefficients to desired precision.

Since we lifted the curve  $E$  in a canonical way, its Endomorphism ring is preserved, and hence the trace of the Frobenius  $F$  of  $E$  is unchanged. Moreover, the trace of an endomorphism is the same as the trace of its dual. Denoting the dual of the Frobenius, the *Verschiebung*, by  $V$  and its canonical lift by  $\mathcal{V}$ , we have

$$\text{End}(E) \cong \text{End}(\mathcal{E}) \Rightarrow \text{Tr}(F) = \text{Tr}(\mathcal{F}) = \text{Tr}(\mathcal{V}).$$

As explained above, the Frobenius  $F$  can be decomposed into the product of little Frobenius isogenies which build a cycle of the curve  $E$  and its  $d$  conjugates. Similarly, we can decompose the dual of Frobenius and its lift. Thus, for  $\mathcal{V}$ , we have

$$\text{Tr}(\mathcal{V}) = \text{Tr}(\hat{\Sigma}_{d-1} \circ \hat{\Sigma}_{d-2} \circ \cdots \circ \hat{\Sigma}_0).$$

The next step is to consider the action of  $\mathcal{V}$  on the invariant differential of the curve  $\mathcal{E}$ . Let  $\omega$  be the invariant differential on  $\mathcal{E}$ . Satoh proves a proposition which gives the following very elegant and simple formula for the trace of  $\mathcal{V}$ :

$$\text{Tr}(\mathcal{V}) = c + \frac{q}{c},$$

with  $\mathcal{V}^*(\omega) = c\omega$  being the action of  $\mathcal{V}$  on  $\omega$ .

Using the decomposition of  $\mathcal{V}$ , we can compute  $c$  from the action of  $\hat{\Sigma}_i$  on the invariant differential  $\omega_{i+1}$  of  $\mathcal{E}_{i+1}$  for  $i = 0, \dots, d-1$ . For the computation of each of the  $c_i$ , the kernel of  $\hat{\Sigma}_i$  is needed. This computation is done with the help of Vélú's formulae [Vél71].

Computing  $\ker(\hat{\Sigma}_i)$  turns out to be the most delicate step of the algorithm. The idea is to find a factor of the  $p$ -division polynomial  $\Psi_p(x)$  by using a type of quadratic Hensel lift (which resembles a Newton iteration). This factor gives us the  $x$ -coordinates of the points in  $\ker(\hat{\Sigma}_i)$  which lead immediately to the trace of Frobenius.

### 3.1 The Cycle of Curves and their $j$ -Invariants

The first step is the computation of the cycle of  $d$  curves  $E_i$  and their  $j$ -invariants  $j_i$ .

The input of the algorithm was an elliptic curve  $E$  defined over a finite field  $\mathbb{F}_q$ , where  $q = p^d$  for some prime number  $p$ . Following (3.1), obtaining the number of elements on  $E$  reduces to computing the trace of the  $q$ -th power Frobenius endomorphism on  $E$ .

**Definition 3.1.** Let  $E(\mathbb{F}_q)$  be an elliptic curve. The *conjugate curve*  $E^{(p)}$  of  $E$  is obtained by applying  $\sigma$ , the  $p$ -th power Frobenius endomorphism on  $\mathbb{F}_q$ , to the coefficients of  $E$ .

In fact, the map  $\sigma: E \rightarrow E^{(p)}$  is an isogeny. Furthermore, the construction can be repeated, giving an isogeny from  $E^{(p)}$  to  $E^{(p^2)}$  and so on. Since the coefficients of  $E$  are elements of  $\mathbb{F}_q$ , applying  $\sigma$  to them  $d$  times will set them back to their original value, since the  $q$ -th power Frobenius on  $\mathbb{F}_q$  is just the identity map. Hence, repeating the construction  $d$  times will provide a cycle of isogenies returning to the initial curve  $E$ .

We write  $E_0$  for  $E$ ,  $E_{d-1}$  for  $E^{(p)}$ ,  $\sigma_{d-1}$  for the isogeny between them and so on. With this notation, we can draw the figure:

$$E_0 \xrightarrow{\sigma_{d-1}} E_{d-1} \xrightarrow{\sigma_{d-2}} \dots \xrightarrow{\sigma_1} E_1 \xrightarrow{\sigma_0} E_0$$

*Remark 3.2.* The  $q$ -th power Frobenius on  $E$  is the curve endomorphism  $F = \sigma_0 \circ \sigma_1 \circ \dots \circ \sigma_{d-1}$ .

In the same way we constructed a cycle of curves over  $\mathbb{F}_q$ , if we have a curve  $\mathcal{E}$  over  $\mathbb{Z}_q$ , the  $p$ -th power Frobenius  $\Sigma$  on  $\mathbb{Z}_q$ , see (2.1.3), can be extended to a map between  $\mathcal{E}$  and the conjugate curve  $\mathcal{E}^\Sigma$ . Repeating this construction  $d$  times brings us back to the initial curves, yielding another cycle of isogenies and curves:

$$\mathcal{E}_0 \xrightarrow{\Sigma_{d-1}} \mathcal{E}_{d-1} \xrightarrow{\Sigma_{d-2}} \dots \xrightarrow{\Sigma_1} \mathcal{E}_1 \xrightarrow{\Sigma_0} \mathcal{E}_0$$

Computing the  $j$ -invariants of each of the curves  $E_i$  is really just calculational work.

## 3.2 Lifting the $j$ -Invariants

The following Theorem by Lubin, Serre and Tate allows us to lift a curve  $E$  over  $\mathbb{F}_q$  to a curve  $\mathcal{E}$  over  $\mathbb{Z}_q$  canonically, by lifting its  $j$ -invariant.

**Theorem 3.3** (Lubin-Serre-Tate). *Let  $E$  be a curve over  $\mathbb{F}_q$  with  $j$ -invariant  $j \in \mathbb{F}_q \setminus \mathbb{F}_{p^2}$ , then there is a unique  $J \in \mathbb{Z}_q$  such that*

$$\Phi_p(J, \Sigma(J)) = 0 \text{ and } J \equiv j \pmod{p}, \quad (3.3)$$

*and  $J$  is the  $j$ -invariant of the canonical lift  $\mathcal{E}$  of  $E$  (up to isomorphism).*

Recall that the curve  $\mathcal{E}$  is the unique lift of  $E$  having the same endomorphism ring.

It is possible to use this Theorem directly by lifting  $j$  together with its conjugate  $\sigma(j)$ . Unfortunately, the computation of the polynomial  $\Phi_p(J, \Sigma(J))/p^n$ , which appears in the algorithm, requires computations of  $\Sigma$  and this is slow in practice. Furthermore,  $\Theta(d)$  such steps are required.

It turns out to be much faster to use the whole cycle of  $J_i$ 's. Writing out all the equations  $\Phi_p(J_i, J_{i+1})$  for  $0 \leq i < d$  produces an algebraic system over  $\mathbb{Z}_q$  without  $\Sigma$ . This system can be solved quickly by  $O(\log d)$  steps of a multivariate Newton iteration, which will be described later on in this section.

The hypothesis  $j(E) \notin \mathbb{F}_{p^2}$  in (3.3) is necessary to ensure that a specific partial derivative of  $\Phi_p$  does not vanish modulo  $p$ . This condition is therefore necessary to guarantee the uniqueness of the solution of (3.3).

Aside from that, the case  $j(E) \in \mathbb{F}_{p^2}$  can be solved easily. Since  $j(E) \in \mathbb{F}_{p^2}$ , there exists an elliptic curve  $E'$  defined over  $\mathbb{F}_{p^m}$  with  $m = 1$  or  $m = 2$ , which is isomorphic to  $E$  over  $\mathbb{F}_q$  (for details see [Sil92](V.2.4)). Now, let

$$t_k = p^{mk} + 1 - \#E'(\mathbb{F}_{p^{mk}}),$$

then

$$t_{k+1} = t_1 t_k - p^m t_{k-1}$$

with  $t_0 = 2$  and therefore

$$\#E(\mathbb{F}_q) = p^n + 1 - t_{n/m}.$$

We can therefore leave this case out of our considerations. Since supersingularity implies that  $j(E) \in \mathbb{F}_{p^2}$  ([Sil92], V.3.1), cancelling out this possibility also implies that we will not be working with supersingular curves,

The cycle of lifted curves is:

$$\begin{array}{ccccccccc} \mathcal{E}_0 & \xrightarrow{\Sigma_{d-1}} & \mathcal{E}_{d-1} & \xrightarrow{\Sigma_{d-2}} & \dots & \xrightarrow{\Sigma_1} & \mathcal{E}_1 & \xrightarrow{\Sigma_0} & \mathcal{E}_0 \\ \pi \downarrow & & \pi \downarrow & & & & \pi \downarrow & & \\ E_0 & \xrightarrow{\sigma_{d-1}} & E_{d-1} & \xrightarrow{\sigma_{d-2}} & \dots & \xrightarrow{\sigma_1} & E_1 & \xrightarrow{\sigma_0} & E_0 \end{array}$$

With  $F$  being the composition of the  $\sigma_i$ , the curve endomorphism  $\mathcal{F}$ , as the lift of  $F$ , is the composition of the  $\Sigma_i$ . Hence, this diagram can be shortly written as:

$$\begin{array}{ccc} \mathcal{E} & \xrightarrow{\mathcal{F}} & \mathcal{E} \\ \pi \downarrow & & \pi \downarrow \\ E & \xrightarrow{F} & E \end{array}$$

Due to difficulties that arise when one wants to lift  $\sigma$ , we consider its dual isogeny  $\hat{\sigma}$ , which is called the *Verschiebung*. Since we are working with ordinary curves, the map  $\hat{\sigma}$  is separable, whereas  $\sigma$  is purely inseparable. The lift of an inseparable map is very difficult, whereas a separable map can be lifted in a canonical way. Satoh makes use of this property by working with the dual isogeny, since it can be lifted easily via its kernel. This lift will be discussed in detail in 3.3.2.

When considering the duals, we get the same cycles of curves, just in the opposite order:

$$\begin{array}{ccccccccc} \mathcal{E}_0 & \xrightarrow{\hat{\Sigma}_0} & \mathcal{E}_1 & \xrightarrow{\hat{\Sigma}_1} & \dots & \xrightarrow{\hat{\Sigma}_{d-2}} & \mathcal{E}_{d-1} & \xrightarrow{\hat{\Sigma}_{d-1}} & \mathcal{E}_0 \\ \pi \downarrow & & \pi \downarrow & & & & \pi \downarrow & & \\ E_0 & \xrightarrow{\hat{\sigma}_0} & E_1 & \xrightarrow{\hat{\sigma}_1} & \dots & \xrightarrow{\hat{\sigma}_{d-2}} & E_{d-1} & \xrightarrow{\hat{\sigma}_{d-1}} & E_0 \end{array} \tag{3.4}$$

Naturally, we also get the dual endomorphism  $\hat{F} = \hat{\sigma}_{d-1} \circ \hat{\sigma}_{d-2} \circ \cdots \circ \hat{\sigma}_0$  as the composition of the dual isogenies  $\hat{\sigma}_i$ 's and its lift  $\hat{\mathcal{F}}$ :

$$\begin{array}{ccc} \mathcal{E} & \xrightarrow{\hat{\mathcal{F}}} & \mathcal{E} \\ \pi \downarrow & & \pi \downarrow \\ E & \xrightarrow{\hat{F}} & E \end{array}$$

The existence of a lift of  $\hat{\sigma}_i$  is not trivial and follows from 2.34 which can be found in [Mes72].

The Theorem of Lubin, Serre and Tate ensures that the cycle of  $J_i$  we want to construct is characterized by  $\Phi_p(J_i, \Sigma(J_i)) = 0$ , that is,  $\Phi(J_i, J_{i+1}) = 0$  for all  $0 \leq i < d$ , where the indices are taken modulo  $d$ . Computing this cycle from the cycle of the  $j_i$  is done by applying a Newton iteration based on the function  $\Theta: \mathbb{Z}_q^d \rightarrow \mathbb{Z}_q^d$  defined by

$$\Theta(x_0, \dots, x_{d-1}) = (\Phi_p(x_0, x_1), \Phi_p(x_1, x_2), \dots, \Phi_p(x_{d-1}, x_0)). \quad (3.5)$$

Then we clearly have

$$\Theta(J(\mathcal{E}_0), J(\mathcal{E}_1), \dots, J(\mathcal{E}_{d-1})) = (0, 0, \dots, 0). \quad (3.6)$$

The basic Newton iteration algorithm can be adapted to this multivariate case rather easily. The consideration of the derivative of  $\Theta$  leads to its Jacobian matrix

$$D\Theta(x_0, \dots, x_{d-1}) = \begin{pmatrix} \Phi'_p(x_0, x_1) & \Phi'_p(x_1, x_0) & 0 & \cdots & 0 \\ 0 & \Phi'_p(x_1, x_2) & \Phi'_p(x_2, x_1) & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & \Phi'_p(x_{d-2}, x_{d-1}) \\ \Phi'_p(x_0, x_{d-1}) & 0 & 0 & \cdots & \Phi'_p(x_{d-1}, x_0) \end{pmatrix},$$

where  $\Phi'_p(X, Y)$  denotes the partial derivative of  $\Phi_p(X, Y)$  with respect to  $X$ . Note that  $\Phi'_p(Y, X)$  is the partial derivative of  $\Phi_p(X, Y)$  with respect to  $Y$  since  $\Phi_p(X, Y)$  is symmetric. The iteration is essentially done as follows:

$$(x_0, \dots, x_{d-1}) \leftarrow (x_0, \dots, x_{d-1}) - ((D\Theta)^{-1}\Theta)(x_0, \dots, x_{d-1}). \quad (3.7)$$

During the iteration, one has to take care of the working precision at each stage. Also, we need to consider the difficulty of inverting the Jacobian matrix.

If the computations are organized in a suitable way, the inversion can be done with a linear (in  $d$ ) number of arithmetic operations. The diagonal elements  $\Phi'_p(x_i, x_{i+1})$  are all invertible, so elimination can be done without having to look for a pivot. Furthermore, the off-diagonal elements are all divisible by  $p$ . This ensures that the matrix is diagonally dominant which in turn implies the invertibility of  $D\Theta$ .

To compute the  $j$ -invariants of the lifted curves to precision  $n$ , we apply a Newton iteration to the function  $\Theta$  as defined above. We wish to find a root of  $\Theta$  to precision  $n$ . As mentioned before, we have that

$$\Theta(J(\mathcal{E}_0), J(\mathcal{E}_1), \dots, J(\mathcal{E}_{d-1})) = (0, 0, \dots, 0).$$

Hence finding a root  $(x_0, x_1, \dots, x_{d-1})$  of  $\Theta$  to precision  $n$  would mean to have found the cycle of  $j$ -invariants of the lifted curves  $\mathcal{E}_i \pmod{p^n}$ .

Starting with the desired precision  $n$  and a cycle  $j_i \in \mathbb{F}_q \setminus \mathbb{F}_{p^2}$  such that

$$\Phi_p(j_i, j_{i+1}) \equiv 0 \pmod{p} \text{ for all } 0 \leq i < d,$$

we wish to retrieve a cycle  $J_i \in \mathbb{Z}_q$  such that

$$\pi(J_i) = j_i \text{ and } \Phi_p(J_i, J_{i+1}) \equiv 0 \pmod{p^n} \text{ for all } 0 \leq i < d.$$

The  $p$ -th modular polynomial satisfies the Kronecker relation

$$\Phi_p(X, Y) \equiv (X^p - Y)(X - Y^p) \pmod{p}, \quad (3.8)$$

and we can derive a formula for the partial derivative mod  $p$ , with respect to  $X$  we obtain:

$$\Phi'_p(X, Y) \equiv (p+1)X^p - Y - pX^{p-1}Y^p \equiv X^p - Y \pmod{p}.$$

Now since  $j_i \notin \mathbb{F}_{p^2}$  and  $j_i = j_{i+1}^p$ , we have

$$\begin{aligned} \Phi'_p(j_i, j_{i+1}) &\equiv (j_{i+1})^{p^2} - j_i \not\equiv 0 \pmod{p}, \\ \Phi'_p(j_{i+1}, j_i) &\equiv (j_{i+1})^p - (j_{i+1})^p \equiv 0 \pmod{p}. \end{aligned}$$

The above equations imply that  $D\Theta(x_0, \dots, x_{d-1}) \pmod{p}$  is a diagonal matrix with non-zero diagonal elements. Therefore, the Jacobian matrix is invertible over  $\mathbb{Z}_q$  and we conclude that

$$\delta := ((D\Theta)^{-1}\Theta)(x_0, \dots, x_{d-1}) \in \mathbb{Z}_q^d.$$

Note that we can simply apply Gauss elimination to solve

$$(D\Theta)(x_0, x_1, \dots, x_{d-1})\delta = \Theta(x_0, x_1, \dots, x_{d-1})$$

since the diagonal elements are all invertible. Using row operations we move the bottom left element  $\Phi'_p(x_0, x_{d-1})$  towards the right. After  $n$  row operations this element becomes

$$(-1)^n \Phi'_p(x_0, x_{d-1}) \prod_{i=0}^{n-1} \frac{\Phi'_p(x_{i+1}, x_i)}{\Phi'_p(x_i, x_{i+1})}, \quad (3.9)$$

which clearly is divisible by  $p^n$  since  $\Phi'_p(x_{i+1}, x_i) \equiv 0 \pmod{p}$  for all  $0 \leq i < d$ .

Hence it rapidly becomes zero within working precision. Then the matrix has a simple upper triangular form which can be solved directly, from bottom to top.

The input values for the algorithm LIFTJINVARIANTS ([FGH00]) are a desired precision  $n$  and a cycle  $j_i \in \mathbb{F}_q \setminus \mathbb{F}_{p^2}$  such that  $\Phi_p(j_i, j_{i+1}) \equiv 0 \pmod{p}$  for all  $0 \leq i < d$ . It returns a cycle  $J_i \in \mathbb{Z}_q$  such that  $\pi(J_i) = j_i$  and  $\Phi_p(J_i, J_{i+1}) \equiv 0 \pmod{p^n}$  for all  $0 \leq i < d$ .

**Algorithm 2** LiftJInvariants [FGH00]

---

```

1: procedure LIFTJINVARIANTS( $n, (j_i)$ )
2:   if  $n = 1$  then
3:     Pick arbitrary  $J_i$  such that  $\pi(J_i) = j_i$ 
4:     return  $(J_0, J_1, \dots, J_{d-1})$ 
5:   end if
6:    $n' \leftarrow \lceil \frac{n}{2} \rceil$ 
7:    $(J_0, J_1, \dots, J_{d-1}) \leftarrow$  LIFTJINVARIANTS( $n', (j_0, j_1, \dots, j_{d-1})$ )
8:    $(J_0, J_1, \dots, J_{d-1}) \leftarrow$  UPDATEJS( $n, (J_0, J_1, \dots, J_{d-1})$ )
9:   return  $(J_0, J_1, \dots, J_{d-1})$ 
10: end procedure

1: procedure UPDATEJS( $n, (j_i)$ )
2:   Allocate arrays  $D[0 \dots (d-2)]$ ,  $P[0 \dots (d-1)]$  and  $\mathfrak{J}[0 \dots (d-1)]$  of elements of  $\mathbb{Z}_q$ 
    $\triangleright D$  stores the off-diagonal entries,  $P$  contains the result vector and  $\mathfrak{J}$  will contain the
   updated cycle of  $j$ -invariants.
3:   for  $i = 0$  to  $d-2$  do
4:      $t \leftarrow 1/\Phi'(J_i, J_{i+1})$ 
5:      $D_i \leftarrow t \cdot \Phi'(J_{i+1}, J_i)$   $\triangleright$  Divide rows by the diagonal entries.
6:      $P_i \leftarrow t \cdot \Phi(J_i, J_{i+1})$   $\triangleright$  Divide the result vector by the diagonal entries.
7:   end for
8:    $m \leftarrow \Phi'(J_0, J_{d-1})$   $\triangleright$  The matrix' lower left entry.
9:    $f \leftarrow \Phi(J_{d-1}, J_0)$   $\triangleright$  The result vector's bottom entry.
10:  for  $i = 0$  to  $d-2$  do
11:     $f \leftarrow f - m \cdot P_i$   $\triangleright$  Introduce zeros in the last row.
12:     $m \leftarrow -m \cdot D_i$   $\triangleright$  Move the lower left entry to the right.
13:    if  $m \equiv 0 \pmod{p^n}$  then
14:      go to ..  $\triangleright$  We can stop here, since  $m$  is zero to the desired precision.
15:    end if
16:  end for
17:   $m \leftarrow m + \Phi'(J_{d-1}, J_0)$   $\triangleright$  Form the bottom right element from  $m$  and the former
  lower right entry.
18:   $P_{d-1} \leftarrow f/m$   $\triangleright$  Retrieve the result vector.
19:  for  $i = d-2$  to  $0$  do
20:     $P_i \leftarrow P_i - D_i \cdot P_{i+1}$   $\triangleright$  Solve the system by inserting what is known.
21:  end for
22:  for  $i = 0$  to  $d-1$  do
23:     $\mathfrak{J}_i \leftarrow J_i - P_i$   $\triangleright$  Update the  $j$ -invariants.
24:  end for
25:  return  $(\mathfrak{J}_0, \mathfrak{J}_1, \dots, \mathfrak{J}_{d-1})$ 
26: end procedure

```

---

### 3.3 Characteristic $p \geq 5$

In this Section, we describe Satoh's original algorithm for point counting on elliptic curves over finite fields of characteristic  $p \geq 5$ .

#### 3.3.1 Lifting each Curve

When the characteristic  $p$  is at least 5, we consider that the elliptic curve  $E$  over  $\mathbb{F}_q$  has an equation of the form

$$E: y^2 = x^3 + ax + b.$$

In this case, we want to lift  $E$  to a curve

$$\mathcal{E}: y^2 = x^3 + Ax + B$$

over  $\mathbb{Z}_q$  whose  $j$ -invariant is already known. Fouquet, Gaudry and Harley proposed to do this by arbitrarily lifting one of the coefficients and lift the other with a univariate Newton iteration based on the equation

$$J = \frac{-1728(4A)^3}{\Delta} \text{ where } \Delta = -16(4A^3 + 27B^2),$$

yielding

$$J = \frac{6912A^3}{4A^3 + 27B^2}.$$

Note that the hypotheses  $p \geq 5$  and  $J \notin \mathbb{F}_{p^2}$  ensure that the denominator in the iteration is non-zero. Hence, the precision exactly doubles at each iteration.

Skjernaa suggested to simply take  $A = 3\gamma$  and  $B = 2\gamma$  with

$$\gamma = \frac{j(E)}{1728 - j(E)}.$$

This model is minimal, has the correct  $j$ -invariant and reduces to  $E$ . In fact, it requires only one inversion in  $\mathbb{Z}_q$ , so this method is much faster than the above algorithm.

#### 3.3.2 Lifting each Curve's Torsion Subgroup

To finally compute the trace of Frobenius, we recall that the canonical lift  $\mathcal{E}$  of an elliptic curve  $E$  over  $\mathbb{F}_q$  has the property that  $\text{End}(\mathcal{E}) \cong \text{End}(E)$ . If  $F$  is the Frobenius endomorphism on  $E$  and  $\mathcal{F}$  the image of  $F$  under this ring isomorphism, then we have  $\text{Tr}(F) = \text{Tr}(\mathcal{F})$ .

Furthermore, since  $\mathbb{Q}_q$  has characteristic zero, we will be able to compute the exact value of  $\text{Tr}(F)$  and not just modulo  $p$ . The following proposition by Satoh gives a relation between the trace of an endomorphism  $f \in \text{End}_{\mathbb{Q}_q}(E)$  and its action on the invariant differential of  $E$ .

**Proposition 3.4** (Sato). *Let  $\mathcal{E}$  be an elliptic curve over  $\mathbb{Q}_q$  having good reduction modulo  $p$  and let  $f \in \text{End}_{\mathbb{Q}_q}(E)$  be an endomorphism of degree  $d$ . Let  $\omega$  be the invariant differential on  $E$  and let  $f^*(\omega) = c\omega$  be the action of  $f$  on  $\omega$ . Then the trace of  $f$  is given by*

$$\text{Tr}(f) = c + \frac{d}{c}.$$

In our case, we want to apply the above proposition to the Frobenius endomorphism  $\mathcal{F}$ . We get that

$$\text{Tr}(\mathcal{F}) = \gamma + \frac{q}{\gamma} \quad \text{with} \quad \mathcal{F}^*(\omega) = \gamma\omega.$$

**Lemma 3.5.** *With the above notation, either  $\gamma$  or  $\frac{q}{\gamma}$  is a unit in  $\mathbb{Z}_q$ .*

**Proof.** Recall that  $E$  is an ordinary elliptic curve. Suppose neither of  $\gamma$  and  $\frac{q}{\gamma}$  was a unit in  $\mathbb{Z}_q$ . Then

$$\gamma, \frac{q}{\gamma} \in p\mathbb{Z}_q.$$

Hence, their sum  $\gamma + \frac{q}{\gamma}$  lies in  $p\mathbb{Z}_q$ , too, since  $p\mathbb{Z}_q$  is an ideal. But  $\gamma + \frac{q}{\gamma} \in \mathbb{Z}_p$ , and hence

$$\gamma + \frac{q}{\gamma} \in p\mathbb{Z}_q \cap \mathbb{Z}_p = p.$$

This would imply that  $p \mid \text{Tr}(\mathcal{F}) = \text{Tr}(F)$  which is impossible, since  $E$  is ordinary.

Hence one of  $\gamma$  and  $\frac{q}{\gamma}$  has to be a unit in  $\mathbb{Z}_q$ .  $\square$

However, since  $\mathcal{F}$  is inseparable,  $\mathcal{F}^*(\omega) = 0$  (by [Sil92], II.4.2). Hence  $\gamma \equiv 0 \pmod{p}$ , which implies  $\gamma \equiv 0 \pmod{q}$  since  $\frac{q}{\gamma}$  has to be a unit in  $\mathbb{Z}_q$  by lemma (3.5).

So, if we tried to compute  $\text{Tr}(\mathcal{F}) \pmod{p^N}$ , we would need to determine  $\gamma \pmod{p^{d+N}}$ . It also turns out to be quite difficult to compute  $\gamma$  directly. We will show later that it would be necessary to know  $\ker(\Sigma_i)$ . However, since  $\ker(\sigma_i)$  is trivial, it cannot be simply lifted to  $\ker(\Sigma_i)$ , but we would need to factor the  $p$ -division polynomial of  $\mathcal{E}$ .

To avoid these problems, Sato works with the Verschiebung  $V$ , which is separable since  $E$  is ordinary. Thus  $\ker(V)$  can be lifted to  $\ker(\mathcal{V})$  in a canonical way. Furthermore, the trace of an endomorphism equals the trace of its dual, so we have

$$\text{Tr}(\mathcal{F}) = \text{Tr}(\mathcal{V}) = c + \frac{q}{c}$$

with  $\mathcal{V}^*(\omega) = c\omega$  and  $c$  a unit in  $\mathbb{Z}_q$ , by a similar argument as above.

As we saw before,  $\mathcal{V}$  can be written as

$$\mathcal{V} = \hat{\Sigma}_{d-1} \circ \hat{\Sigma}_{d-2} \circ \cdots \circ \hat{\Sigma}_0$$

and thus we can compute  $c$  from the action of  $\hat{\Sigma}_i$  on the invariant differential  $\omega_{i+1}$  of  $E_{i+1}$  for  $i = 0, \dots, d-1$ . Explicitly, let  $c_i$  be defined by

$$\hat{\Sigma}_i^*(\omega_{i+1}) = c_i\omega_i,$$

where  $\omega_i$  denotes the invariant differential of  $E_i$ . Then

$$c = \prod_{0 \leq i < n} c_i.$$

Since  $V$  is separable, by [Sil92],II.4.2,  $c$  will be non-zero modulo  $p$  and we conclude

$$\mathrm{Tr}(F) \equiv \prod_{0 \leq i < n} c_i \pmod{q}.$$

In Section 3.3 we computed the equations for  $\mathcal{E}_i$  and  $\mathcal{E}_{i+1}$ . Let us now consider the following diagram:

$$\begin{array}{ccc} \mathcal{E}_i & \xrightarrow{\hat{\Sigma}_i} & \mathcal{E}_{i+1} \\ & \searrow \nu_i & \nearrow \lambda_i \\ & \mathcal{E}_i / \ker(\hat{\Sigma}_i) & \\ \pi \downarrow & & \downarrow \pi \\ E_i & \xrightarrow{\hat{\sigma}_i} & E_{i+1} \end{array}$$

Given the kernel  $\ker(\hat{\Sigma}_i)$  of the lifted dual isogeny between  $\mathcal{E}_i$  and  $\mathcal{E}_{i+1}$ , we can use Vélú's formulae [Vél71] to compute an equation for the curve  $\mathcal{E}_i / \ker(\hat{\Sigma}_i)$  and the isogeny  $\nu_i$ .

**Theorem 3.6.** *The explicit formula for the isogenies  $\nu_i \in \mathrm{Isog}(\mathcal{E}_i, \mathcal{E}_i / \ker(\hat{\Sigma}_i))$  is the following:*

$$\nu_i(X, Y) = \left( X + \sum_{P \in \ker(\hat{\Sigma}_i) \setminus \mathcal{O}} X_{(X,Y)+P} - X_P, \sum_{P \in \ker(\hat{\Sigma}_i)} Y_{(X,Y)+P} \right),$$

where the plus sign in  $(X, Y) + P$  is the addition on the elliptic curve.

By way of construction,  $\nu_i$  and  $\hat{\Sigma}_i$  are both separable. Since by 2.25(b) also  $\ker(\nu_i) = \ker(\hat{\Sigma}_i)$ , there exists an isogeny

$$\lambda_i: \mathcal{E}_i / \ker(\hat{\Sigma}_i) \rightarrow \mathcal{E}_{i-1}$$

which makes the diagram commutative, by 2.25(a).

By lemma 2.25,

$$\deg(\hat{\Sigma}_i) = \# \ker(\hat{\Sigma}_i) = \# \ker(\nu_i) = \deg(\nu_i).$$

Now, since

$$\deg(\nu_i) = \deg(\hat{\Sigma}_i) \quad \text{and} \quad \deg(\hat{\Sigma}_i) = \deg(\nu_i) \deg(\lambda_i),$$

it follows that  $\deg(\lambda_i) = 1$ , hence the isogeny  $\lambda_i$  is an isomorphism.

Now we need to compute

$$\hat{\Sigma}_i^*(\omega_{i+1}) = \nu_i^*(\lambda_i^*(\omega_{i+1})).$$

Let  $\omega_{i,k}$  denote the invariant differential on the curve  $\mathcal{E}_i/\ker(\hat{\Sigma}_i)$ . Due to Vélú's construction,  $\nu_i$  acts trivially on  $\omega$ , that is

$$\nu_i^*(\omega_{i,k}) = \omega_i.$$

It is therefore sufficient to compute the action of  $\lambda_i$  on  $\omega_{i+1}$ .

**Proposition 3.7.** *The subgroup  $\ker(\hat{\Sigma}_i)$  of  $\mathcal{E}_i[p]$  is of order  $p$ .*

**Proof.** Consider the following diagram:

$$\begin{array}{ccccc} \mathcal{E}_i & \xrightarrow{\hat{\Sigma}_i} & \mathcal{E}_{i+1} & \xrightarrow{\hat{\Sigma}_i} & \mathcal{E}_i \\ \pi \downarrow & & \pi \downarrow & & \pi \downarrow \\ E_i & \xrightarrow{\hat{\sigma}_i} & E_{i+1} & \xrightarrow{\sigma_i} & E_i \end{array}$$

Since our lift is canonical,  $\text{End}(\mathcal{E}_i)$  is isomorphic to  $\text{End}(E_i)$ . Let  $\varphi$  denote the isomorphism between these two rings. Let  $\psi$  and  $\hat{\psi}$  be defined as in (2.35).

We need to show that  $\#\ker(\hat{\Sigma}_i) = p$ . As we saw in 2.25, this is equivalent to showing that  $\deg(\hat{\Sigma}_i) = p$ . Hence, we have to show that

$$\Sigma_i \circ \hat{\Sigma}_i = [p].$$

We know that

$$\sigma_i \circ \hat{\sigma}_i = [p],$$

which is equivalent to

$$\psi(\Sigma_i) \circ \hat{\psi}(\hat{\Sigma}_i) = [p].$$

The equality

$$\psi(\Sigma_i) \circ \hat{\psi}(\hat{\Sigma}_i) = \varphi(\Sigma_i \circ \hat{\Sigma}_i)$$

holds by (2.3). Then we have

$$\psi(\Sigma_i) \circ \hat{\psi}(\hat{\Sigma}_i) = \varphi(\Sigma_i \circ \hat{\Sigma}_i) = [p].$$

But since  $\varphi$  was an isomorphism,

$$\Sigma_i \circ \hat{\Sigma}_i = [p],$$

which was to be shown. □

Let  $H_i(x)$  be defined as

$$H_i(x) = \prod_{P \in S} (x - x_P),$$

where  $S$  is defined as in 2.2.5 (with  $m = p$ ), then  $H_i(x)$  divides the  $p$ -division polynomial  $\Psi_{p,i}(x)$  of  $\mathcal{E}_i$  by (2.38).

To find the correct factor of  $\Psi_{p,i}(x)$ , Satoh proves the following lemma.

**Lemma 3.8** (Satoh). *Let  $p \geq 3$ .*

- (i) *Then  $\ker(\hat{\Sigma}_i) = \mathcal{E}_i[p] \cap \mathcal{E}_i(\mathbb{Z}_q^{ur})$ , where  $\mathbb{Z}_q^{ur}$  is the valuation ring of the maximal unramified extension  $\mathbb{Q}_q^{ur}$  of  $\mathbb{Q}_q$ .*
- (ii) *The polynomial  $H_i(x)$  is square free.*

We will prove part (ii) of the lemma, the proof of (i) can be found in [Sat00], Corollary 3.3.

The lemma implies that  $H_i(x)$  is a monic polynomial of degree  $(p-1)/2$  which divides  $\Psi_{p,i}(x)$ .

**Proof.** Suppose there exist two points  $P$  and  $Q$  in  $S$  such that  $x_P \equiv x_Q \pmod{p}$ . Then we know that the reduction of their  $x$ -coordinates modulo  $p$  is equal. Therefore,  $\pi(P) = \pm\pi(Q)$ . Without loss of generality, suppose  $\pi(P) = \pi(Q)$ . Since  $\pi$  is a homomorphism, we have that

$$\pi(P - Q) = \mathcal{O}.$$

Hence,

$$P - Q \in \ker(\pi) \cap \ker(\hat{\Sigma}_i).$$

It remains to show, that  $\ker(\pi) \cap \ker(\hat{\Sigma}_i) = \{\mathcal{O}\}$ . Now, by Proposition 3.7 we know that  $\#\ker(\hat{\Sigma}_i) = p$ . Thus  $\#(\ker(\pi) \cap \ker(\hat{\Sigma}_i))$  is either 1 or  $p$ , because  $(\ker(\pi) \cap \ker(\hat{\Sigma}_i))$  is certainly a subgroup of  $\ker(\hat{\Sigma}_i)$ . But since  $\ker(\hat{\Sigma}_i) \subseteq \mathcal{E}_i(\mathbb{Z}_q^{ur})$ , it follows from the theory of the formal group of an elliptic curve ([Sil92], IV) that  $\ker(\hat{\Sigma}_i)$  contains only one element whose projection is  $\mathcal{O}$ . Therefore

$$(\ker(\pi) \cap \ker(\hat{\Sigma}_i)) = \{\mathcal{O}\}.$$

Hence,  $P - Q = \mathcal{O}$  and we conclude that  $P = Q$ . Thus,  $H_i(x) \pmod{p}$  is square free.  $\square$

**Definition 3.9.** An irreducible polynomial  $f$  over a field  $K$  of characteristic  $p$  is of *inseparable degree*  $p$  if there exists a separable irreducible polynomial  $g \in K[x]$  such that

$$f(x) = g(x^{p^e}) \quad \text{for some } e \geq 1 \in \mathbb{N}.$$

The inseparable degree of  $f$  is denoted by  $\deg_i(f)$ .

By construction,  $H_i(x)$  is a monic polynomial of degree  $(p-1)/2$  which divides  $\Psi_{p,i}(x)$  and  $H_i(x) \bmod p$  is square free. Since  $E_i$  is ordinary, Lemma (2.31) implies that  $\ker(\hat{\sigma}_i) = E_i[p]$  and  $\Psi_{p,i}(x) \bmod p$  has inseparable degree  $p$ .

**Lemma 3.10.** *The degree of  $(\Psi_{p,i}(x) \bmod p)$  is  $p(p-1)/2$ .*

**Proof.** We know that  $\deg_i(\Psi_{p,i}(x) \bmod p) = p$ , hence  $\deg(\Psi_{p,i}(x) \bmod p)$  has to be divisible by  $p$ . Now

$$\Psi_{p,i}(x) = px^{\frac{p^2-1}{2}} + \text{lower order terms.}$$

Therefore, the degree of  $\Psi_{p,i}(x) \bmod p$  is strictly smaller than  $\frac{p^2-1}{2} = \frac{(p+1)(p-1)}{2}$ . We need to find the biggest factor  $k$  such that

$$kp < \frac{p^2-1}{2}.$$

Since  $k$  has to be an integer and  $\frac{(p+1)(p-1)}{2} > \frac{p^2-1}{2}$ , we find that  $k = \frac{p-1}{2}$ . Hence,

$$\deg(\Psi_{p,i}(x) \bmod p) = \frac{p(p-1)}{2}.$$

□

Now  $\deg(H_i(x)) = \frac{p-1}{2}$  and  $\deg(\Psi_{p,i}(x) \bmod p) = p \cdot \frac{p-1}{2}$ . Hence it follows that

$$H_i(x)^p \equiv \Psi_{p,i}(x) \bmod p.$$

This implies that we cannot apply Hensel's lemma, since  $H_i(x) \bmod p$  and  $\Psi_{p,i}(x)/H_i(x) \bmod p$  are not coprime. To solve this problem, Satoh uses a modified Hensel lifting, which also has quadratic convergence.

**Lemma 3.11** (Satoh ([FGH00])). *Let  $p \geq 3$  be a prime and  $\Psi(x) \in \mathbb{Z}_q[x]$  a polynomial satisfying*

$$\Psi'(x) \equiv 0 \pmod{p} \quad \text{and} \quad \Psi'(x) \not\equiv 0 \pmod{p^2}.$$

*Let  $h(x) \in \mathbb{Z}_q[x]$  be a monic polynomial known to precision  $n$  such that*

1.  $h(x) \bmod p$  is square-free and coprime to  $\pi(\Psi'(x)/p)$ ,
2.  $h(x)$  divides  $\Psi(x)$  to precision  $n+1$ .

*Then the polynomial*

$$H(x) = h(x) + \left( \frac{\Psi(x)}{\Psi'(x)} h'(x) \bmod h(x) \right)$$

*has the following properties*

- $H(x) \equiv h(x) \pmod{p^n}$ ,

- $H(x)$  divides  $\Psi(x)$  to precision  $2n + 1$ .

A proof for this Lemma can be found in [Sat00].

To turn this lemma into an algorithm, it remains to initialize the iteration. For this, we need a way to construct the first approximation  $h(x)$  for the factor  $H(x)$  of  $\Psi_p(x)$ . To obtain this, we investigate the  $p$ -th division polynomial  $\Psi_{p,i}$  of  $\mathcal{E}_i$ . Note that  $\pi(\Psi_p(x, A_i, B_i))$  factors as  $cf(x)^p$  with a monic and square free factor  $f \in \mathbb{F}_q[x]$ , that is,  $\deg_i(\pi(\Psi_p(x, A_i, B_i))) = p$ , and  $c \in F_q^\times$ .

Actually,

$$f(x) = \sum_{k=0}^{\frac{(p-1)}{2}} \frac{u_k}{u_{\frac{p-1}{2}}} x^k,$$

where  $u_k$  is the coefficient of  $x^{pk}$  in the  $p$ -th division polynomial of  $\mathcal{E}_{i+1}$ ,  $\Psi_p(x, A_{i+1}, B_{i+1})$ .

Furthermore,  $f$  is coprime to  $\pi(\frac{1}{p}\Psi'_p(x, A_i, B_i))$ . Hence, we can use  $f$  as the polynomial  $h(x)$ .

The input values for the algorithm LIFTH ([FGH00]) are a desired precision  $n$ , an elliptic curve  $E_{i+1}$  over  $\mathbb{F}_q$  and an elliptic curve  $\mathcal{E}_i$  over  $\mathbb{Z}_q$ . It returns a monic factor  $H(x)$  of the  $p$ -th division polynomial of  $\mathcal{E}_i$ , representing the kernel of  $\hat{\Sigma}_i$ .

---

**Algorithm 3** LiftH [FGH00]

---

```

1: procedure LIFTH( $n, E_{i+1}, E_i$ )
2:    $\Psi(x) \leftarrow$  the  $p$ -th division polynomial of  $\mathcal{E}_i$ 
3:    $\psi(x) \leftarrow$  the  $p$ -th division polynomial of  $E_i$ 
   degree  $\frac{p-1}{2}$ 
4:    $h(x) \leftarrow 0$ 
5:   for  $k = 0$  to  $\frac{p-1}{2}$  do
6:      $h(x) \leftarrow h(x) + \frac{u_k}{u_{\frac{p-1}{2}}} x^k$ , with  $u_k$  as above
7:   end for
8:    $H(x) \leftarrow$  LIFTHBIS( $n, \Psi(x), h(x)$ )
9:   return  $H(x)$ 
10: end procedure

1: procedure LIFTHBIS( $n, \Psi(x), h(x)$ )
2:   if  $n = 1$  then
3:     return  $h(x)$ 
4:   end if
5:    $n' \leftarrow \lceil \frac{n-1}{2} \rceil$ 
6:    $H(x) \leftarrow$  LIFTHBIS( $n', \Psi(x), h(x)$ )
7:    $H(x) \leftarrow H(x) + \left( \frac{\Psi(x)}{\Psi'(x)} H'(x) \bmod H(x) \right)$ 
8:   return  $H(x)$ 
9: end procedure

```

---

 $\triangleright$  Note  $\deg(\Psi) = \frac{p^2-1}{2}$  $\triangleright$  Note  $\psi = h^p$  for some  $h(x)$  with

### 3.3.3 The Formal Group Approach

Originally, Satoh formulated the situation in terms of formal groups of elliptic curves. This is not much different from the method described above using the invariant differential. However, it might give some more insight into Satoh's original idea, so we will state it here for completeness, referring to [Sat00] for more details.

**Proposition 3.12** (Satoh). *Let  $\mathcal{E}$  be an elliptic curve over a field  $K$  and let  $f \in \text{End}(\mathcal{E})$  be of degree  $q$ . We denote its local parameter at  $\mathcal{O}$  by  $\tau$ . Assume that the reduction  $\pi(f)$  of  $f$  modulo  $p$  is separable and that  $f(\ker \pi) \subset \ker \pi$ . Let  $\hat{f}$  be the homomorphism induced from  $f$  over the formal group  $\hat{\mathcal{E}}$  of  $\mathcal{E}$ . Then,*

$$\text{Tr}(f) = c_1 + \frac{q}{c_1}, \quad (3.10)$$

where  $\hat{f} = \sum_{n=1}^{\infty} c_n \tau^n$ .

Satoh's proof is quite straightforward, in fact, Satoh reckoned that his observation was already known, although he could not find any evidence for it in the literature. For completeness, we will just state Satoh's proof here.

**Proof.**[[Sat00]] Since  $f$  satisfies the equation

$$f \circ f - \text{Tr}(f)f + q = \mathcal{O}$$

in  $\text{End}(\mathcal{E})$ , we have

$$(c_1^2 - \text{Tr}(f)c_1 + q)\tau + O(\tau^2) = \mathcal{O}$$

on  $\hat{\mathcal{E}}$ , and hence the coefficient of  $\tau$  must vanish, so that

$$c_1^2 + q = \text{Tr}(f)c_1.$$

On the other hand, separability of  $\pi(f)$  implies  $\pi(c_1) \neq 0$  and hence  $c_1 \neq 0$ . (Cf. the proof of case 2 of Silverman ([Sil92], Cor. IV.7.5).) Therefore, (3.10) holds.  $\square$

Now the Frobenius endomorphism  $\mathcal{F}$  is separable, so we cannot apply the proposition to it. However, since we are working with an ordinary elliptic curve, the Verschiebung  $\mathcal{V}$  is separable and thus we have

$$\text{Tr}(\mathcal{F}) = \text{Tr}(\mathcal{V}) = c + \frac{q}{c}$$

with  $\tilde{\mathcal{V}}(\tau) = c\tau + O(\tau^2)$ . From (3.4) we know that  $\mathcal{V}$  can be written as  $\mathcal{V} = \hat{\Sigma}_{d-1} \circ \hat{\Sigma}_{d-2} \circ \cdots \circ \hat{\Sigma}_0$  and hence  $c$  can be computed as the product of the leading coefficients of the morphism induced by  $\hat{\Sigma}_i$ . More precisely, let  $c_i$  be defined by

$$\tau_{i+1} \circ \hat{\Sigma}_i = c_i \tau_i + O(\tau_i^2),$$

with  $\tau_i$  the local parameter of  $\mathcal{E}_i$  at  $\mathcal{O}$ , then

$$c = \prod_{0 \leq i < d} c_i.$$

Since  $\mathcal{V}$  is separable,  $c$  is non-zero modulo  $p$  and we can conclude

$$\mathrm{Tr}(\mathcal{F}) \equiv \prod_{0 \leq i < d} c_i \pmod{q}.$$

*Remark 3.13.* Since all the commutative squares in (3.4) are conjugates of each other, we can also recover the trace of Frobenius as the norm of  $c_0$  from  $\mathbb{Q}_q$  to  $\mathbb{Q}_p$ , i.e.

$$\mathrm{Tr}(F) \equiv N_{\mathbb{Q}_q/\mathbb{Q}_p}(c_0) \pmod{q}.$$

**Definition 3.14.** If  $K$  is a field and  $L$  is a Galois extension of  $K$ , the *norm* of an element  $\alpha \in L$  is defined as the product of all conjugates  $g(\alpha)$  of  $\alpha$ , for  $g$  in the Galois group  $G$  of  $L/K$ .

Since  $N(\alpha)$  is immediately seen to be invariant under  $G$ , it follows that the norm lies in  $K$ .

From this property, we can conclude that  $\mathrm{Tr}(F) \in \mathbb{Q}_p$ , in fact, it lies in  $\mathbb{Z}_p$ , since all computations leading to it were carried out in the ring.

### 3.3.4 Computing the Trace from the Lifted Data

The remaining step is to compute the trace from the lifted data. Let  $E_i$  be given by the equation  $y^2 = x^3 + A_i x + B_i$ . The isogeny constructed in (3.6) was explicitly given by the equation

$$\nu_i(X, Y) = \left( X + \sum_{P \in \ker(\hat{\Sigma}_i) \setminus \mathcal{O}} X_{(X,Y)+P} - X_P, \sum_{P \in \ker(\hat{\Sigma}_i)} Y_{(X,Y)+P} \right),$$

where the plus sign in  $(X, Y) + P$  was the curve addition. Vélú's [Vél71] formula also gives the Weierstrass model of  $\mathcal{E}_i / \ker(\hat{\Sigma}_i)$ . Denote by  $s_k$  the coefficient of  $x^{\frac{p-1}{2}-k}$  in  $H_i(x)$ . Then

$$\mathcal{E}_i / \ker(\hat{\Sigma}_i) = y^2 + \alpha_i x + \beta_i$$

with

$$\begin{aligned} \alpha_i &:= A_i - 5 \cdot \sum_{P \in \ker(\hat{\Sigma}_i) \setminus \{\mathcal{O}\}} (3x_P^2 + A_i) \\ &= (6 - 5p)A_i - 30(s_1^2 - 2s_2), \\ \beta_i &:= B_i - 7 \cdot \sum_{P \in \ker(\hat{\Sigma}_i) \setminus \{\mathcal{O}\}} (5x_P^3 + 3A_i x_P + 2B_i) \\ &= (15 - 14p)B_i - 70(-s_1^3 + 3s_1 s_2 - 3s_3) + 42A_i s_1, \end{aligned}$$

where we set  $s_3 = 0$  in the case that  $\frac{p-1}{2} - k < 0$ .

Given the equation for  $\mathcal{E}_i / \ker(\hat{\Sigma}_i)$ , we can now compute the isomorphism  $\lambda_i$  to

$$\mathcal{E}_{i+1}: y^2 = x^3 + A_{i+1}x + B_{i+1}.$$

Since the only change of variables preserving the form of these equations is

$$\lambda_i: (x, y) \mapsto (g_i^2 x, g_i^3 y)$$

with

$$g_i^2 = \frac{\alpha_i B_{i+1}}{\beta_i A_{i+1}},$$

the coefficients  $c_i$  are completely determined. The action of  $\lambda_i$  on  $\omega_{i+1}$  is given by

$$\lambda_i^*(\omega_{i+1}) = g_i^{-1} \omega_{i,k}$$

and therefore we find the following formula for the  $c_i^2$ :

$$c_i^2 = \frac{\beta_i A_{i+1}}{\alpha_i B_{i+1}}.$$

It remains to compute  $c^2 = \prod_{i=0}^{n-1} c_i^2$  and to take the square root. This gives the trace of Frobenius up to the sign. As it is shown in the proof of [Sil92], Theorem V.4.1, we have

$$t \equiv \gamma \gamma^\sigma \dots \gamma^{\sigma^{n-1}} \pmod{p},$$

where  $\gamma$  is the coefficient of  $x^{p-1}$  in the polynomial  $(x^3 + ax + b)^{\frac{p-1}{2}}$ . Hence, we can determine the trace of  $E$  uniquely.

The input values for the algorithm COMPUTETRACEODDCHAR ([FGH00]) are an elliptic curve  $E_{i+1}$  over  $\mathbb{F}_{p^d}$ , with  $j(E) \notin \mathbb{F}_{p^2}$ , given by its equation  $y^2 = x^3 + ax + b$ . It returns the trace of the Frobenius of  $E$ .

---

**Algorithm 4** ComputeTraceOddChar [FGH00]

---

```

1: procedure COMPUTETRACEODDCHAR( $E: y^2 = x^3 + ax + b$ )
2:    $n \leftarrow \lceil \log_p 4 + \frac{d}{2} \rceil$ 
3:    $(j_i)_{0 \leq i < d}, (a_i)_{0 \leq i < d}, (b_i)_{0 \leq i < d} \leftarrow$  Conjugates of  $j(E)$ , of  $a$ , of  $b$ 
4:    $(J_i)_{0 \leq i < d} \leftarrow$  LIFTJINVARIANTS( $n, (j_i)$ )
5:    $N \leftarrow 1$ 
6:    $D \leftarrow 1$ 
7:   for  $i = 0$  to  $d - 1$  do
8:      $(\gamma_i) \leftarrow \frac{j_i}{1728 - j_i}$ 
9:      $A \leftarrow 3\gamma$ 
10:     $B \leftarrow 2\gamma$ 
11:     $H \leftarrow$  LIFTH( $n, a_{i+1}, b_{i+1}, A, B$ )
12:     $s_1, s_2, s_3 \leftarrow$  coefficients of  $x^{\frac{p-1}{2}-1}, x^{\frac{p-1}{2}-2}$  and  $x^{\frac{p-1}{2}-3}$  in  $H(x)$ 
13:     $\alpha \leftarrow (6 - 5p)A - 30(s_1^2 - 2s_2)$ 
14:     $\beta \leftarrow (15 - 14p)B - 70(-s_1^3 + 3s_1s_2 - 3s_3) + 42As_1$ 
15:     $N \leftarrow N\beta A$ 
16:     $D \leftarrow D\alpha B$ 
17:   end for
18:    $c \leftarrow \sqrt{N/D}$ 
19:    $h_E \leftarrow$  Hasse invariant of  $E$ 
20:   if  $c \not\equiv h_E \pmod{p}$  then  $c \leftarrow -c$ 
21:   end if
22:   Reduce  $c$  to an integer in  $0 \dots p^n$ 
23:   if  $c > 2\sqrt{q}$  then
24:      $c \leftarrow c - p^n$ 
25:   end if
26:   return  $c$ 
27: end procedure

```

---

### 3.4 Characteristic $p = 2$

In this Section, we describe the extension of Satoh's algorithm to characteristic two by Mireille Fouquet, Pierrick Gaudry and Robert Harley [FGH00].

#### 3.4.1 Lifting each Curve

In characteristic two, every non-supersingular elliptic curve  $E$  (i.e.  $j(E) \neq 0$ ) can be written in the form

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

Recall from chapter 2 that  $a_6$  is a unit in  $\mathbb{F}_2^*$  and  $a_2 \in \{0, \gamma\}$ . For a given value of  $a_6$ , the two curves, with  $a_2 = 0$  and  $a_2 = \gamma$ , respectively, are *twists* of each other.

**Twist:** To an elliptic curve  $E$ , a non-isomorphic curve  $E'$  whose trace is the negation of  $\text{Tr}(E)$  is called a *twist* of  $E$ .

Hence, either  $E$  or its twist are of the form

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

Since from computing the trace we can trivially obtain its negation, we can confine ourselves to this form in order to simplify the calculations.

To obtain the coefficient  $A$  in the equation  $y^2 + xy = x^3 + A$  of the lifted curve  $\mathcal{E}$ , we perform a Newton iteration to lift the coefficient  $a$  of our initial curve  $E$ . We are already given the  $j$ -invariant  $J$  of the lifted curve. Since  $J = \frac{1}{\Delta}$ , where  $\Delta = -A - 432A^2$  is the discriminant of the curve, the polynomial on which we base our Newton iteration is

$$f(x) = 1 + J(x + 432x^2).$$

Finding a root  $A$  of precision  $n$  of  $f(x)$ , we have found the coefficient  $A$  of our lifted curve  $\mathcal{E}$  to precision  $n$ .

In this case  $f'(x) = J(1 + 864x)$ . One way to proceed would be to apply a simple Newton iteration, since we know that  $\pi(J) \neq 0$ . This way, we could go from precision  $n$  to  $2n$  at each step. In fact, there is a slightly better algorithm than the generic method. With this special algorithm, we can instead go to precision  $2n + 4$ .

Let  $x$  be an approximate root at precision  $n$ , with error term  $f(x) = O(2^n)$ . Let

$$y = x - \frac{f(x)}{f'(x)}$$

be the improved root after one step. Then the new error term can be written explicitly as

$$\begin{aligned} f(y) &= 1 + J(y + 432y^2) \\ &= 432J \frac{f(x)^2}{f'(x)^2} \end{aligned}$$

Since we know that  $2 \nmid f'(x)$  and  $2^4 \parallel 432$ , we can conclude that  $f(x) = O(2^{2n+4})$ .

Since the precision increases not only by a factor but also by the summand  $+4$ , it follows that an initial approximate root is not actually needed to start the iteration. For efficiency we might as well use  $-\frac{1}{j} \pmod{16}$ .

The input values for the algorithm LIFTA ([FGH00]) are a desired precision  $n$  and the  $j$ -invariant  $J$  of a curve  $\mathcal{E}$  in  $\mathbb{Z}_q$ , with precision  $n$ . It returns the coefficient  $A$  of the lifted curve  $\mathcal{E}$  in  $\mathbb{Z}_q$  with precision  $n$ .

---

**Algorithm 5** LiftA [FGH00]

---

```

1: procedure LIFTA( $n, J$ )
2:   if  $n \leq 4$  then
3:     return  $-\frac{1}{J}$ 
4:   end if
5:    $n' \leftarrow \lceil \frac{n-4}{2} \rceil$ 
6:    $A \leftarrow$  LIFTA( $n', J$ )
7:    $A \leftarrow A - \frac{1+J(A+432A^2)}{J(1+864A)}$ 
8:   return  $A$ 
9: end procedure

```

---

### 3.4.2 Lifting the 2-Torsion

This algorithm for lifting the 2-torsion is again based on a Newton iteration.

Let  $E_i$  be given by the equation  $y^2 + xy = x^3 + a_i$  with  $a_i \in F_q$  and  $\mathcal{E}_i$  by  $y^2 + xy = x^3 + A_i$  with  $A_i \in \mathbb{Z}_q$ . Since  $\hat{\Sigma}_i$  is separable and of degree 2,

$$\ker(\hat{\Sigma}) = \{\mathcal{O}, P_i\}.$$

To compute the lifted non-trivial torsion point  $P_i$ , Fouquet, Gaudry and Harley use the fact that

$$\ker(\hat{\Sigma}) \subset \mathcal{E}_i[2].$$

This implies that the  $X$ -coordinate of  $P_i$  satisfies the 2-division polynomial

$$\psi_2(x) = 4x^3 + x^2 + 4A_i.$$

A root  $X \in \mathbb{Z}_q$  of this polynomial is necessarily 0 modulo 2. Hence, we compute  $Z_i = \frac{X_{P_i}}{2}$  as a zero of the modified division polynomial  $f(z) = 8z^3 + Z^2 + A_i$ .

We have to overcome the problem that there are two candidate roots in  $\mathbb{Z}_q$  of which only one can correspond to the non-trivial point in the kernel of  $\hat{\Sigma}_i$ . Actually, it suffices to initialise the Newton iteration with the correct square root of  $-A \pmod{8}$ .

The following description by Fouquet, Gaudry and Harley shows how to do this in a deterministic way. This solves the problem Satoh encountered in his paper while avoiding the probabilistic polynomial factorization modulo 8 that he suggests.

**Choosing the correct root** Using Vélú's formulae again to compute the  $j$ -invariant  $J_{i+1}$  of  $\mathcal{E}_i$  explicitly in terms of  $A_i$  and the as-yet-unknown  $Z_i$  yields

$$J_{i+1} = J(\mathcal{E}_{i+1}) = \frac{-1}{Z_i^2 - Z_i + A_i} \pmod{4}.$$

Since  $Z_i$  is a root of  $f(z)$ , we have  $Z_i^2 + A_i \equiv 0 \pmod{4}$  (in fact even modulo 8, but this is not needed here) and thus

$$Z_i \equiv \frac{1}{J_{i+1}} \pmod{4}.$$

This determines  $Z_i$  uniquely and provides a sufficiently precise initial root  $Z_{i,0}$  for the Newton iteration. Indeed,

$$f'(x) = 2(12x^2 + x) \quad \text{and} \quad 12Z_{i,0}^2 + Z_{i,0} \equiv \frac{1}{J_{i+1}} \pmod{4}$$

and  $\frac{1}{J_{i+1}}$  is non-zero modulo 2. Hence, following the notation from 2.4.3, we have  $k = 1$  and we need

$$f(Z_{i,0}) \equiv 0 \pmod{2^{2k+1}} = 2^3$$

to start the iterations. Now

$$\begin{aligned} f(Z_{i,0}) &\equiv Z_{i,0}^2 + A_i \pmod{8} \\ &\equiv \frac{1}{J_{i+1}^2} + A_i \quad (\text{the value of } Z_{i,0} \pmod{4} \text{ determines } Z_{i,0}^2 \pmod{8}) \\ &\equiv \frac{1}{J_{i+1}^2} - \frac{1}{J_i}. \quad (\text{from lifting of } A_i \text{ in section 3.4.1}) \end{aligned}$$

Hence it remains to show that  $J_{i+1}^2 \equiv J_i \pmod{8}$ . By definition,  $J_i = \Sigma(J_{i+1})$ , so the desired result certainly holds modulo 2. A short calculation shows that the Kronecker relation

$$\Phi_2(X, Y) \equiv (X^2 - Y)(X - Y^2)$$

actually holds modulo 16 and not just modulo 2:

$$\begin{aligned} \Phi_2(X, Y) &\equiv X^3 + Y^3 - X^2Y^2 + 15XY \pmod{16} \\ &\equiv X^3 + Y^3 - X^2Y^2 - XY \pmod{16} \\ &\equiv (X^2 - Y)(X - Y^2). \end{aligned}$$

We therefore have

$$\Phi_2(\omega(j_i), \omega(j_{i+1})) \equiv 0 \pmod{16},$$

where  $\omega$  denotes the Teichmüller lift as defined in (2.10). Considering the semi-Witt decomposition (2.11) of  $J_{i+1}$ , we see that the terms  $x_1$ ,  $x_2$  and  $x_3$  are zero. Thus  $J_i \equiv J_{i+1}^2 \pmod{16}$  and hence certainly modulo 8 as required.

**Lifting the root.** To finally compute the correct root  $Z$ , we use the procedure NEWTONITERATIONS with  $k = 1$  and initial root  $\frac{1}{J_{i+1}} \pmod{4}$ .

As we saw in 2.4.3, the precision increases at each step from  $n$  to  $2n - k$ , so in this case, to  $2n - 1$ . Let  $x$  be the approximate value at precision  $n$  with error term  $f(x) = \mathcal{O}(2^{n+1})$ . Let  $y = x - \frac{f(x)}{f'(x)}$  be the improved root after one step. Then the new error term is

$$\begin{aligned} f(y) &= 8y^3 + y^2 + A \\ &= \frac{f(x)^2}{f'(x)^2}(24x + 1) - 8\frac{f(x)^3}{f'(x)^3}. \end{aligned}$$

From  $k = 1$  we have that  $2 \parallel f'(x)$  and therefore

$$f(y) = \mathcal{O}(2^{\min(2n, 3n+3)}) = \mathcal{O}(2^{2n})$$

as expected.

The input values for the algorithm LIFTZ ([FGH00]) are a desired (integer) precision  $n$ , the  $j$ -invariant  $J_{i+1}$  to precision 2 and the coefficient  $A$  to precision  $n + 1$ . It returns the  $Z$  of the lifted curve  $\mathcal{E}$  to precision  $n$ .

---

**Algorithm 6** LiftZ [FGH00]

---

```

1: procedure LIFTZ( $n, J_{i+1}, A$ )
2:   if  $n \leq 2$  then
3:     return  $-\frac{1}{J_{i+1}}$ 
4:   end if
5:    $n' \leftarrow \lceil \frac{n+1}{2} \rceil$ 
6:    $Z \leftarrow \text{LIFTZ}(n', J_{i+1}, A)$ 
7:    $Z \leftarrow Z - \frac{8Z^3 + Z^2 + A}{2(12Z^2 + Z)}$ 
8:   return  $Z$ 
9: end procedure

```

---

### 3.4.3 Computing the Trace from the Lifted Data

Recall the equation of  $\mathcal{E}_i$ ,

$$y^2 + xy = x^3 + A_i,$$

and that of  $\mathcal{E}_{i+1}$ ,

$$y^2 + xy = x^3 + A_{i+1}.$$

Consider again the following diagram:

$$\begin{array}{ccc}
 \mathcal{E}_i & \xrightarrow{\hat{\Sigma}_i} & \mathcal{E}_{i+1} \\
 \pi \downarrow & \searrow \nu_i & \nearrow \lambda_i \\
 & \mathcal{E}_i / \ker(\hat{\Sigma}_i) & \\
 \downarrow & & \downarrow \pi \\
 E_i & \xrightarrow{\hat{\sigma}_i} & E_{i+1}
 \end{array}$$

In the case of characteristic two, the kernel of  $\hat{\Sigma}_i$  is a subgroup of order 2 of the 2-torsion of the form

$$\ker(\hat{\Sigma}_i) = \{\mathcal{O}, Q_i\}.$$

Let  $Q_i = (X_{Q_i}, Y_{Q_i})$  be the non-trivial point of this subgroup.

Once we have computed  $X_{Q_i}$ , we can recover  $Q_i$ , since  $Q_i = -Q_i$  implies that

$$y_{Q_i} = -\frac{X_{Q_i}}{2}.$$

The curve  $\mathcal{E}_i / \ker(\hat{\Sigma}_i)$  given by Vélú's formulae [Vél71] has the equation

$$y^2 + xy = x^3 + \mathcal{A}_4x + \mathcal{A}_6,$$

where

$$\mathcal{A}_4 = -5t \quad \text{and} \quad \mathcal{A}_6 = A_i - t - 7w,$$

with

$$t = 3X_{Q_i}^2 - Y_{Q_i} \quad \text{and} \quad w = X_{Q_i}t.$$

Here again, the knowledge of the equations for  $\mathcal{E}_i / \ker(\hat{\Sigma}_i)$  and  $\mathcal{E}_{i+1}$  allows us to compute the isomorphism  $\lambda_i$  and then the coefficient  $c_i^2$ . The isogeny  $\nu_i$  is explicitly given by the equation

$$\nu_i(x, y) = (x + X_{(x,y)+X_{Q_i}} - X_{Q_i}, X_{(x,y)+Q_i}).$$

The isomorphism  $\lambda_i$  has the general form

$$(x, y) \mapsto (u_i^2x + r_i, u_i^3y + u_i^2s_ix + t_i), \quad (u_i, r_i, s_i, t_i) \in \mathbb{Q}_q^* \times \mathbb{Q}_q^3,$$

compare [Sil92], III.1. From the formal group expansion of the isomorphism, we learn that  $c_i^2 = u_i^{-2}$  [Skj03]. Solving the equations satisfied by  $(u_i, r_i, s_i, t_i)$  given in [Sil92], Table 1.2, finally leads to

$$c_i^2 = -\frac{864\beta_i - 72\alpha_i + 1}{(48\alpha_i - 1)(1 + 864A_{i+1})}. \quad (3.11)$$

Alternatively, one can compute the  $c_i^2$  using the equations of  $\mathcal{E}_{i+1}$  and  $\mathcal{E}_i / \ker(\hat{\Sigma}_i)$ . For this, a translation of the axes is needed, which does not affect the first coefficient in the

formal group expansion of the isomorphism (compare [Skj03]). After this translation, the equations are as follows:

$$\begin{aligned}\mathcal{E}_{i+1} &: y^2 = x^3 - \frac{1}{48}x + \frac{1}{864} + A_{i+1} \\ \mathcal{E}_i / \ker(\hat{\Sigma}_i) &: y^2 = x^3 + (\mathcal{A}_4 - \frac{1}{48})x + \frac{1}{864} + \mathcal{A}_6 - \frac{\mathcal{A}_4}{12}.\end{aligned}$$

In this case, the isomorphism  $\lambda_i$  is again of the form  $(x, y) \mapsto (g_i^2 x, g_i^3 y)$ , where

$$g_i^2 = \frac{\alpha_i B_{i+1}}{\beta_i A_{i+1}},$$

hence

$$g_i^2 = \frac{-\frac{1}{48}}{\frac{1}{864} + A_{i+1}} \cdot \frac{\frac{1}{864} + \mathcal{A}_6 - \frac{\mathcal{A}_4}{12}}{\mathcal{A}_4 - \frac{1}{48}},$$

which simplifies to

$$g_i^2 = \frac{72\mathcal{A}_4 - 1 - 864\mathcal{A}_6}{(48\mathcal{A}_4 - 1)(1 + 864A_{i+1})}.$$

Replacing  $\mathcal{A}_4$  and  $\mathcal{A}_6$  with their expressions in terms of  $A_i$  and  $X_{Q_i}$  leads to the following term:

$$g_i^2 = \frac{-18144X_{Q_i}^3 - 4536X_{Q_i}^2 - 252X_{Q_i} + 1 + 864A_i}{(1 + 120X_{Q_i} + 720X_{Q_i}^2)(1 + 864A_{i+1})}.$$

A further simplification is made by reducing this formula modulo the minimal polynomial of  $X_{Q_i}$ , which is  $4X_{Q_i}^3 + X_{Q_i}^2 + 4A_i$ . We get

$$g_i^2 = \frac{1 - 252x_i + 19008A_i}{(1 + 120(x_i + 6x_i^2))(1 + 864A_{i+1})},$$

and hence

$$c_i^2 = \frac{1 - 252x_i + 19008A_i}{(1 + 120(x_i + 6x_i^2))(1 + 864A_{i+1})},$$

as above.

Note that we lifted  $Z_i = X_{Q_i}/2$  instead of  $X_{Q_i}$  and that we need one extra bit of precision for the 2-adic square root. Remembering this, we get the following algorithm.

The input values for the algorithm COMPUTETRACECHAR2 ([FGH00]) are an elliptic curve  $E$  over  $\mathbb{F}_{2^a}$ , with  $j(E) \notin \mathbb{F}_4$ , given by its equation  $y^2 + xy = x^3 + a$ . It returns the trace of the Frobenius of the curve.

**Algorithm 7** ComputeTraceChar2 [FGH00]

---

```

1: procedure COMPUTETRACECHAR2( $E$ )
2:    $j_0 \leftarrow j(E)$ 
3:   for  $i = d - 1$  to  $1$  do
4:      $j_i \leftarrow j_{i+1}^2$ 
5:   end for
6:    $n \leftarrow \lceil \frac{d}{2} \rceil + 1$ 
7:    $(J_i)_{0 \leq i < d} \leftarrow \text{LIFTJINVARIANTS}(n, (j_i))$ 
8:    $N \leftarrow 1$ 
9:    $D \leftarrow 1$ 
10:  for  $i = 0$  to  $d - 1$  do
11:     $A \leftarrow \text{LIFTA}(n, J_i)$ 
12:     $Z \leftarrow \text{LIFTZ}(n - 1, J_{i+1}, A)$ 
13:     $A \leftarrow 864A$ 
14:     $N \leftarrow N(1 - 504Z + 22A)$ 
15:     $D \leftarrow D(1 + 240(Z + 12Z^2))(1 + A)$ 
16:  end for
17:   $c \leftarrow \sqrt{N/D}$ 
18:  if  $c \not\equiv 1 \pmod{4}$  then
19:     $c \leftarrow -c$ 
20:  end if
21:  Reduce  $c$  to an integer in  $0 \dots 2^{n+1}$ .
22:  if  $c > 2\sqrt{q}$  then
23:     $c \leftarrow c - 2^{n+1}$ 
24:  end if
25:  return  $c$ 
26: end procedure

```

---

▷  $N$  has precision  $n + 2$   
 ▷  $D$  has precision  $n + 2$

Recall from (3.13) that in step 17, both the numerator  $N$  and denominator  $D$  are in  $\mathbb{Z}_2$  and hence  $c$  is, too. Further, the 2-adic square root can be found via a Newton iteration for the inverse square root, i.e. via a Newton iteration on

$$s(X) = c^2 X^2 - 1.$$

We have the following properties:

$$s(1/c) = 0 \quad \text{and} \quad s'(1/c) \equiv 0 \pmod{2}.$$

Furthermore, we know that  $E$  has a point of order 4, hence

$$s'(1/c) \not\equiv 0 \pmod{4} \quad \text{and} \quad c \equiv 1 \pmod{4}.$$

The vanishing of  $s'(1/c)$  modulo 2 means that we lose exactly one bit of precision in the computation of the square root and therefore we need to compute  $c^2$  modulo  $2^{\lceil \frac{n+6}{2} \rceil}$ .

---

**Algorithm 8** TwoAdicSqrt [FGH00]

---

```
1: procedure TWOADICSQRT( $A, m$ )
2:    $X \leftarrow 1$ 
3:    $i \leftarrow -1$ 
4:   while  $(X^2 - A) \bmod 2^i = 0$  do
5:      $i \leftarrow i + 1$ 
6:   end while
7:   while  $i < m$  do
8:      $Z \leftarrow \frac{(X^2 - A)}{2^i} \bmod 2$ 
9:      $X \leftarrow X + Z \cdot 2^{i-1}$ 
10:     $i \leftarrow i + 1$ 
11:  end while
12:  return  $X$ 
13: end procedure
```

---

The algorithm TWOADICSQRT ([FGH00]) computes the 2-adic square root  $S$  of an element  $A$  with precision  $m$ , such that  $S^2 \equiv A \pmod{2^m}$  and  $S \equiv 1 \pmod{4}$ .

### 3.5 An Example

We present an example of a curve  $E$  over the finite field  $\mathbb{F}_{2^5} \cong \mathbb{F}[t]/(t^5 + t^2 + 1)$ . The curve is given by the equation

$$E: y^2 + xy = x^3 + t^3.$$

We are working to precision  $m = \lceil \frac{5}{2} \rceil + 1 = 4$ .

The cycle of lifted  $j$ -invariants  $J_i$  is as follows:

$$\begin{aligned} J_0 &= 12 + 9t + 7t^2 + 10t^3 + 13t^4 \\ J_1 &= 5 + 2t + 9t^2 + 9t^3 + 13t^4 \\ J_2 &= 12 + 12t + 5t^2 + 10t^3 + 9t^4 \\ J_3 &= 14 + 11t + 3t^2 + 8t^3 + 4t^4 \\ J_4 &= 3 + 14t + 8t^2 + 11t^3 + 9t^4 \end{aligned}$$

From that, the coefficients  $A_i$  of the lifted curves  $\mathcal{E}_i$  are computed using a Newton iteration on the function

$$f(A_i) = 1 + J_i(A_i + 432A_i),$$

as explained in Section 3.4.1.

$$\begin{aligned} A_0 &= -8 - 6t - 12t^2 - 7t^3 - 2t^4 \\ A_1 &= -5 - t - 2t^3 - 5t^4 \\ A_2 &= -10 - 11t - 15t^2 + 5t^3 - 7t^4 \\ A_3 &= -8 - 15t - 9t^2 + 7t^3 - 10t^4 \\ A_4 &= -4 - 15t - 12t^2 + 11t^3 - 8t^4 \end{aligned}$$

Finally, the lifted 2-torsion points  $Z_i$  are computed from the  $A_i$  via Newton iteration applied to the function

$$f(Z_i) = 8Z^3 + Z^2 + A.$$

Computed to precision 4, the resulting sequence is:

$$\begin{aligned} Z_0 &= 5 + 9t + 8t^2 + 6t^3 + 5t^4 \\ Z_1 &= 6 + 15t + 15t^2 + 5t^3 + 3t^4 \\ Z_2 &= 11t + 13t^2 + 11t^3 + 6t^4 \\ Z_3 &= 4 + 3t + 7t^3 \\ Z_4 &= 8 + 2t + 12t^2 + 11t^3 + 2t^4 \end{aligned}$$

The square of the trace can now be computed modulo  $2^7$  by the formula

$$c^2 = \frac{\prod_i (1 - 504Z_i + 19008A_i)}{\prod_i (1 + 240(Z_i + 12Z_i^2))(1 + 864A_i)} \equiv \frac{57}{49} \equiv 969.$$

The two square roots modulo  $2^6$  are

$$c \equiv \pm 3,$$

and we choose the one that is congruent 1 modulo 4. Hence

$$c = -3,$$

and the number of points on  $E$  is

$$\#E = 36.$$

### 3.6 Extensions of Satoh's Algorithm

Apart from the extension by Fouquet, Gaudry and Harley [FGH00] which was described in Section 3.4, some further extensions were published in order to enhance both time and memory efficiency.

Independently from Fouquet, Gaudry and Harley, Berit Skjernaas [Skj03] extended Satoh's algorithm to characteristic  $p = 2$ . Based on this, a memory efficient algorithm was developed by Frederik Vercauteren, Bart Preneel and Joos Vandewalle [VPV01] with memory usage reduced from  $\mathcal{O}(d^3)$  to  $\mathcal{O}(d^2)$  (see Section 4.5).

**SST** In 2001, Satoh, Skjernaas and Yuichiro Taguchi [SST03] developed an improved algorithm which reduced the runtime to  $\mathcal{O}(d^{2+1/2}(\log d)(\log \log d))$  using precomputed traces for a basis. It uses two new ideas. The first is the actual computation of the Frobenius of  $\mathbb{Z}_q$ . While all former algorithms avoided computations of the Frobenius since this was assumed to be very slow, Satoh, Skjernaas and Taguchi used an alternative representation of  $\mathbb{Z}_q$  that allows fast Frobenius computations.

Let  $E$  be an ordinary elliptic curve over  $\mathbb{F}_{p^d}$  with  $j(E) \notin \mathbb{F}_{p^2}$ . Satoh, Skjernaas and Taguchi obtain an algorithm to compute the  $j$ -invariant  $j(\mathcal{E}) \bmod p^{\mathcal{O}(d)}$  of the canonical lift  $\mathcal{E}$  of  $E$  in time  $\mathcal{O}(d^{2\mu+1/(1+\mu)})$  (where  $\mu$  is a constant such that the time for multiplying to  $m$ -bit integers is  $\mathcal{O}(m^\mu)$ ), not including precomputations, and using  $\mathcal{O}(d^2)$  memory.

The second idea is a fast norm computation algorithm which runs in  $\mathcal{O}(d^{2\mu+0.5})$  bit-operations and uses  $\mathcal{O}(d^2)$  memory.

**Kim et al.** In [KPC<sup>+</sup>02], Kim, Park, Cheon, Kim and Hahn proposed to use finite fields with a Gaussian Normal Basis (GNB) of small type as an improvement of the SST algorithm. Such a basis can be lifted trivially to  $\mathbb{Z}_q$  and allows fast computation of arbitrary iterates of Frobenius. Combined with a norm computation algorithm suggested by Kedlaya, they obtained a point counting algorithm for elliptic curves over  $\mathbb{F}_{p^d}$  with Gaussian Normal Basis which requires  $\mathcal{O}(d^{2\mu+1/(1+\mu)})$  bit-operations and  $\mathcal{O}(d^2)$  space, without precomputation.

**Mestre's AGM Algorithm** In a letter to Gaudry and Harley, Mestre [Mes00] described a very elegant method based on the Arithmetic Geometric Mean (AGM) to count the number of points on ordinary curves of genus 1 and 2 over  $\mathbb{F}_{2^d}$ . The resulting algorithm runs in  $\mathcal{O}(d^2 + 1)$  bit-operations and requires  $\mathcal{O}(d^2)$  memory.

The AGM algorithm was reformulated in the language of schemes and extended to characteristic 3 by Carls [Car04].

In the following, we will briefly describe the main ideas of Mestre's AGM method. We will only consider the case of ordinary elliptic curves over finite fields of characteristic 2.

The objects computed by this algorithm are the same as those computed in Satoh's algorithm, but the computation is done in a different way. The canonical lift is approximated using the so-called *AGM sequence*.

**Definition 3.15.** Let  $a, b \in \mathbb{R}$  be two non-negative real numbers. We define a sequence of numbers by setting

$$a_{n+1} = \frac{a_n + b_n}{2} \text{ and } b_{n+1} = \sqrt{a_n b_n},$$

for  $n \geq 0$ , where  $a_0 = a$  and  $b_0 = b$ . The sequence  $(a_n, b_n)_{n \in \mathbb{N}}$  is called *AGM sequence*.

The limits of the sequences  $a_n$  and  $b_n$  exist and

$$\lim_{n \rightarrow \infty} a_n = \lim_{n \rightarrow \infty} b_n.$$

The common limit is called the *arithmetic geometric mean* of  $a$  and  $b$ .

We can now use the AGM sequence to work with elliptic curves. Starting with an elliptic curve  $E: y^2 = x(x-a)(x-b)$ , we define a sequence  $E_n$  of elliptic curves by setting

$$E_n: y^2 = x(x - a_n^2)(x - b_n^2),$$

where  $a_n$  and  $b_n$  are elements in the AGM sequence of  $a$  and  $b$ .

Now we are given a rational map  $E_n \rightarrow E_{n+1}$  by

$$(x, y) \mapsto \left( \frac{(x + a_n b_n)^2}{4x}, \frac{y(a_n b_n - x)(a_n b_n + x)}{8x^2} \right).$$

This map is a 2-isogeny, that is to say, an surjective morphism of groups whose kernel is a subgroup of order 2 of  $E_n$ .

Mestre considered the sequence of curves over the 2-adic field  $\mathbb{Q}_2$ . As a result, the sequence of  $j$ -invariants

$$j_n = 2^8 \frac{(a_n^4 - a_n^2 b_n^2 + b_n^4)^3}{a_n^4 b_n^4 (a_n^2 - b_n^2)^2}$$

of the curves  $E_n$  converges if  $E_0$  has ordinary reduction, that is, the reduction of  $E_0$  is an elliptic curve with a rational point of order 2.

The sequence  $E_n$  has a higher dimensional analogon, a sequence of 2-isogenous abelian varieties over  $\mathbb{Q}_2$  which build the foundation of Mestre's algorithm for ordinary hyperelliptic curves over finite fields of characteristic 2.

**MSST** At first sight, combining the AGM and the SST algorithm seems difficult. However, in [Gau02] Pierrick Gaudry devised a simplified modular polynomial  $\mathcal{Y}_2(x, y)$  based on the AGM iteration and then uses the SST algorithm to find a solution of  $\mathcal{Y}_2(x, \Sigma(x)) = 0$ . The time and space complexities are the same as those of the SST algorithm. However, Gaudry's algorithm is much faster since the polynomial  $\mathcal{Y}_2(x, y)$  has lower degree than  $\Phi_2(x, y)$ .

**Harley** In an email to the NMBRTHRY list [Har02] in December 2002, Harley announced an algorithm to count the number of points on an ordinary elliptic curve over  $\mathbb{F}_{p^d}$ , which runs in  $\mathcal{O}(d^{2\mu} \log d)$  bit-operations and uses  $\mathcal{O}(d^2)$  space for fixed  $p$ . A complete description including the missing routines can be found in [Ver03].

**Lercier-Lubicz** In [LL03], Reynald Lercier and David Lubicz presented a  $p$ -adic point counting algorithm without precomputation which runs in  $\mathcal{O}(d^{2+\varepsilon})$  and uses  $\mathcal{O}(d^2)$  space for finite fields with Gaussian Normal Basis.

## 4 Implementation

In this chapter, we will explain the idea and structure of our implementation. We will discuss the representation of the mathematical structures  $\mathbb{Z}_2$  and  $\mathbb{Z}_{2^n}$  as well as the algorithms used for the inner arithmetic. We are going to relate the implementation to the pseudo code shown in the previous chapter.

Finally, we will discuss how our program is integrated in the project FlexiECPProvider<sup>2</sup> developed by the institute of theoretical computer science at the TU Darmstadt.

### 4.1 Representation of $\mathbb{Z}_2$ and $\mathbb{Z}_{2^n}$

Recall from Section 2.1 the rings  $\mathbb{Z}_2$  and  $\mathbb{Z}_{2^n}$ . Every element  $a \in \mathbb{Z}_2$  can be represented by an infinite sequence

$$(a_1, a_2, \dots, a_n, \dots),$$

where  $a_k \in \mathbb{Z}/2^k\mathbb{Z}$  for  $k \geq 1$ . This sequence has the property, that

$$a_{k-1} \equiv a_k \pmod{2^k}, \quad \text{for all } k \geq 1.$$

This means that once we know the sequence element  $a_m$ , we can compute all  $a_k$  for  $k \leq m$  by

$$a_k \equiv a_{k+1} \pmod{2^{k+1}} \equiv a_{k+2} \pmod{2^{k+2}} \equiv \dots \equiv a_m \pmod{2^m}.$$

Since an implementation can only store a finite number of sequence elements, we can only represent  $a$  by a finite sequence

$$(a_1, a_2, \dots, a_m).$$

In this case we say that we know  $a$  to *precision*  $m$ , meaning we know  $a_m \equiv a \pmod{2^m}$ .

Hence, an element  $a \in \mathbb{Z}_2$  is represented by its precision  $m$  and its  $m$ -th sequence element  $a_m$ . Due to the above property, the remaining sequence elements are stored implicitly.

Arithmetic operations in  $\mathbb{Z}_2$  are performed componentwise. Naturally, we simply perform the operation on  $a_m$  and retrieve the correct result. Assuming the elements  $a$  and  $b$  are represented in the same precision  $m$ , the following rules of arithmetic apply:

$$\begin{aligned} a \pm b &\equiv (a_m \pm b_m \pmod{2^m}) \pmod{2^m}, \\ a \cdot b &\equiv (a_m \cdot b_m \pmod{2^m}) \pmod{2^m}. \end{aligned}$$

Since  $\mathbb{Z}_2$  is a ring, division is more complicated. An element  $a \in \mathbb{Z}_2$  is invertible if and only if  $a \equiv 1 \pmod{2}$ . In this case, division is performed componentwise, too.

Every element  $a \in \mathbb{Z}_2$  can be written as

$$a = 2^k \cdot u,$$

---

<sup>2</sup><http://www.flexiprovider.de>

where  $k \geq 0$  and  $u$  is invertible, that is,  $u \equiv 1 \pmod{2}$ . The integer  $k$  is called the *valuation* of  $a$ .

In our implementation, elements of the ring  $\mathbb{Z}_2$  are almost exclusively used as coefficients of polynomials representing elements from  $\mathbb{Z}_{2^n}$ .

Elements from  $\mathbb{Z}_{2^n}$  are represented as polynomials in  $\mathbb{Z}_2[t]/(f(t))$ , where  $f(t)$  is a monic irreducible trinomial or pentanomial of degree  $n$  over  $\mathbb{Z}_2$ . Most of the arithmetic is based on polynomial arithmetic. Special cases are the division and calculation of inverses. These will be discussed later in this chapter.

The ring structure is implemented in the class `Z2nRing.java` where the defining polynomial  $f(t)$  is stored.

## 4.2 The Subsidiary Classes

Before going into the details of the main classes `SatohFGH` and `Skjernaa`, we are going to focus on the subsidiary classes `Z2nRing` and `Z2nRingElement` first.

**The class `Z2nRing`** This class stores the information necessary to characterize the ring  $\mathbb{Z}_{2^n}$ . The field  $\mathbb{F}_{2^n}$ , its degree  $n$  and the ring polynomial  $f(t)$ . The ring class also contains the constants 0, 1 and 2, since these are used frequently.

Moreover, the class stores an indicator whether the ring polynomial is a trinomial or a pentanomial. It also contains the array `globalPrecMask` which plays an important role in the precision handling, which is described in 4.4.

**The class `Z2nRingElement`** This class does most of the arithmetic work. A `Z2nRingElement` contains the `Z2nRing` in which it lies, the polynomial representing this element, as described in 4.1, and the degree of this polynomial.

From outside the class, such an element can be created from a `java.math.BigInteger[] s`, a `de.flexiprovider.ec.arithmetic.gf.Bitstring b` or a single `BigInteger x`. The former two constructors create the polynomial directly from the given sequence of elements before reducing it modulo the ring polynomial. The latter constructor creates a `Z2nRingElement` with a constant polynomial `x`.

All the external constructors need a parameter `prec` to make sure the element is represented to the right precision. From within the class, two private constructors can be used which do not need the precision parameter. These are used for internal arithmetic operations which cut the polynomials to the right precision before constructing a new element.

Most of the arithmetic operations are performed in the usual polynomial arithmetic. Exceptions are division and inversion, since we are working with polynomials over a ring.

The most time consuming operation is the multiplication of two elements. We implemented the Karatsuba multiplication algorithm for polynomials, which increased the performance significantly. More on the program's performance can be found in section 4.4.

### 4.3 The Main Classes

**The class SatohFGH** This class implements the Satoh-FGH algorithms as described in section 3.4. The program was implemented, as far as possible, directly from the pseudo code by Fouquet, Gaudry and Harley in [FGH00].

**The class SkjernaVPV** This class implements the algorithm developed by Berit Skjerna [Skj03] and extended by Frederik Vercauteren, Bart Preneel and Joos Vandewalle [VPV01]. We implemented both algorithms to compare their respective runtimes when using the same underlying arithmetic. The main differences between the two main classes are described in 4.5.

### 4.4 Encountered Difficulties

During the process of implementing the Satoh-FGH algorithm, several problems had to be faced and solved.

Since the implementation was based on the pseudo code from [FGH00], we basically followed the instructions. This was however not always as simple as it sounds. The major difficulties were the precision handling, the implementation of some arithmetic operations in  $\mathbb{Z}_{2^n}$  and, last but not least, the performance.

Let us start by describing the arithmetic difficulties occurring during inversion, division and reduction modulo the ring defining polynomial.

**Inversion** The inversion of an invertible element  $a \in \mathbb{Z}_{2^n}$  is done in a fairly straightforward way. We perform a Newton iteration on the function  $x \mapsto \frac{1}{x} - a$ .

For an efficient implementation of this (and every other) Newton iteration, it was important to work with the lowest precision possible at each stage of the algorithm. Starting with precision one, we double the precision at each iteration until the required precision is reached. This way, we run a predetermined number of iterations to receive the correct result.

Another approach would have been to calculate with the required precision. In general, this reduces the number of iterations. Anyway, the expense of calculating with a higher precision weighs too heavy, so that this approach was not applicable here.

**Division** Closely related to the concept of inverting elements is the division by elements from  $\mathbb{Z}_{2^n}$ . The naïve way of dividing one ring element by another would be the multiplication by its inverse. This however can and will lead to many exceptions. As discussed earlier, an element of  $\mathbb{Z}_{2^n}$  is invertible if and only if at least one of the coefficients of its representing polynomial is not divisible by 2. Hence an element like

$$x = 68 + 42t + 38t^2 + 4t^3 + 94t^4$$

is certainly not invertible. However, the quotient

$$\frac{y}{x} = \frac{12 + 44t + 36t^2 + 8t^3 + 50t^4}{68 + 42t + 38t^2 + 4t^3 + 94t^4}$$

can be computed, since we can simply cancel out a 2 in both numerator and denominator. This leads to the quotient

$$\frac{6 + 22t + 18t^2 + 4t^3 + 25t^4}{34 + 21t + 19t^2 + 2t^3 + 47t^4}$$

where the inverse of the denominator can be computed by the method INVERSE.

Finding the highest power of 2 by which both numerator and denominator are divisible is done using the `java.math.BigInteger` function `getLowestSetBit`. It determines the position of the lowest bit of both elements. Naturally, if the denominator's lowest bit is higher than the numerator's, the quotient cannot be computed.

**Reduction modulo the ring polynomial** The representation of  $\mathbb{Z}_{2^n}$  elements by polynomials requires reducing these representing polynomials by the defining ring polynomial. Since the ring polynomial is always chosen to be a trinomial or, if that is not possible, a pentanomial, reduction is done more efficiently than simply computing the remainder of a division by the ring polynomial.

The ring polynomial is simply the lifted field polynomial of the underlying field  $\mathbb{F}_{2^n}$ . Hence, its coefficients are either 0 or 1. During the construction of our ring, we determine whether it is a trinomial or a pentanomial and store the positions of the non-zero coefficients in an array. This array is then used for reduction. Suppose the ring we are dealing with is represented as

$$\mathbb{Z}_{2^n} \cong \mathbb{Z}_2[t]/(f(t)),$$

where  $f(t) = t^n + t^k + 1$ . Now assume the element  $a$  is, prior to reduction modulo  $f(t)$ , given by a polynomial  $a(t) = \sum_{i=0}^s c_i t^i$  of degree  $s \geq n$  with leading coefficient  $c_s$ . The naive approach would be to multiply  $f(t)$  by  $c_s t^{s-n}$  and to subtract the resulting polynomial from  $a(t)$ . Repeating this process yields the desired polynomial of degree  $r < n$  representing  $a$ .

To avoid the multiplication, we utilize that multiplication by a monomial  $t^{s-n}$  equals shifting all the coefficients  $(s-n)$  positions to the left. Since we already know the positions of the 1s in  $f(t)$ , we don't actually have to perform the multiplication. We can immediately write down the product

$$c_s t^{s-n} f(t) = c_s t^{n+(s-n)} + c_s t^{k+(s-n)} + c_s t^{s-n} = c_s t^s + c_s t^{s-(n-k)} + c_s t^{s-n}.$$

Subtracting this result from  $a(t)$  is simulated by setting the new  $c_s$  to zero and subtracting the old  $c_s$  from  $c_{s-(n-k)}$  and from  $c_{s-n}$ . After that,  $s$  is replaced by the highest exponent  $r$  such that  $c_r \neq 0$ . This process is repeated until  $r < n$ .

The input values for the algorithm REDUCETRINOMIAL are a nonzero polynomial  $a(t) = \sum_{i=0}^s c_i t^i$  of degree  $s$ , integers  $n$  and  $k$  marking the positions of the 1s in  $f(t)$ . It returns a polynomial  $b(t)$  of degree  $r < n$  with  $a(t) \equiv b(t) \pmod{f(t)}$ .

---

**Algorithm 9** reduceTrinomial

---

```

1: procedure REDUCETRINOMIAL( $a(t), n, k$ )
2:   for  $d = s$  down to  $n$  do
3:     lead  $\leftarrow c_s$ 
4:     if lead  $\neq 0$  then
5:        $c_s \leftarrow 0$ 
6:        $c_{s-(n-k)} \leftarrow c_{s-(n-k)} - \text{lead}$ 
7:        $c_{s-n} \leftarrow c_{s-n} - \text{lead}$ 
8:     end if
9:   end for
10:  for  $d$  down to 0 do
11:    if  $c_d \neq 0$  then
12:       $r = d$  break
13:    end if
14:  end for
15:   $b(t) \leftarrow \sum_{i=0}^r c_i t^i$ 
16:  return  $b(t)$ 
17: end procedure

```

---

An analogous procedure is used for the reduction modulo a pentanomial.

**Precision handling** This was one of the crucial points of the implementation. The first intention was to equip each `Z2nRingElement` with a field `precision`. It turned out to be faster and easier to follow a different approach. Each arithmetic operation is called with a precision parameter which determines the precision of the output. Also, each external constructor is called with a precision parameter as described in 4.2. Hence, at every point during the algorithm, each element is represented to the currently needed precision.

In comparison to the former approach, this simplifies for example the addition of two elements with different precisions.

The second part of the precision handling consisted of developing a method to set an element to a certain precision. Instead of computing the specific power of 2 each time, we create an array `globalPrecMask` in which all the needed precisions are stored. So the process of setting an element to precision  $m$  works as follows: we check the array `globalPrecMask` at entry  $m$  for a value. If this is not `null`, the entry will be  $2^m - 1$ . If there is no entry at position  $m$  yet, we set it to  $2^m - 1$ . Now we can simply use the `BigInteger` operation `and` to set our element to precision  $m$ .

The array `globalPrecMask` is stored in the class `Z2nRing`.

**Program performance** Performance was an important factor although some restrictions had to be accepted. A plain JAVA implementation without underlying arithmetic

libraries can certainly not compete with C++ based programs like Magma<sup>3</sup> or GAP<sup>4</sup>.

For example, the arithmetic structures are based on the type `java.math.BigInteger`. This is an immutable class and therefore takes up a lot of time while initialising new objects.

The class `Z2nRingElement.java` is mainly immutable, too. Only operation like addition, subtraction and negation are implemented as element-changing operations. This is mainly due to the immutability of `BigInteger`, since the more complex operations cannot work on the element itself, so there is no benefit from making the more complex operations mutable.

To enhance the performance of our program, we replaced most of the recursions appearing in the pseudo code by faster algorithms.

## 4.5 Satoh-FGH vs Skjerna-VPV

The implementation contains two main classes. The Skjerna-VPV algorithm runs, with our underlying arithmetic, faster than the implementation of Satoh-FGH. Since performance was an important factor, we included it in our implementation. We will now shortly describe the differences. A more detailed description of the Skjerna-VPV algorithm can be found in [VPV01].

We can structure the algorithms in two main parts: the first part includes lifting the  $j$ -invariants, the second part deals with the lift of the curve and the torsion subgroup and the explicit formula for the trace. We will consider these parts separately.

**Lifting the  $j$ -invariants** This part of the Skjerna-VPV algorithm was developed by Vercauteren, Preneel and Vandewalle. They work around the multivariate Newton iteration described in Section 3.2 by using a univariate Newton iteration on the polynomial  $\Phi_2(X, J_{i+1})$ , where  $\Phi_2$  is the second modular polynomial (cp. Section 2.3).

The algorithm `LiftPrevJInv` computes coefficients  $A, B, C \in \mathbb{Z}_{2^n} \text{ mod } 2^N$  such that

$$\Phi_2(X, J_{i+1}) \equiv X^3 + AX^2 + BX + C \pmod{2^N},$$

and then calls the recursive algorithm `LiftPrevJInvRec` which performs the Newton iteration on the cubic polynomial  $X^3 + AX^2 + BX + C$ .

With every call of the algorithm `LiftPrevJInv` one bit of precision is gained. So suppose we would like to compute  $J_0 \equiv j(\mathcal{E}_0) \pmod{2^N}$ , then it suffices to start with  $j(E_{N-1}) \equiv j(\mathcal{E}_{N-1}) \pmod{2}$  and iterate this algorithm  $N - 1$  times. This leads then to the algorithm `LiftFirstJInv`, which computes the lift of the first  $j$ -invariant as a basis for the former two algorithms.

---

<sup>3</sup><http://magma.maths.usyd.edu.au/magma/>

<sup>4</sup><http://www.gap-system.org/>

**Lifting the curve and the torsion subgroup** While the Satoh-FGH algorithm explicitly computes the lifted curve coefficients  $A_i$ , Skjernaa gives an expression for the square of the trace just in terms of the lifted  $j$ -invariants and the non-trivial torsion points  $Z_i$ . The formula for  $Z_i$  given by

$$Z_i = \frac{J_{i+1}^2 + 195120J_{i+1} + 4095J_i + 660960000}{8(J_{i+1}^2 + J_{i+1}(563760 - 512J_i) + 372735J_i + 8981280000)}.$$

Given the lifted  $J_i$  and  $Z_i$ , Skjernaa-VPV computes the square of the trace using the formula

$$c^2 = \frac{\prod_i J_i - (504 + 12096Z_i)T_i}{\prod_i J_i + 240T_i},$$

where  $T_i = (12Z_i^2 + Z_i)(J_i - 1728) - 36$ .

Given the lifted  $A_i$  and  $Z_i$ , Satoh-FGH computes the square of the trace via the formula

$$c^2 = \frac{\prod_i (1 - 504Z_i + 19008A_i)}{\prod_i (1 + 240(Z_i + 12Z_i^2))(1 + 864A_i)}.$$

Not computing the  $Z_i$  externally but directly from the  $J_i$  accelerates the second part of the algorithm significantly.

## 4.6 Integration in the FlexiECPProvider

The FlexiProvider is a powerful toolkit for the Java Cryptography Architecture (JCA/JCE). It provides cryptographic modules that can be plugged into every application that is built on top of the JCA.

One section of the FlexiECPProvider is concerned with the generation of public keys. For this purpose, elliptic curve parameters are generated in a way that ensures the security of a public key. The parameters are chosen so that the curve resists all known attacks on the *Elliptic Curve Discrete Logarithm Problem* (ECDLP).

Our algorithm was built into the FlexiECPProvider class `GF2nParameterGenerator`. This class initialises a set of curve parameters consisting of the size of the field  $\mathbb{F}_q$ , the equation for the curve  $E(\mathbb{F}_q): y^2 + xy = x^3 + a$ , a point  $P$  on  $E$  of large prime order  $r$ , the cofactor  $h = \#E(\mathbb{F}_q)/r$  and the field polynomial  $f(t)$ .

The parameters need to comply with the following requirements. In the following, let  $N$  denote the number of points on  $E$  and  $q$  the size of the field  $\mathbb{F}_q$ .

1. Verify that  $N$  is divisible by a large prime  $r$  so that  $h := N/r \in \{1, 2, 3, 4\}$ .
2. Verify that  $r \nmid q^k - 1$  for  $1 \leq k \leq 20$ .
3. Verify that  $r \neq q$ .
4. Choose a point  $P'$  on  $E$  such that  $P := hP' \neq \infty$ .

Then  $P$  has order  $n$  and the curve is secure against all known attacks on the ECDLP.

## 4.7 Numerical Results

In this section, we will present running times for elliptic curves over fields  $\mathbb{F}_{2^n}$ , where  $n$  varies between 63 and 300. The tables contain a column for the total runtime and a column labeled *Multiplication*. This allows the reader to see how much of the total time was used to multiply `Z2nRingElements`. To simplify the comparison of both algorithms, the percentage of time for multiplication is given in the third column.

$n$	Total runtime	Multiplication	Percentage
63	11.64s	9.56s	82.12%
65	29.45s	26.46s	89.84%
126	92.22s	82.34s	89.28%
130	278.48s	254.50s	91.39%
160	386.26s	354.19s	91.69%
200	619.18s	564.09s	91.10%
230	780.90s	704.58s	90.23%
256	1182.61s	1076.13s	91.00%
260	2124.55s	2024.96s	95.31%

Table 1: Runtimes using `SatohFGH.java`

$n$	Total runtime	Multiplication	Percentage
63	6.90s	5.81s	84.29%
65	16.22s	14.69s	90.56%
126	56.01s	49.86s	89.01%
130	116.85s	107.74s	92.20%
160	161.32s	147.55s	91.46%
200	265.37s	244.35s	92.08%
230	367.12s	337.12s	91.83%
256	488.61s	449.45s	91.98%
260	884.30s	834.16s	94.33%

Table 2: Runtimes using `SkjernaaVPV.java`

Obviously, the highest potential to increase the performance lies in the multiplication algorithm for `Z2nRingElements` which is the most frequently called operation. Hence, one could reach better results by modifying the arithmetic to reduce the number of multiplications. A more extensive approach would be to implement one of the extensions of Satoh's algorithm.

**Parameter Generation** We tested the class `ECGF2nParameterGenerator` for the field  $\mathbb{F}_{2^{101}}$ . We found the secure curve

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

where (in hexadecimal notation)

$$a = 00000007 \text{ a98a5689 } 50070980 \text{ 9fd57d52}$$

and

$$b = 0000001b \text{ b4cdc41c } 757142db \text{ a155d061}.$$

The seed which was used for the SHA-256 hash-function while computing  $b$  is

$$\begin{aligned} S = & \text{ 794dba52 9a41c1ce 47a65437 a6ee10b8 f18a5807 e7be9fe4} \\ & \text{ eb304483 48f908ff 2ec945a2 58c01df9 10b0354a 6fa1e206} \\ & \text{ da6fcb06 fbe81d41 1b0eaae1 7d929148.} \end{aligned}$$

The number of points on  $E$  is

$$\#E = 2535301200456457404178519631926,$$

the cofactor is 2.

A point of order

$$\#E/2 = 1267650600228228702089259815963 = 0000000f \text{ ffffffff fffd83e4 } 83c8941b$$

on  $E$  is

$$\begin{aligned} P = & (00000004 \text{ 17b2d256 b51bbd56 4a44a7a2,} \\ & 0000001b \text{ 22b25ceb 3d7adad9 278cb6ee).} \end{aligned}$$

The number of iterations needed to find a secure curve varied from 1 to 183 (for 11 to 107 bits) during the test runs. Without further detailed research we cannot make a comment on how long it takes to generate the parameters *on average*. The number of iterations usually increases with the bitsize, although a large bitsize never rules out the possibility of a quick parameter generation.

## 5 Earlier Point Counting Algorithms

Elliptic curves and the number of their rational points have been used in cryptography for a long time. Over the years, several mathematicians developed algorithms for point counting on elliptic curves.

In general, one distinguishes  $l$ -adic and  $p$ -adic methods. In  $l$ -adic methods, one computes the trace that leads to the number of points modulo several primes  $l$  and uses the Chinese Remainder Theorem to obtain the actual trace. In  $p$ -adic methods however, the trace is computed modulo powers of one prime  $p$ , the characteristic of the ground field.

The first polynomial time (with respect to  $\log_q$ )  $l$ -adic algorithm was obtained by René Schoof [Sch95]. Based on his result, Elkies and Atkin developed faster algorithms and their combination led to the so-called SEA (Schoof-Elkies-Atkin) algorithm [Elk98]. Further contributions to the SEA algorithm were made by J.M. Couveignes [CM94], M. Fouquet and F. Morain [FM02]. J. Pila succeeded to generalize Schoof's algorithm to abelian varieties of higher dimension [Pil90]. Both algorithms are restricted to elliptic curves, whereas other algorithms were developed to count points on arbitrary algebraic varieties.

### 5.1 Schoof

We will briefly describe the basic ideas and the main steps of Schoof's algorithm.

Let  $E$  be an elliptic curve defined over the finite field  $\mathbb{F}_q$  of characteristic  $p \geq 5$ . On this curve the Frobenius isogeny  $\phi: E \rightarrow E$  is given by

$$(x, y) \mapsto (x^q, y^q) \text{ and } \mathcal{O} \mapsto \mathcal{O}.$$

Denoting by  $\hat{\phi}$  the dual isogeny of  $\phi$ , the trace of the Frobenius isogeny is defined as the number  $t$  such that  $\phi + \hat{\phi} = [t]$ . The trace also satisfies the equation

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

Also, for any point  $P \in E$  we have that

$$\mathcal{O} = \phi(P) - [t](P) + \hat{\phi}(P).$$

Applying  $\phi$  again and using the fact that  $\phi \circ \hat{\phi} = [q]$  gives us

$$\mathcal{O} = \phi^2(P) - [t]\phi(P) + [q](P). \tag{5.1}$$

By Hasse's Theorem we have a bound on  $t$ , namely

$$|t| \leq 2\sqrt{q}.$$

The set of  $l$ -torsion points on  $E$ , where  $l$  is any integer, is

$$E[l] = \{P \in E: l \cdot P = \mathcal{O}\}.$$

Now (5.1) holds for any point on  $E$ , hence naturally also for any point in  $E[l]$ . Let  $l$  be a prime. Denoting by  $q_l$  and  $t_l$  the numbers in  $\{0, \dots, l-1\}$  such that  $q_l \equiv q \pmod{l}$  and  $t_l \equiv t \pmod{l}$ , we see that the following holds for a point  $P \in E[l]$ :

$$\phi^2(P) - [t_l]\phi(P) + [q_l](P) = \mathcal{O}.$$

Since  $|t| \leq 2\sqrt{q}$ , if we can find  $t \pmod{l}$  for so many primes  $l$  that their product is greater than  $4\sqrt{q}$ , we can recover  $t$  using the Chinese Remainder Theorem. If we can find a number  $\tau \in \{0, \dots, l-1\}$  such that

$$\phi^2(P) + [q_l](P) = [\tau]\phi(P), \tag{5.2}$$

we have found  $t \pmod{l}$ . If  $l$  is small, it will be feasible to find  $\tau$  by trial and error.

The crucial point is to find a way to see if  $l$ -torsion points exists such that (5.2) holds.

**Time Complexity** Assuming a naive multiplication algorithm, Schoof's algorithm runs in  $\mathcal{O}(\log^8 q)$  bit operations.

- The number of primes  $l$  needed for fixed characteristic  $p$  lies in  $\mathcal{O}(\log q)$ ,
- every prime  $l$  has a magnitude in  $\mathcal{O}(\log q)$ ,
- the calculation of  $t \pmod{l}$  for each prime  $l$  includes the computation of greatest common divisors of polynomials of magnitude in  $\mathcal{O}(\log^3 q)$ .

Taking all this together, we get a complexity in  $\mathcal{O}(\log^7 q)$  for each prime  $l$  and thus the complexity of the algorithm lies in  $\mathcal{O}(\log^8 q)$ .

## 5.2 Schoof-Elkies-Atkin

While Schoof's algorithm requires the computation of greatest common divisors of division polynomials (of degree  $\mathcal{O}(l^2)$ ), Elkies and Atkin construct a factor of degree  $\mathcal{O}(l)$  of the division polynomial and work with it instead. Finding such a factor is done by factoring the modular polynomial to find eigenspaces of the Frobenius endomorphism  $F$  restricted to  $E[l]$ .

N. Elkies improved its running time and his algorithm supposed to run in  $\mathcal{O}(\log^6 q)$  bit operations for most elliptic curves. Its running time depends on how many primes split in the quotient field of the given elliptic curve. For the rest of primes, there is an algorithm by A.O.L Atkin. The combination of these algorithms is called the SEA algorithm.

## 5.3 Comparison

When counting the number of points on a given curve, one has to decide between the different algorithms. The most popular algorithms are SEA and Satoh-FGH. Two cases

have to be dealt with differently: the case of large characteristic and the case of small characteristic.

While Satoh's algorithm is very efficient in small characteristic, it has a bad dependency on the characteristic of the base field. When  $p$  is large, it is not efficient at all. This behaviour is due to the use of the modular polynomial  $\Phi_p$  for the lifting of the curves. This polynomial has  $\mathcal{O}(p^2)$  coefficients that have to be known at least modulo  $p^{(d/2)+\mathcal{O}(1)}$ . Thus a complexity which is exponential in  $p$  appears to be unavoidable. The SEA algorithm, however, is polynomial-time independently of  $p$ . So curves over fields with large characteristic should be dealt with using SEA.

In small characteristic, however, Satoh's algorithm is efficient. In particular in characteristic 2, Satoh-FGH is definitely faster than SEA in practice.

Running time comparisons for SEA and Satoh-FGH can be found in [FGH01].

*Remark 5.1.* In some cases, the SEA and Satoh-FGH algorithm can be combined to speed-up the point-counting. A particular application is the point counting on curves over fields of a size such that the maximum precision required in Satoh-FGH is a little more than a multiple of the machine word-size.

### Overview of runtimes and memory usage

	Time	Dependency on $p$	Memory
Satoh	$\mathcal{O}(\log^3 q)$	badly dependent	$\mathcal{O}(\log^3 q)$
Schoof	$\mathcal{O}(\log^8 q)$	independent	$\mathcal{O}(\log^2 q)$
SEA	$\mathcal{O}(\log^6 q)$	independent	$\mathcal{O}(\log^2 q)$

## 5.4 Conclusion

The Satoh-FGH algorithm was clearly proven to be the best choice whenever one wants to compute the number of points on a random elliptic curve defined over a field of characteristic two. Anyway, one should not abandon the SEA algorithm tool quickly. Over fields of larger characteristic, it is the only practical method available.

Moreover, Lercier [Ler97] developed an early-abort strategy which is used while searching for cryptographically *good* curves with the SEA algorithm. Even in small characteristic, this strategy in combination with the SEA algorithm is valuable when looking for a curve for cryptographic use.

Fouquet, Gaudry and Harley proposed a way of combining the early-abort strategy with the Satoh-FGH algorithm [FGH01]. This yields an efficient way of computing secure curves and the use of precomputed curves in cryptography is no longer necessary.

## A On Time Complexity

Complexity theory as a section of theoretical computer science deals with the complexity of problems in algorithmic form on different mathematically defined formal computer models, and with the quality of the solution algorithms. The analysis of complexity thereby refers to the usage of resources by the algorithms, the most common resources are time (how many steps of a certain size it takes to solve a problem) and space (how much memory it takes).

The difference between complexity theory and computability theory is that the latter deals with whether a problem can be solved at all, regardless of the resources required, whereas the former analyses given solvable problems to narrow down the amount of *efficiently* solvable problems.

### A.1 Aspects of Complexity Theory

**Classification of Complexity in Theoretical Computer Sciences** Along with computability theory and formal language theory, complexity theory is one of the three major sections of theoretical computer science. Its essential research objective is the classification of problems with respect to the resources needed to solve them. Thereby, the confine of efficiently solvable problems plays an important role. Complexity theory therefore narrows down those problems for which the other disciplines of computer science should search for efficient solutions at all.

**Problems from the View of Complexity Theory** The central topic of complexity theory are problems. In this context, a *problem* describes an abstract question: Actually, one wants to achieve conclusions for arbitrary instances of this question.

Solutions of instances of certain abstract questions are only of limited value, they are only applicable to similar problems. Hence, complexity theory is concerned with statements that are independent from concrete instances.

**Problem Sizes** Once a problem is defined in a mathematical formal way, one wants to make statements about the behaviour of an algorithm when computing the problem. Potentially, there are many different aspects of the problem to be considered. However, there usually there exists one quantity which influences the behaviour of the algorithm decisively with respect to the usage of resources. This quantity is referred to as the *problem magnitude* or *problem size*.

The next step is to analyse the algorithm's behaviour with respect to different problem magnitudes. The interest of complexity theory lies in the following question: What is the additional expense relative to increasing problem magnitudes? Is the increase of resources linear, polynomial, exponential or even more than that?

**Best, Worst and Average Case** Not only while comparing different problem sizes, within the problem there are also different behaviour patterns of algorithms to observe.

One cannot expect an algorithm to work equally well or fast when given different conditions, even when the magnitude stays the same. Since the amount of instances for a given problem is practically infinite, there is a way of grouping them roughly into three groups: *best case*, *worst case* and *average case*.

**Upper and Lower Bounds for Problems** The observation of best, worst and average case corresponds to an arbitrary but fixed problem size. Even though this consideration is of great interest in practice, complexity theory demands even more abstractness. Due to permanent advances in technology, problem sizes that are considered big or practically relevant can turn irrelevant pretty fast. It is therefore justifiable to analyse the behaviour of algorithms with respect to a problem independently from concrete problem magnitudes. For this purpose, one considers the behaviour of the algorithms for increasing, potentially infinitely large, problem sizes. In this case, we speak of the *asymptotical* behaviour of the algorithm.

In this analysis of the asymptotic consumption of resources, upper and lower bounds play a central role. It is of interest, how much resources are needed at least or at most for the decision of a problem. Complexity theory is primarily interested in the lower bounds: One wants to show what minimal amount of resources is needed for a certain type of problem to eliminate the possibility that there is an algorithm that consumes less resources.

Contrary to the verification of upper bounds, which can usually be obtained by analysing concrete algorithms, the research of lower bounds needs to consider algorithms which have not been developed yet, hence the set of all, also unknown, algorithms deciding a certain problem. This demands a fundamentally different and more abstract approach than the analysis of already known algorithms. Hence, this is considered a difficult to access section of computer science.

## A.2 Big $\mathcal{O}$ Notation

The Big  $\mathcal{O}$  Notation is a frequently used way to compare complexities. It is defined as follows.

**Definition A.1.** Suppose  $f(x)$  and  $g(x)$  are two functions defined on some subset of the real numbers. We say that

$$f(x) \in \mathcal{O}(g(x)) \quad \text{as } x \rightarrow \infty$$

if and only if there exist  $x_0 \in \mathbb{R}$  such that

$$|f(x)| \leq M|g(x)| \quad \text{for } x \geq x_0 \quad \text{and some } M \in \mathbb{R}.$$

The notation can also be used to describe the behavior of  $f$  near some real number  $a$ . We say that

$$f(x) \in \mathcal{O}(g(x)) \quad \text{as } x \rightarrow a$$

if and only if there exist  $\delta > 0$  and  $M \in \mathbb{R}$  such that

$$|f(x)| \leq M|g(x)| \quad \text{for } |x - a| < \delta.$$

If  $g(x)$  is non-zero for values of  $x$  sufficiently close to  $a$ , both of these definitions can be unified using the limit superior:

$$f(x) \in \mathcal{O}(g(x)) \quad \text{as } x \rightarrow a$$

if and only if

$$\limsup_{x \rightarrow a} \left| \frac{f(x)}{g(x)} \right| < \infty.$$

In mathematics, both asymptotic behaviors near  $\infty$  and near  $a$  are considered. In computational complexity theory, only asymptotics near  $\infty$  are used; furthermore, only positive functions are considered, so the absolute value bars may be left out.

Beside the big  $\mathcal{O}$ , there are some other notations that are also frequently used in complexity theory. The respective symbols are  $\circ, \Omega, \omega, \Theta$ .

During the analysis of magnitudes of the amount of resources, complexity theory makes ample use of the  $\mathcal{O}$  notation. Thereby, one leaves constants or factors out of the consideration. This might seem surprising at first, since in practice the bisection of an amount of needed resources is often already of big importance.

This position is justifiable by a certain technique called linear acceleration or *Speedup-Theorem*. For this, we confine ourselves to the case of time complexity. There are analogous methods for space and other resources.

Simplified, the Speedup-Theorem indicates that, for every Turing machine solving a problem in  $\mathcal{O}(f)$  time, one can build a new Turing machine solving the problem in  $\mathcal{O}(f')$  time, where  $f' = \varepsilon f$ ,  $\varepsilon > 0$ . Another way of saying that would be that every Turing machine solving a certain problem can be accelerated by a linear factor. The price for this acceleration consists in significantly enlarged magnitude of working alphabet and set of states, which ultimately means hardware.

This acceleration is achieved independently of the problem size. Hence, there is no point in taking linear factors into account. Neglecting linear factors, which is phrased in the  $\mathcal{O}$  notation, is therefore not only of practical use but also helps to avoid falsifications in complexity theoretical considerations.

## B The JAVA Code

In this section, we provide the code for the parameter generating class `GF2nParameterGenerator`, the algorithm classes `SatohFGH` and `SkjernaVPV` and the arithmetic classes `Z2nRing` and `Z2nRingElement`.

### B.1 The class `ECGF2nParameterGenerator`

This class provides the integration in the `FlexiECPProvider` as described in 4.6.

```
package de.flexiprovider.ec.ecparameters;

import java.math.BigInteger;
import java.security.SecureRandom;
import java.security.NoSuchAlgorithmException;
import java.security.spec.InvalidParameterSpecException;
import java.security.*;

import satoh.SkjernaVPV;
import satoh.SatohFGH;

import de.flexiprovider.ec.ecparameters.spec.ECParameterSpec;
import de.flexiprovider.ec.arithmetic.curves.EllipticCurveGF2n;
import de.flexiprovider.ec.arithmetic.curves.PointGF2n;
import de.flexiprovider.ec.arithmetic.curves.exceptions.InvalidPointException;
import de.flexiprovider.ec.arithmetic.gf.GF2nPolynomialElement;
import de.flexiprovider.ec.arithmetic.gf.GF2nPolynomialField;
import de.flexiprovider.ec.arithmetic.gf.Bitstring;

/**
 * This class generates ec domain parameters over  $GF(2^m)$ . They are
 * used by ECDSA (see Package
 \* de.flexiprovider.ec.ecdsa Description</i></a>\), ECNR \(see 
 \\* de.flexiprovider.ec.ecnr Description</i></a>\\) and ECDH see 
 \\\* de.flexiprovider.ec.ecdh Description</i></a>\\\).
 \\\* <p>
 \\\*
 \\\* @see SkjernaVPV
 \\\* @see SatohFGH
 \\\*/
public class ECGF2nParameterGenerator extends ECParameterGenerator {

    /\\\*\\\*
```

```

* Initializes this class with the desired key length <code>size</size> and
* an instance <code>random</code> of the class <code>SecureRandom</code>.
*
* @param size the desired key length
* @param random an instance of the class <code>SecureRandom</code>
*/
protected void engineInit(int size, SecureRandom random) {
    if (random != null)
        mRandom = random;
    else {
        try {
            mRandom = SecureRandom.getInstance("SHA1");
        } catch (NoSuchAlgorithmException NSAEExc) {
            throw new RuntimeException("NoSuchAlgorithmException: "
                + NSAEExc.getMessage());
        }
    }
}

// Check the input size for primality and whether it is big enough (at
// least 163 bit).
if (size <= 162)
    size = 163;
else {
    while (!BigInteger.valueOf(size).isProbablePrime(100)) {
        size++;
    }
}

final int l = 256;
final int s = (size - 1) / l;
final int v = size - s * l;
final int g = 512;
byte[] S = new byte[g / 8];
byte[] h;
Bitstring[] bi = new Bitstring[s + 1];
Bitstring[] si = new Bitstring[s + 1];

GF2nPolynomialField F = new GF2nPolynomialField(size);
GF2nPolynomialElement a, b, j, j4;

BigInteger number;
BigInteger cofactor = BigInteger.ONE;
EllipticCurveGF2n E;
boolean safe;
do {

```

```

safe = true;
do {
    do {
        // Choose a random curve parameter b, not zero.
        try {
            // Choose a random byte array S (the seed), hash it
            // using the SHA-256 algorithm and compute b from it.
            mRandom.nextBytes(S);
            MessageDigest md = MessageDigest.getInstance("SHA-256");
            md.update(S);
            h = md.digest();
            // Choose the v rightmost bits of h.
            bi[0] = new Bitstring(v, (new BigInteger(h))
                .mod(BigInteger.ZERO.setBit(v)));
            BigInteger z = new BigInteger(S);
            // If we require more than 1 bits, we have to do hash
            // the other parts of the random bits and concatenate
            // them.
            for (int i = 1; i <= s; i++) {
                si[i] = new Bitstring(g, (z.add(BigInteger
                    .valueOf(i)))
                    .mod(BigInteger.ZERO.setBit(g)));
                md.reset();
                md.update(si[i].toByteArray());
                h = md.digest();
                bi[i] = new Bitstring(v, (new BigInteger(h))
                    .mod(BigInteger.ZERO.setBit(v)));
            }
        } catch (NoSuchAlgorithmException NSAExc) {
            throw new RuntimeException("NoSuchAlgorithmException: "
                + NSAExc.getMessage());
        }
        // Compute b from the hash (concatenate if necessary).
        Bitstring bBit = bi[0];
        for (int i = 0; i < s; i++) {
            bBit.shiftLeft(1);
            bBit = bBit.add(bi[i + 1]);
        }
        b = new GF2nPolynomialElement(F, bBit);
    } while (b.isZero()); // b = 0 does not give us a valid curve.

    // Prevents that j lies in F_4 by checking whether j^4 == j.
    j = b.invertEEA();
    j4 = (GF2nPolynomialElement) j.multiply(j).multiply(j)

```

```

        .multiply(j);
    } while (j4.equals(j));

    // Select the coefficient a randomly such that either a = 0 or
    // Tr(a) = 1.
    a = new GF2nPolynomialElement(F, mRandom);
    if(a.trace() == 0) a = new GF2nPolynomialElement(F, "ZERO");

    // From the chosen parameter b, construct a curve E.
    E = new EllipticCurveGF2n(a, b, F);

    // This is where the work is done: computing the number of points on
    // E using SkjernaaVPV, since this is the faster of the two
    // algorithms. The other option, SatohFGH, is commented out.
    number = SkjernaaVPV.NumberOfPoints(E);
    // number = SatohFGH.NumberOfPoints(E);

    // Now the security checks take place.

    // Computation of the cofactor. In the case of a = 0, we know that
    // the cofactor has to be 4 for the curve to be "good". If Tr(a) != 0,
    // it has to be 2.
    if (a.isZero()) {
        cofactor = BigInteger.valueOf(4);
    }
    else {
        cofactor = BigInteger.valueOf(2);
    }

    // Checks whether the cofactor is small enough. If not, try a new curve.
    number = number.divide(cofactor);
    if (!number.isProbablePrime(100)) {
        safe = false;
        continue;
    }

    // Checks if number = fieldsize (=2^(size)). If so, try a new curve.
    if (number.equals(BigInteger.ZERO.setBit(size))) {
        safe = false;
        continue;
    }

    else {
        // Checks if number divides (2^size)^k for some 1 <= k <= 20. If
        // so, try a new curve.

```

```

        BigInteger div = BigInteger.ZERO.setBit(size);
        for (int i = 1; i <= 20; i++) {
            if ((div.pow(i).subtract(BigInteger.ONE)).mod(number)
                .equals(BigInteger.ZERO)) {
                safe = false;
                break;
            }
        }
    }
} while (!safe); // Checks if all conditions are fulfilled. If not,
// go back and try a different curve.

PointGF2n P;

// Choose a random point P on E so that order(P) = number/cofactor.
do {
    P = new PointGF2n(E, false);
    P = (PointGF2n) P.multiply(cofactor);
} while (P.isZero());

// Create the ECPParameterSpec mSpec from the size, the point, its order,
// the cofactor and the middle exponent(s) of the field polynomial.
if (F.isTrinomial()) {
    try {
        mSpec = new ECPParameterSpec(BigInteger.valueOf(size), P,
            number, cofactor, BigInteger.valueOf(F.getTc()));
    } catch (InvalidParameterSpecException IPExc) {
        throw new InvalidPointException("InvalidParameterException: "
            + IPExc.getMessage());
    }
}

if (F.isPentanomial()) {
    try {
        mSpec = new ECPParameterSpec(BigInteger.valueOf(size), P,
            number, cofactor, BigInteger.valueOf(F.getPc()[0]),
            BigInteger.valueOf(F.getPc()[1]), BigInteger.valueOf(F
                .getPc()[2]));
    } catch (InvalidParameterSpecException IPExc) {
        throw new InvalidPointException("InvalidParameterException: "
            + IPExc.getMessage());
    }
}

// Return the seed in some way... Maybe by adding a mSeed field to the

```

```

        // ECPParameterSpec class.
    }
}

```

## B.2 The class SatohFGH

This class provides the implementation of the Satoh-FGH algorithm described in 3.4.

```

package satoh;

import java.math.BigInteger;
import de.flexiprovider.ec.arithmetic.gf.GF2nPolynomialElement;
import de.flexiprovider.ec.arithmetic.gf.GF2nPolynomialField;
import de.flexiprovider.ec.arithmetic.curves.*;

/**
 * This class computes the trace of Frobenius of an elliptic curve E
 * over a finite field F with  $q = 2^n$  elements via
 * the algorithm by Fouquet, Gaudry and Harley (based on Satoh's algorithm for
 * odd characteristic). From the trace, it computes the number of rational
 * points on that curve.
 *
 * @author Lea Poeplau
 * @see SkjernaaVPV
 */
public class SatohFGH {

    /**
     * Lifts the given cycle of GF2nPolynomialElements  $j_{i-1}$ 
     * to the  $Z_{2^n}$ Ring  $r$  with precision  $N$ .
     *
     * @param j
     *         the cycle of  $j$ -invariants to be lifted
     * @param N
     *         the precision
     * @param r
     *         the  $Z_{2^n}$ Ring to which  $j$  will be lifted
     * @return a new  $Z_{2^n}$ RingElement[], containing the cycle of lifted
     *          $j$ -invariants
     */
    private static Z2nRingElement[] LiftJInvariants(GF2nPolynomialElement[] j,
        int N, Z2nRing r) {

        int n = j[0].getDegree();

        Z2nRingElement[] J = new Z2nRingElement[n];
    }
}

```

```

    for (int i = 0; i < n; i++) {
        J[i] = new Z2nRingElement(r, j[i].getBitstring());
    }
    int k = 1;
    do {
        k = Math.min(N, 2 * k);
        J = UpdateJs(J, k);
    } while (k < N);

    return J;
}

/**
 * Used by LiftJInvariants to perform the multivariate Newton iteration.
 *
 * @param J
 *     the cycle of j-invariants to precision N/2
 * @param N
 *     the desired precision
 * @return the cycle of <i>j</i>-invariants lifted to precision <i>N</i>
 */
private static Z2nRingElement[] UpdateJs(Z2nRingElement[] J, int N) {

    int n = J.length;

    Z2nRingElement[] sqJ = new Z2nRingElement[n];
    Z2nRingElement[] D = new Z2nRingElement[n - 1];
    Z2nRingElement[] P = new Z2nRingElement[n];
    Z2nRingElement t;

    for (int i = 0; i < n; i++) {
        sqJ[i] = J[i].multiply(J[i], N);
    }

    Z2nRing r = J[0].ring;
    Z2nRingElement c5 = new Z2nRingElement(r, N, BigInteger
        .valueOf(1574640000000000L));
    for (int i = 0; i < n - 1; i++) {
        t = (modPolyDeriv(J[i], J[i + 1], sqJ[i], sqJ[i + 1], N))
            .inverse(N);
        D[i] = t.multiply(modPolyDeriv(J[i + 1], J[i], sqJ[i + 1], sqJ[i],
            N), N);
        P[i] = t.multiply(
            modPoly(J[i], J[i + 1], sqJ[i], sqJ[i + 1], c5, N), N);
    }
}

```

```

    }

    Z2nRingElement m = modPolyDeriv(J[0], J[n - 1], sqJ[0], sqJ[n - 1], N);
    Z2nRingElement f = modPoly(J[n - 1], J[0], sqJ[n - 1], sqJ[0], c5, N);

    for (int i = 0; i < n - 1; i++) {
        f = f.subtract(m.multiply(P[i], N), N);
        m = m.negate().multiply(D[i], N);
        if (m.isZeroWithPrec(N))
            break;
    }

    m = m.add(modPolyDeriv(J[n - 1], J[0], sqJ[n - 1], sqJ[0], N), N);
    P[n - 1] = f.divide(m, N);

    for (int i = n - 2; i >= 0; i--) {
        P[i] = P[i].subtract(D[i].multiply(P[i + 1], N), N);
    }

    for (int i = 0; i < n; i++) {
        J[i] = J[i].subtract(P[i], N);
    }

    return J;
}

/**
 * Computes the 2-modular polynomial applied to <i>x</i> and <i>y</i>.
 *
 * @param x
 *         the first parameter
 * @param y
 *         the second parameter
 * @return <code>\Phi<sub>2</sub>(x,y)</code>
 */
protected static Z2nRingElement modPoly(Z2nRingElement x, Z2nRingElement y,
    Z2nRingElement x2, Z2nRingElement y2, Z2nRingElement c5, int N) {
    Z2nRingElement term2 = ((x.multiply(y2, N).add(x2.multiply(y, N), N)))
        .multiply(1488, N).subtract(
        ((x.multiply(x, N).add(y.multiply(y, N), N))).multiply(
        16200, N), N);
    Z2nRingElement term3 = x.multiply(40773375, N).multiply(y, N).add(
        (x.add(y, N)).multiply(8748000000L, N), N).subtract(c5, N);
    return x.multiply(x2, N).add(y.multiply(y2, N), N).subtract(
        x2.multiply(y2, N), N).add(term2, N).add(term3, N);
}

```

```

}

/**
 * Computes the derivative with respect to X of the 2-modular polynomial
 * applied to x and y.
 *
 * @param x
 *         the first parameter
 * @param y
 *         the second parameter
 * @return  $\Phi'_{2}(x,y)$ 
 */
protected static Z2nRingElement modPolyDeriv(Z2nRingElement x,
        Z2nRingElement y, Z2nRingElement x2, Z2nRingElement y2, int N) {
    Z2nRingElement term1 = (x2.multiply(3, N)).subtract(x.multiply(2, N)
        .multiply(y2, N), N);
    Z2nRingElement term2 = ((y2.add(x.multiply(y.multiply(2, N), N), N)))
        .multiply(1488, N).subtract(x.multiply(324000, N), N);
    Z2nRingElement tmp = new Z2nRingElement(x.ring, N, BigInteger
        .valueOf(8748000000L));
    Z2nRingElement term3 = y.multiply(40773375, N).add(tmp, N);
    return term1.add(term2, N).add(term3, N);
}

/**
 * Computes the coefficient A of the Z2nRing-curve characterized by
 * the j-invariant J to precision N.
 *
 * @param J
 *         the j-invariant of the curve
 * @param N
 *         the desired precision
 * @return the coefficient of the curve in a new Z2nRingElement
 */
protected static Z2nRingElement LiftA(Z2nRingElement J, int N) {
    if (N <= 4) {
        J = new Z2nRingElement(J.ring, 4, J.poly);
        return ((J.inverse(4)).negate());
    }
    Z2nRing ring = J.ring;
    J = new Z2nRingElement(ring, N, J.poly);
    Z2nRingElement A = LiftA(J, (N - 3) / 2);
    Z2nRingElement num = ring.ONE.add(J.multiply(A.add(A.multiply(A
        .multiply(432, N), N), N), N), N);
    Z2nRingElement den = J.multiply(ring.ONE.add(A.multiply(864, N), N), N);

```

```

    A = A.subtract(num.divide(den, N), N);
    return A;
}

/**
 * Computes the non-trivial 2-torsion point of the Z2nRing-curve
 * characterized by J to precision N.
 *
 * @param A
 *         the curve's coefficient
 * @param J
 *         the curve's j-invariant
 * @param N
 *         the desired precision
 * @return the non-trivial 2-torsion point of the curve in a new
 *         Z2nRingElement
 */
protected static Z2nRingElement LiftZ(Z2nRingElement A, Z2nRingElement J,
    int N) {
    if (N <= 2) {
        J = new Z2nRingElement(J.ring, 2, J.poly);
        return (J.inverse(2));
    }
    Z2nRing ring = J.ring;
    J = new Z2nRingElement(ring, N + 1, J.poly);
    A = new Z2nRingElement(ring, N + 1, A.poly);
    Z2nRingElement Z = LiftZ(A, J, (N + 2) / 2);
    Z = new Z2nRingElement(ring, N + 1, Z.poly);
    Z2nRingElement num = Z.multiply(Z, N + 1).multiply(
        Z.multiply(8, N + 1).add(ring.ONE, N + 1), N + 1).add(A, N + 1);
    Z2nRingElement den = (Z.multiply(2, N + 1).multiply(Z.multiply(12,
        N + 1).add(ring.ONE, N + 1), N + 1));
    Z = Z.subtract(num.divide(den, N + 1), N + 1);
    return Z;
}

/**
 * Computes the trace of the Frobenius of the GF2n-curve characterized by
 * the j-invariant j0.
 *
 * @param F
 *         the curve's field
 * @param j0
 *         the curve's j-invariant
 * @return the trace of the Frobenius of the curve

```

```

*/
public static BigInteger ComputeTrace(GF2nPolynomialField F,
    GF2nPolynomialElement j0) {
    int n = j0.getDegree();
    Z2nRing.initGlobalPrecMask((n + 1) / 2 + 4);
    int N = (n + 1) / 2 + 1;
    GF2nPolynomialElement[] j = new GF2nPolynomialElement[n];
    j[0] = j0;
    for (int i = n - 1; i > 0; i--) {
        j[i] = (GF2nPolynomialElement) j[(i + 1) % n].multiply(j[(i + 1)
            % n]);
    }

    Z2nRing ring = new Z2nRing(F);

    Z2nRingElement[] J = LiftJInvariants(j, N, ring);
    System.out.println("J-invariants are lifted!");

    Z2nRingElement A;
    Z2nRingElement Z;

    Z2nRingElement Num = ring.ONE;
    Z2nRingElement Den = ring.ONE;

    for (int i = 0; i < n; i++) {
        A = LiftA(J[i], N);

        Z = LiftZ(A, J[(i + 1) % n], N - 1);

        A = A.multiply(864, N + 2);
        Num = Num.multiply(ring.ONE.subtract(Z.multiply(504, N + 2), N + 2)
            .add(A.multiply(22, N + 2), N + 2), N + 2);
        Den = Den.multiply(
            ring.ONE.add((Z.add(Z.multiply(12, N + 2)
                .multiply(Z, N + 2), N + 2)).multiply(240, N + 2),
                N + 2), N + 2).multiply(ring.ONE.add(A, N + 2),
                N + 2);
    }

    BigInteger num = Num.reduce();
    BigInteger den = Den.reduce();
    BigInteger c2 = num.multiply(den.modInverse(BigInteger.ZERO
        .setBit(N + 2)));
    BigInteger t = Z2nRingElement.TwoAdicSqrt(c2, N + 2);
    if (!t.mod(BigInteger.valueOf(4)).equals(BigInteger.ONE)) {

```

```

        t = t.negate().mod(BigInteger.ZERO.setBit(N - 1));
    }
    if (t.multiply(t).compareTo(BigInteger.ZERO.setBit(n + 2)) == 1) {
        t = t.subtract(BigInteger.ZERO.setBit(N + 1));
    }
    return t;
}

/**
 * Computes the number of points on E from the trace. The algorithm works
 * with curves of the form E:  $y^2 + xy = x^3 + b$ , i.e.  $a=0$ . Since the curves
 * E and E':  $y^2 + xy = x^3 + ax^2 + b$  (for  $\text{Tr}(a) = 1$ ) are twists of each
 * other,  $\text{trace}(E) = -\text{trace}(E')$ . We therefore compute the trace of E and
 * check whether  $a = 0$  or  $a \neq 0$  for the computation of #E.
 *
 * @param E
 *         the curve's equation
 * @return the number of points on E.
 */
public static BigInteger NumberOfPoints(EllipticCurveGF2n E) {
    GF2nPolynomialElement j0 = (GF2nPolynomialElement)E.getB().invert();
    GF2nPolynomialField F = (GF2nPolynomialField)j0.getField();
    BigInteger trace = ComputeTrace(F, j0);
    if (E.getA().trace() != 0)
        trace = trace.negate();
    return trace.negate().add(
        BigInteger.ZERO.setBit(F.getDegree()).add(BigInteger.ONE));
}
}

```

### B.3 The class SkjernaavPV

This class provides the implementation of the Skjernaav-PV algorithm which was shortly presented in 4.5.

```

package satoh;

import java.math.BigInteger;

import de.flexiprovider.ec.arithmetic.curves.EllipticCurveGF2n;
import de.flexiprovider.ec.arithmetic.gf.GF2nPolynomialElement;
import de.flexiprovider.ec.arithmetic.gf.GF2nPolynomialField;

/**
 * This class computes the trace of Frobenius of an elliptic curve E

```

```

* over a finite field  $F$  with  $q = 2^n$  elements via
* the algorithm by Skjerna and Vercauteren, Preneel and Vandewalle (based on
* Satoh's algorithm for odd characteristic). From the trace, it computes the
* number of rational points on that curve.
*
* @author Lea Poeplau
* @see SatohFGH
*/
public class SkjernaavPV {

    /**
     * This method lifts the first  $j$ -invariant. It provides the
     * base-case for the recursion.
     *
     * @param j0
     *         the  $j$ -invariant to be lifted.
     * @param N
     *         the required precision
     * @param ring
     *         the ring  $j_0$  is lifted to.
     * @return the lifted first  $j$ -invariant  $J_0$ .
     */
    protected static Z2nRingElement LiftFirstJInv(GF2nPolynomialElement j0,
        int N, Z2nRing ring) {
        int n = ring.getDegree();
        int M = (n + ((-N - 1) % n)) % n;
        int i;
        for (i = 0; i < M; ++i)
            j0 = (GF2nPolynomialElement) j0.multiply(j0);
        Z2nRingElement J = new Z2nRingElement(ring, j0.getBitstring());
        for (i = 2; i <= N; ++i)
            J = LiftPrevJInv(J, i);
        return J;
    }

    /**
     * This method computes coefficients  $A, B, C$  in  $Z_{2^n}$ 
     * such that  $\Phi_{2^{i+1}}(X, J_{2^i}) = X^3 +$ 
     *  $AX^2 + BX + C \pmod{2^N}$  and then calls the
     * recursive method LiftPrevJInvRec. This leads to the
     * method LiftFirstJInv.
     *
     * @param J
     *         the  $j$ -invariant of the preceding curve.
     * @param N

```

```

*           the required precision.
* @return the lifted j-invariant.
*/
protected static Z2nRingElement LiftPrevJInv(Z2nRingElement J, int N) {
    Z2nRingElement J2 = J.multiply(J, N);
    Z2nRingElement J3 = J2.multiply(J, N);
    Z2nRingElement A = J2.negate().add(J.multiply(1488, N), N).subtract(
        162000L, N);
    Z2nRingElement B = J2.multiply(1488, N)
        .add(J.multiply(40773375L, N), N).add(8748000000L, N);
    Z2nRingElement C = J3.add(J2.multiply(-162000L, N), N).add(
        J.multiply(8748000000L, N), N).add(-157464000000000L, N);
    Z2nRingElement[] Jarray = { J.ring.ONE, J.ring.ONE };
    Jarray = LiftPrevJInvRec(J, A, B, C, N, Jarray);
    return Jarray[0].divide(Jarray[1], N);
}

/**
* This method performs the Newton iteration on the cubic polynomial
*  $X^3 + AX^2 + BX + C$ .
*
* @param J
*           the j-invariant of the preceding curve.
* @param A
*           the coefficient A of  $X^3$ .
* @param B
*           the coefficient B of  $X^2$ .
* @param C
*           the coefficient C of  $X$ .
* @param N
*           the required precision
* @param J2
*           the returned Z2nRingElement[] containing
*           numerator and denominator of the lifted j-invariant.
* @return the numerator and denominator of the lifted j-invariant.
*/
protected static Z2nRingElement[] LiftPrevJInvRec(Z2nRingElement J,
    Z2nRingElement A, Z2nRingElement B, Z2nRingElement C, int N,
    Z2nRingElement[] J2) {
    if (N == 1) {
        J2[0] = J.multiply(J, 1);
        return J2;
    } else {
        int N2 = (N + 1) / 2;
        J2 = LiftPrevJInvRec(J, A, B, C, N2, J2);
    }
}

```

```

        boolean first = true;
        Z2nRingElement F, G;
        if (first) {
            Z2nRingElement T = J2[1].multiply(J2[1], N);
            Z2nRingElement U = J2[1].multiply(T, N);
            Z2nRingElement V = A.multiply(J2[1], N);
            F = J2[0].multiply(J2[0], N).multiply(
                J2[0].multiply(2, N).add(V, N), N).subtract(
                C.multiply(U, N), N);
            G = (J2[0].multiply(J2[0].multiply(3, N).add(V.multiply(2, N),
                N), N).add(B.multiply(T, N), N)).multiply(J2[1], N);
        } else {
            Z2nRingElement X = J2[0];
            Z2nRingElement Y = J2[1];
            Z2nRingElement U = X.multiply(2, N);
            Z2nRingElement V = X.multiply(X, N);
            Z2nRingElement W = V.multiply(Y, N);
            Z2nRingElement T = Y.multiply(Y, N);
            Z2nRingElement S = Y.multiply(T, N);
            F = V.multiply(U, N).add(A.multiply(W, N), N).subtract(
                C.multiply(S, N), N);
            G = W.multiply(3, N).add(T.multiply(U.multiply(A, N), N), N)
                .add(B.multiply(S, N), N);
        }
        J2[0] = F;
        J2[1] = G;
        return J2;
    }
}

/**
 * Computes the trace of the Frobenius of the GF2n-curve characterized by
 * the j-invariant <i>j<sub>0</sub></i>.
 *
 * @param F
 *         the curve's field
 * @param j0
 *         the curve's j-invariant
 * @return the trace of the Frobenius of the curve
 */
public static BigInteger ComputeTrace(GF2nPolynomialField F,
    GF2nPolynomialElement j0) {
    int n = j0.getDegree();
    Z2nRing.initGlobalPrecMask((n + 1) / 2 + 4 + 20);
    int N = (n + 1) / 2 + 13;

```

```

int N2 = (n + 1) / 2 + 3;
// int N2 = N - 10;

Z2nRing ring = new Z2nRing(F);
Z2nRingElement J;
Z2nRingElement S = ring.ONE;
Z2nRingElement T = ring.ONE;

J = LiftFirstJInv(j0, N, ring);

for (int i = 0; i < n; i++) {
    Z2nRingElement Z, V, tmp;
    Z2nRingElement J2 = LiftPrevJInv(J, N);

    tmp = J.multiply(J, N);

    Z = tmp.add(J.multiply(195120L, N), N).add(J2.multiply(4095, N), N)
        .add(660960000L, N);

    tmp = tmp.add(J.multiply(563760L, N), N).subtract(
        J.multiply(J2, N).multiply(512, N), N);
    tmp = tmp.add(J2.multiply(372735, N), N).add(8981280000L, N);
    tmp = tmp.shiftRight(9).negate();
    Z = Z.shiftRight(12).multiply(tmp.inverse(N), N);

    V = Z.multiply(Z.multiply(12, N2).add(ring.ONE, N2), N2);
    V = V.multiply(J2.subtract(1728, N2), N2);
    V = V.subtract(36, N2);

    S = S.multiply(J2.subtract(
        V.multiply(ring.ONE.add(Z.multiply(24, N2), N2), N2)
            .multiply(504, N2), N2), N2);
    T = T.multiply(V.multiply(240, N2).add(J2, N2), N2);

    J = J2;
}

BigInteger num = S.reduce();
BigInteger den = T.reduce();
BigInteger c2 = num
    .multiply(den.modInverse(BigInteger.ZERO.setBit(N2)));
BigInteger t = Z2nRingElement.TwoAdicSqrt(c2, N2);
if (!t.mod(BigInteger.valueOf(4)).equals(BigInteger.ONE)) {
    t = t.negate().mod(BigInteger.ZERO.setBit(N2 - 3));
}

```

```

        if (t.multiply(t).compareTo(BigInteger.ZERO.setBit(n + 2)) == 1) {
            t = t.subtract(BigInteger.ZERO.setBit(N2 - 1));
        }
        return t;
    }

    /**
     * Computes the number of points on E from the trace. The algorithm works
     * with curves of the form E: y^2 + xy = x^3 + b, i.e. a=0. Since the curves
     * E and E': y^2 + xy = x^3 + ax^2 + b (for Tr(a) = 1) are twists of each
     * other, trace(E) = -trace(E'). We therefore compute the trace of E and
     * check whether a = 0 or a != 0 for the computation of #E.
     *
     * @param E
     *         the curve's equation
     * @return the number of points on E.
     */
    public static BigInteger NumberOfPoints(EllipticCurveGF2n E) {
        GF2nPolynomialElement j0 = (GF2nPolynomialElement)E.getB().invert();
        GF2nPolynomialField F = (GF2nPolynomialField)j0.getField();
        BigInteger trace = ComputeTrace(F, j0);
        if (E.getA().trace() != 0)
            trace = trace.negate();
        return trace.negate().add(
            BigInteger.ZERO.setBit(F.getDegree()).add(BigInteger.ONE));
    }
}

```

## B.4 The class `Z2nRing`

In this class, we implement the structure of the ring  $\mathbb{Z}_{2^n}$ .

```

package satoh;

import de.flexiprovider.ec.arithmetic.gf.*;
import java.math.BigInteger;

/**
 * This class implements the abstract class <code>pAdicRing</code> for <i>n>1</i>.
 * It holds the ringpolynomial.
 *
 * Z2nRing is used by Z2nRingElement which implements the elements of this ring.
 *
 * @author Lea Poeplau
 * @see Z2nRingElement
 */

```

```
*/  
  
public class Z2nRing {  
  
    /**  
     * The degree of the ring over Z_2.  
     */  
    private int degree;  
  
    /**  
     * The underlying field F_{2^degree}.  
     */  
    private GF2nPolynomialField field;  
  
    /**  
     * The zero element.  
     */  
    final public Z2nRingElement ZERO;  
  
    /**  
     * The one element.  
     */  
    final public Z2nRingElement ONE;  
  
    /**  
     * The element two.  
     */  
    final public Z2nRingElement TWO;  
  
    /**  
     * Type of the ring polynomial (tri- or pentanomial).  
     */  
    int polyType;  
  
    /**  
     * Specifier for the polyType (trinomial).  
     */  
    static final int IS_TRINOM_POLY_TYPE = 1;  
  
    /**  
     * Specifier for the polyType (pentanomial).  
     */  
    static final int IS_PENTANOM_POLY_TYPE = 2;  
  
    /**
```

```

    * The positions of the nonzero coefficients of the ring polynomial.
    */
int[] ringPolyCoefs;

protected static BigInteger[] globalPrecMask; // the precision mask array

/**
 * Reads the <i>m</i>th entry of the globalPrecMask array and returns it if
 * it is not null. Otherwise, it sets it to <i>2<sup>m</sup>-1</i> and
 * return it.
 *
 * @param m
 *         the requested entry
 * @return <i>2<sup>m</sup>-1</i>
 */
static BigInteger getGlobalPrecMask(int m) {
    if (globalPrecMask[m] == null)
        globalPrecMask[m] = BigInteger.ZERO.setBit(m).subtract(
            BigInteger.ONE);
    return globalPrecMask[m];
}

/**
 * Initialites the globalPrecMask array that is used for faster calculations
 * in the required precision.
 *
 * @param N
 *         the length of the array, set to the maximal needed length.
 */
static void initGlobalPrecMask(int N) {
    globalPrecMask = new BigInteger[N];
}

// ***** CONSTRUCTOR *****

/**
 * Constructs an instance of the ring <i>Z<sub>2</sub><sup>n</sup></i>
 * using the fieldpolynomial of <i>F</i> as ring polynomial.
 *
 * @param F
 *         the GF2nPolynomialField providing the fieldpolynomial.
 */
public Z2nRing(GF2nPolynomialField F) {
    field = F;
    degree = F.getDegree();
}

```

```

ZERO = new Z2nRingElement(this, 2, BigInteger.ZERO);
ONE = new Z2nRingElement(this, 2, BigInteger.ONE);
TWO = new Z2nRingElement(this, 2, BigInteger.valueOf(2));
Bitstring rp = new Bitstring(F.getFieldPolynomial());
ringPolyCoefs = new int[3];
int count = 0;
for (int i = 1; i < rp.getLength() - 1; i++) {
    if (rp.testBit(i)) {
        ringPolyCoefs[count] = i;
        count++;
    }
}
if (count == 1)
    polyType = IS_TRINOM_POLY_TYPE;
else if (count == 3)
    polyType = IS_PENTANOM_POLY_TYPE;
else
    throw new RuntimeException("Wrong poly");
}

// ***** METHODS *****

/**
 * Decides, whether the given ring <code>a</code> is the same as this
 * ring.
 *
 * @return (this == other)
 */
public boolean equals(Z2nRing a) {
    if (degree != a.getDegree()) {
        System.err.println("Not the same degree!");
        return false;
    }
    for (int i = 0; i < 3; i++) {
        if (ringPolyCoefs[i] != a.ringPolyCoefs[i])
            return false;
    }
    return true;
}

/**
 * Returns the degree of this ring.
 *
 * @return this.degree
 */

```

```

public int getDegree() {
    return degree;
}

/**
 * Returns the field underlying this ring.
 *
 * @return this.field
 */
public GF2nPolynomialField getField() {
    return field;
}

/**
 * Returns a String representing this ring by its degree, its ring
 * polynomial and its precision.
 *
 * @return a String (degree, ringPoly, prec)
 */

public String toString() {
    StringBuffer sb = new StringBuffer();
    sb.append("Degree = " + degree + ", ringPoly = 1");
    for (int i = 0; i < 3; i++) {
        if (ringPolyCoefs[i] != 0)
            sb.append(" + t^" + ringPolyCoefs[i] + " ");
    }
    sb.append(" + t^" + degree);
    return sb.toString();
}
}

```

## B.5 The class `Z2nRingElement`

In this class, we implemented the arithmetic structure of the elements of the ring  $\mathbb{Z}_2^n$ .

```

package satoh;

import de.flexiprovider.ec.arithmetic.gf.*;
import java.math.BigInteger;

/**
 * This class implements the abstract class Z2nRingElement for
 *  $\mathbb{Z}_2^n$ . It holds the coefficients of the element in polynomial
 * representation (over  $\mathbb{Z}_2$ ) to precision  $m$ .
 */

```

```

* @author Lea Poeplau
* @see Z2nRing
*
*/
public class Z2nRingElement {

    /**
     * The Z2nRing this element lies in.
     */
    final protected Z2nRing ring;

    /**
     * The coefficient array of this element in polynomial representation.
     */
    final protected BigInteger[] poly;

    /**
     * The degree of the polynomial representation of this element.
     */
    final protected int degree;

    // ***** CONSTRUCTORS *****

    /**
     * Creates a new Z2nRingElement using the given ring and array. The
     * BigInteger[] poly has as coefficient array the given array.
     *
     * @param r
     *         the Z2nRing to use.
     * @param s
     *         the coefficient array poly
     * @param prec
     *         the precision of this element.
     */
    public Z2nRingElement(Z2nRing r, int prec, BigInteger[] s) {
        ring = r;
        poly = new BigInteger[s.length];
        System.arraycopy(s, 0, poly, 0, s.length);

        // Reduce mod ringpoly
        if (r.polyType == 1)
            reduceTrinomial();
        else
            reducePentanomial();
    }

```

```

    // Reduce mod 2^N
    BigInteger precMask = Z2nRing.getGlobalPrecMask(prec);
    for (int i = 0; i < poly.length; i++)
        poly[i] = poly[i].and(precMask);

    // Compute degree
    degree = computeDeg();
}

/**
 * Creates a new Z2nRingElement using the given ring and array. The
 * BigInteger[] poly is the given array. This private constructor is only
 * called during the arithmetic operations, where the precision is already
 * dealt with.
 *
 * @param r
 *         the Z2nRing to use.
 * @param s
 *         the coefficient array poly
 */
private Z2nRingElement(Z2nRing r, BigInteger[] s) {
    ring = r;
    poly = s;
    degree = computeDeg();
}

/**
 * Creates a new Z2nRingElement using the given ring and the BigInteger <i>x</i>
 * as the value. The constant coefficient is set to <i>x mod 2<sup>prec</sup></i>.
 *
 * @param r
 *         the Z2nRing to use.
 * @param prec
 *         the precision of this element
 * @param x
 *         the BigInteger value of this element.
 */
public Z2nRingElement(Z2nRing r, int prec, BigInteger x) {
    this(r, x.and(Z2nRing.getGlobalPrecMask(prec)));
}

/**
 * Creates a new Z2nRingElement using the given ring and the BigInteger <i>x</i>
 * as the value. This private method is only called with a BigInteger <i>x</i>
 * that is already reduced to the required precision.

```

```

*
* @param r
*         the Z2nRing to use.
* @param x
*         the BigInteger value of this element.
*/
private Z2nRingElement(Z2nRing r, BigInteger x) {
    ring = r;
    poly = new BigInteger[1];
    poly[0] = x;
    degree = computeDeg();
}

// The following method doesn't reduce the resulting poly mod the ringpoly!
/**
 * Creates a new Z2nRingElement from the given Bitstring by setting the
 * corresponding coefficients to 1. Since this is only called with
 * Bitstrings that are already reduced modulo the ring polynomial, this is
 * not done here.
 *
 * @param r
 *         the Z2nRing to use.
 * @param b
 *         the Bitstring representing this element.
 */
public Z2nRingElement(Z2nRing r, Bitstring b) {
    ring = r;
    poly = new BigInteger[b.getLength()];

    // Lift bitstring
    for (int i = 0; i < b.getLength(); i++) {
        poly[i] = b.testBit(i) ? BigInteger.ONE : BigInteger.ZERO;
    }

    degree = computeDeg();
}

// ***** METHODS *****

/**
 * Returns a Z2nRingElement whose coefficient's value is
 * <code>poly[i] >> n</code>.
 *
 * @param n
 *         the shift distance

```

```

    * @return a new Z2nRingElement whose coefficient's value is
    *         <code>poly[i] >> n</code>.
    */
public Z2nRingElement shiftRight(int n) {
    BigInteger[] newCoeff = new BigInteger[degree + 1];
    for (int i = 0; i <= degree; i++)
        newCoeff[i] = poly[i].shiftRight(n);
    return new Z2nRingElement(ring, newCoeff);
}

/**
 * Returns a Z2nRingElement whose value is <code>this/B</code> in
 * precision <i>prec</i>.
 *
 * @param B
 *         the Z2nRingElement by which this is to be divided
 * @param prec
 *         the precision the result is represented in.
 * @return <code>this/B</code>
 */
public Z2nRingElement divide(Z2nRingElement B, int prec) {
    if (!ring.equals(B.ring))
        throw new ArithmeticException("Elements are not in the same ring!");

    Z2nRingElement A = this;

    int lowestBitA = A.lowestBit();
    int lowestBitB = B.lowestBit();

    if (lowestBitB > lowestBitA)
        throw new ArithmeticException("Division not possible!");

    // Cancel out factor (2^(lowestBitA+1))
    A = A.shiftRight(lowestBitB);
    B = B.shiftRight(lowestBitB);

    Z2nRingElement Q = A.multiply(B.inverse(prec), prec);

    return Q;
}

/**
 * Returns true if two Z2nRingElements have the same ring and polynomial and
 * are thus equal.
 */

```

```

* @param a
*         the other Z2nRingElement
* @return true if <i>this</i> equals <i>a</i> (<i>this</i> == <i>a</i>)
*/
public boolean equals(Z2nRingElement a) {
    if (degree != a.degree)
        return false;
    if (!ring.equals(a.ring))
        return false;
    for (int i = 0; i <= degree; i++) {
        if (!poly[i].equals(a.poly[i]))
            return false;
    }
    return true;
}

/**
* Returns the Z2nRingElement <code>this</code><sup>-1</sup> to
* precision <code>prec</code>.
*
* @param prec
*         the precision the result is represented in.
* @return <code>this</code><sup>-1</sup>
*/
public Z2nRingElement inverse(int prec) {
    GF2nPolynomialField F = ring.getField();
    GF2nPolynomialElement k = reduceF2n(F);
    k = k.invertEEA();
    Z2nRingElement x = new Z2nRingElement(ring, k.getBitstring());
    int n = 1;
    do {
        n = Math.min(2 * n, prec);
        x = x.multiply(ring.TWO.subtract(this.multiply(x, n), n), n);
    } while (n < prec);
    return x;
}

/**
* Returns the Z2nRingElement <code>this</code>+<code>a</code> to
* precision <code>prec</code>.
*
* @param a
*         the Z2nRingElement to be added to this element.
* @param prec
*         the precision the result is represented in.

```

```

    * @return <code>this</code>+<code>a</code>
    */
    public Z2nRingElement add(Z2nRingElement a, int prec) {
        if (!ring.equals(a.ring))
            throw new ArithmeticException("Elements are not in the same ring!");

        int minDeg, maxDeg;

        if (degree < a.degree) {
            minDeg = degree;
            maxDeg = a.degree;
        } else {
            minDeg = a.degree;
            maxDeg = degree;
        }

        BigInteger[] newCoeff = new BigInteger[maxDeg + 1];
        BigInteger precMask = Z2nRing.getGlobalPrecMask(prec);
        int i;

        for (i = 0; i <= minDeg; i++)
            newCoeff[i] = poly[i].add(a.poly[i]).and(precMask);
        for (; i <= degree; i++)
            newCoeff[i] = poly[i].and(precMask);
        for (; i <= a.degree; i++)
            newCoeff[i] = a.poly[i].and(precMask);

        return new Z2nRingElement(ring, newCoeff);
    }

    /**
     * Adds the Z2nRingElement <code>a</code> to <code>this</code> and
     * writes the result to precision <code>prec</code> to <code>this</code>.
     *
     * @param a
     *         the Z2nRingElement to be added to this element.
     * @param prec
     *         the precision the result is represented in.
     */
    public void addToThis(Z2nRingElement a, int prec) {
        if (!ring.equals(a.ring))
            throw new ArithmeticException("Elements are not in the same ring!");

        int minDeg = (degree < a.degree) ? degree : a.degree;

```

```

    BigInteger precMask = Z2nRing.getGlobalPrecMask(prec);
    int i;

    for (i = 0; i <= minDeg; i++)
        poly[i] = poly[i].add(a.poly[i]).and(precMask);
    for (; i <= degree; i++)
        poly[i] = poly[i].and(precMask);
    for (; i <= a.degree; i++)
        poly[i] = a.poly[i].and(precMask);
}

/**
 * Returns the Z2nRingElement <code>this</code>+<code>a</code> to
 * precision <code>prec</code>. The long <code>a</code> is added to the
 * constant coefficient of the array <code>poly</code>.
 *
 * @param a
 *         the long to be added to this element.
 * @param prec
 *         the precision the result is represented in.
 * @return <code>this</code>+<code>a</code>
 */
public Z2nRingElement add(long a, int prec) {
    return add(BigInteger.valueOf(a), prec);
}

/**
 * Returns the Z2nRingElement <code>this</code>+<code>a</code> to
 * precision <code>prec</code>. The BigInteger <code>a</code> is added
 * to the constant coefficient of the array <code>poly</code>.
 *
 * @param a
 *         the BigInteger to be added to this element.
 * @param prec
 *         the precision the result is represented in.
 * @return <code>this</code>+<code>a</code>
 */
public Z2nRingElement add(BigInteger a, int prec) {
    BigInteger precMask = Z2nRing.getGlobalPrecMask(prec);
    if (degree == -1)
        return new Z2nRingElement(ring, a.and(precMask));

    BigInteger[] newCoeff = new BigInteger[degree + 1];
    System.arraycopy(poly, 0, newCoeff, 0, degree + 1);
    newCoeff[0] = newCoeff[0].add(a).and(precMask);
}

```

```

    return new Z2nRingElement(ring, newCoeff);
}

/**
 * Returns the Z2nRingElement this-a to
 * precision prec. The long a is subtracted
 * from the constant coefficient of the array poly.
 *
 * @param a
 *         the long to be subtracted from this element.
 * @param prec
 *         the precision the result is represented in.
 * @return this-a
 */
public Z2nRingElement subtract(long a, int prec) {
    return add(-a, prec);
}

/**
 * Returns the Z2nRingElement this-a to
 * precision prec. The BigInteger a is
 * subtracted from the constant coefficient of the array poly.
 *
 * @param a
 *         the BigInteger to be subtracted from this element.
 * @param prec
 *         the precision the result is represented in.
 * @return this-a
 */
public Z2nRingElement subtract(BigInteger a, int prec) {
    return add(a.negate(), prec);
}

/**
 * Returns the Z2nRingElement this-a to
 * precision prec.
 *
 * @param a
 *         the Z2nRingElement to be subtracted from this element.
 * @param prec
 *         the precision the result is represented in.
 * @return this-a
 */
public Z2nRingElement subtract(Z2nRingElement a, int prec) {
    if (!ring.equals(a.ring))

```

```

        throw new ArithmeticException("Elements are not in the same ring!");

    int minDeg, maxDeg;

    if (degree < a.degree) {
        minDeg = degree;
        maxDeg = a.degree;
    } else {
        minDeg = a.degree;
        maxDeg = degree;
    }

    BigInteger[] newCoeff = new BigInteger[maxDeg + 1];
    BigInteger precMask = Z2nRing.getGlobalPrecMask(prec);
    int i;

    for (i = 0; i <= minDeg; i++)
        newCoeff[i] = poly[i].subtract(a.poly[i]).and(precMask);
    for (; i <= degree; i++)
        newCoeff[i] = poly[i].and(precMask);
    for (; i <= a.degree; i++)
        newCoeff[i] = a.poly[i].negate().and(precMask);

    return new Z2nRingElement(ring, newCoeff);
}

/**
 * Subtracts the Z2nRingElement <code>a</code> from <code>this</code>
 * and writes the result to precision <code>prec</code> to
 * <code>this</code>.
 *
 * @param a
 *         the Z2nRingElement to be subtracted from this element.
 * @param prec
 *         the precision the result is represented in.
 */
public void subtractFromThis(Z2nRingElement a, int prec) {
    if (!ring.equals(a.ring))
        throw new ArithmeticException("Elements are not in the same ring!");

    int minDeg = (degree < a.degree) ? degree : a.degree;

    BigInteger precMask = Z2nRing.getGlobalPrecMask(prec);
    int i;

```

```

    for (i = 0; i <= minDeg; i++)
        poly[i] = poly[i].subtract(a.poly[i]).and(precMask);
    for (; i <= degree; i++)
        poly[i] = poly[i].and(precMask);
    for (; i <= a.degree; i++)
        poly[i] = a.poly[i].negate().and(precMask);
}

/**
 * Returns the largest integer k such that
 * this mod  $2^k = 0$ .
 *
 * @return the largest integer k such that this
 *         mod  $2^k = 0$ .
 */
public int lowestBit() {
    int lowestBit = java.lang.Integer.MAX_VALUE;
    for (int i = 0; i <= degree; i++) {
        int tmp = poly[i].getLowestSetBit();
        if (tmp >= 0 && tmp < lowestBit)
            lowestBit = tmp;
    }
    return lowestBit;
}

/**
 * Returns true if this = 0 mod  $2^m$ .
 *
 * @param m
 *         the precision to which this is checked to be
 *         zero.
 * @return this == 0 mod  $2^m$ .
 */
public boolean isZeroWithPrec(int m) {
    if (m <= lowestBit())
        return true;
    return false;
}

/**
 * Returns the Z2nRingElement this*a to
 * precision prec.
 *
 * @param a
 *         the Z2nRingElement this element is to be multiplied by.

```

```

* @param prec
*         the precision the result is represented in.
* @return <code>this</code>*<code>a</code>
*/
public Z2nRingElement multiply(Z2nRingElement a, int prec) {
    if (!ring.equals(a.ring))
        throw new ArithmeticException("Elements are not in the same ring!");
    if (degree == -1 | a.degree == -1)
        return ring.ZERO;
    int N = Math.max(degree, a.degree) + 1;

    if (N != 1 && N % 2 != 0)
        N++;

    BigInteger precMask = Z2nRing.getGlobalPrecMask(prec);
    BigInteger[] p = new BigInteger[N];
    for (int i = 0; i < degree + 1; i++) {
        p[i] = poly[i];
    }
    for (int i = degree + 1; i < N; i++) {
        p[i] = BigInteger.ZERO;
    }

    if (this.equals(a))
        return new Z2nRingElement(ring, prec, square(p, N, precMask));

    BigInteger[] q = new BigInteger[N];
    for (int i = 0; i < a.degree + 1; i++) {
        q[i] = a.poly[i];
    }
    for (int i = a.degree + 1; i < N; i++) {
        q[i] = BigInteger.ZERO;
    }

    BigInteger[] result = new BigInteger[2 * N - 1];
    for (int i = 0; i < 2 * N - 1; i++) {
        result[i] = BigInteger.ZERO;
    }

    Z2nRingElement x = new Z2nRingElement(ring, prec, multiplyKA(p, q, N,
        precMask));
    return x;
}

/**

```

```

* This method implements Karatsuba polynomial multiplication.
*
* @param p
*         a BigInteger[] representing the first factor.
* @param q
*         a BigInteger[] representing the second factor.
* @param N
*         the maximum of the degrees of the polynomials rounded to an
*         even number.
* @param precMask
* @return a BigInteger[] containing the product of <code>p</code> and
*         <code>q</code>.
*/
private BigInteger[] multiplyKA(BigInteger[] p, BigInteger[] q, int N,
    BigInteger precMask) {
    if (N <= 4)
        return multiplyNaive(p, q, 2 * N - 1, precMask);

    BigInteger[] r = new BigInteger[2 * N - 1];
    for (int i = 0; i < 2 * N - 1; i++) {
        r[i] = BigInteger.ZERO;
    }

    int N2 = N / 2;
    N2 = N2 + N2 % 2;

    BigInteger[] t1 = new BigInteger[N2];
    BigInteger[] t2 = new BigInteger[N2];

    BigInteger[] rtmp;
    int i;

    for (i = 0; i < N / 2; i++) {
        t1[i] = p[i].add(p[i + N / 2]);
        t2[i] = q[i].add(q[i + N / 2]);
    }

    if (N / 2 != N2) {
        t1[N / 2] = BigInteger.ZERO;
        t2[N / 2] = BigInteger.ZERO;
    }

    rtmp = multiplyKA(t1, t2, N2, precMask);
    for (i = 0; i < N - 1; i++) {
        r[N / 2 + i] = rtmp[i];
    }
}

```

```

    }

    for (i = 0; i < N / 2; i++) {
        t1[i] = p[i + N / 2];
        t2[i] = q[i + N / 2];
    }
    if (N2 != N / 2) {
        p[N / 2] = BigInteger.ZERO;
        q[N / 2] = BigInteger.ZERO;
    }

    rtmp = multiplyKA(p, q, N2, precMask);
    for (i = 0; i < N - 1; i++) {
        r[i] = r[i].add(rtmp[i]);
        r[N / 2 + i] = r[N / 2 + i].subtract(rtmp[i]);
    }

    rtmp = multiplyKA(t1, t2, N2, precMask);

    for (i = 0; i < N - 1; i++) {
        r[N + i] = r[N + i].add(rtmp[i]);
        r[N / 2 + i] = r[N / 2 + i].subtract(rtmp[i]);
    }

    return r;
}

/**
 * This method squares an element using Karatsuba polynomial multiplication.
 *
 * @param p
 *         a BigInteger[] representing the polynomial to be squared.
 * @param N
 *         the degree of the polynomial rounded to an even number.
 * @param precMask
 * @return a BigInteger[] containing the square of <code>p</code>.
 */
private BigInteger[] square(BigInteger[] p, int N, BigInteger precMask) {
    BigInteger[] r = new BigInteger[2 * N - 1];
    for (int i = 0; i < 2 * N - 1; i++) {
        r[i] = BigInteger.ZERO;
    }

    if (N <= 9)
        return multiplyNaive(p, p, 2 * N - 1, precMask);

```

```
int N2 = N / 2;
if (N2 % 2 != 0)
    N2++;

BigInteger[] t1 = new BigInteger[N2];
BigInteger[] r1;
BigInteger[] rm;
BigInteger[] rh;
int i;

for (i = 0; i < N / 2; i++)
    t1[i] = p[i].add(p[i + N / 2]);

if (N / 2 != N2)
    t1[N / 2] = BigInteger.ZERO;

rm = square(t1, N2, precMask);

for (i = 0; i < N / 2; i++)
    t1[i] = p[i + N / 2];

if (N2 != N / 2)
    p[N / 2] = BigInteger.ZERO;

r1 = square(p, N2, precMask);
rh = square(t1, N2, precMask);

for (i = 0; i < N - 1; i++) {
    r[i] = r1[i];
}

for (i = 0; i < N - 1; i++) {
    r[N + i] = rh[i];
}

for (i = 0; i < N - 1; i++) {
    r[N / 2 + i] = r[N / 2 + i].add(rm[i]).subtract(r1[i]);
    if (i < rh.length)
        r[N / 2 + i] = r[N / 2 + i].subtract(rh[i]);
}
return r;
}

/**
```

```

* Multiplies two elements naively and returns the result in a BigInteger[].
*
* @param p
*       a BigInteger[] representing the first factor.
* @param q
*       a BigInteger[] representing the second factor.
* @param N
*       the maximum of the degrees of the polynomials rounded to an
*       even number.
* @param precMask
* @return a BigInteger[] containing the product of <code>p</code> and
*         <code>q</code>.
*/
private BigInteger[] multiplyNaive(BigInteger[] p, BigInteger[] q, int N,
    BigInteger precMask) {
    int n = (N + 1) / 2;
    BigInteger[] prod = new BigInteger[N];
    for (int i = 0; i < N; i++) {
        prod[i] = BigInteger.ZERO;
    }
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            prod[i + j] = prod[i + j].add(p[i].multiply(q[j]));
        }
    }
    for (int i = 0; i < N; i++) {
        prod[i] = prod[i].and(precMask);
    }
    return prod;
}

/**
* Multiplies this element by <code>a</code> and returns the result to
* precision <code>prec</code>.
*
* @param a
*       the long this elements is multiplied by.
* @param prec
*       the precision the result is represented in.
* @return <code>this</code>*<code>a</code>
*/
public Z2nRingElement multiply(long a, int prec) {
    if (prec < 64)
        a = a % (1L << prec);
    if (a == 0L)

```

```

        return ring.ZERO;
    BigInteger precMask = Z2nRing.getGlobalPrecMask(prec);
    BigInteger A = BigInteger.valueOf(a).and(precMask);
    BigInteger[] k = new BigInteger[poly.length];
    for (int i = 0; i < k.length; i++) {
        k[i] = poly[i].multiply(A).and(precMask);
    }
    return new Z2nRingElement(ring, k);
}

/**
 * Returns -<code>this</code>.
 *
 * @return -<code>this</code>.
 */
public Z2nRingElement negate() {
    if (degree == -1)
        return this;
    BigInteger[] ret = new BigInteger[degree + 1];
    for (int i = 0; i <= degree; i++) {
        ret[i] = poly[i].negate();
    }
    return new Z2nRingElement(ring, ret);
}

/**
 * Negates this element by setting its polynomial coefficients to their
 * negative.
 */
public void negateThis() {
    if (degree == -1)
        return;
    for (int i = 0; i <= degree; i++) {
        poly[i] = poly[i].negate();
    }
}

/**
 * Reduces <i>this</i> to a new BigInteger.
 *
 * @return a new BigInteger (<code>this.poly.getCoAt(0)</code>)
 * @throws ArithmeticException
 *         <code>poly[i] != 0</code> for some <i>i != 0</i>
 */
public BigInteger reduce() {

```

```

    boolean allZero = true;
    for (int i = 1; i < degree; i++) {
        if (!(poly[i].equals(BigInteger.ZERO))) {
            allZero = false;
            break;
        }
    }
    if (!allZero)
        throw new ArithmeticException(
            "Reduction not possible since the element does not lie in Z_p!");
    return poly[0];
}

/**
 * Computes the 2-adic square root of <i>a</i> to precision <i>N</i>.
 *
 * @param a
 *         the BigInteger who's square root is computed
 * @param N
 *         the precision
 * @return the 2-adic square root of <i>a</i> to precision <i>N</i>
 */
protected static BigInteger TwoAdicSqrt(BigInteger a, int N) {
    if (a.equals(BigInteger.ONE))
        return a;
    BigInteger y = BigInteger.ONE;
    int i = a.getLowestSetBit();

    while (i < N) {
        if ((y.multiply(y).subtract(a)).abs().testBit(i))
            y = y.add(BigInteger.ZERO.setBit(i - 1));
        i++;
    }
    return y;
}

/**
 * Reduces <i>this</i> to a new GF2nPolynomialElement in the given field
 * <i>F</i>.
 *
 * @param F
 *         the GF2nPolynomialField this.reduceF2n will be in
 * @return a new GF2nPolynomialElement
 */
public GF2nPolynomialElement reduceF2n(GF2nPolynomialField F) {

```

```

    Bitstring b = new Bitstring(degree + 1);
    for (int i = 0; i <= degree; i++) {
        if (poly[i].testBit(0))
            b.setBit(i);
    }
    return new GF2nPolynomialElement(F, b);
}

/**
 * Returns the polynomial String representation of this Z2nRingElement.
 *
 * @return polynomial String representation of this Z2nRingElement.
 */
public String toString() {
    StringBuffer sb = new StringBuffer();
    if (poly.length == 0) {
        sb.append(0);
        return sb.toString();
    }

    if (poly[0].intValue() != 0)
        sb.append(poly[0]);
    if (poly.length > 1 && (poly[1].intValue() != 0)) {
        if (sb.length() > 0) {
            if (poly[1].intValue() == 1)
                sb.append(" + t");
            else {
                sb.append(" + " + poly[1] + "t");
            }
        } else {
            if (poly[1].intValue() == 1)
                sb.append("t");
            else
                sb.append(poly[1] + "t");
        }
    }

    for (int i = 2; i < poly.length; i++) {
        if (poly[i].intValue() != 0) {
            if (sb.length() > 0) {
                if (poly[i].intValue() == 1)
                    sb.append(" + t^" + i);
                else {
                    sb.append(" + " + poly[i] + "t^" + i);
                }
            } else {

```

```

        if (poly[i].intValue() == 1)
            sb.append("t^" + i);
        else {
            sb.append(poly[i] + "t^" + i);
        }
    }
}

if (sb.length() == 0)
    sb.append("0");
return sb.toString();
}

/**
 * Reduces this element modulo the ring polynomial, if that is a
 * pentanomial.
 */
private void reducePentanomial() {
    int n = ring.getDegree();
    int[] k = ring.ringPolyCoefs;
    int d;

    for (d = poly.length - 1; d >= n; d--) {
        BigInteger lead = poly[d];
        if (!lead.equals(BigInteger.ZERO)) {
            poly[d] = BigInteger.ZERO;
            poly[d - (n - k[2])] = poly[d - (n - k[2])].subtract(lead);
            poly[d - (n - k[1])] = poly[d - (n - k[1])].subtract(lead);
            poly[d - (n - k[0])] = poly[d - (n - k[0])].subtract(lead);
            poly[d - n] = poly[d - n].subtract(lead);
        }
    }
}

/**
 * Reduces this element modulo the ring polynomial, if that is a trinomial.
 */
private void reduceTrinomial() {
    int n = ring.getDegree();
    int[] k = ring.ringPolyCoefs;
    int d;

    for (d = poly.length - 1; d >= n; d--) {
        BigInteger lead = poly[d];

```

```
        if (!lead.equals(BigInteger.ZERO)) {
            poly[d] = BigInteger.ZERO;
            poly[d - (n - k[0])] = poly[d - (n - k[0])].subtract(lead);
            poly[d - n] = poly[d - n].subtract(lead);
        }
    }
}

/**
 * Computes the degree of the array <code>poly</code> and returns the
 * result as an integer.
 *
 * @return the degree of the array <code>poly</code>.
 */
private int computeDeg() {
    int d;
    for (d = poly.length - 1; d >= 0; d--)
        if (!poly[d].equals(BigInteger.ZERO))
            break;
    return d;
}
}
```

## References

- [BSS99] I. F. Blake, G. Seroussi, and N. P. Smart. *Elliptic curves in cryptography*, volume 265 of *London Mathematical Society Lecture Note Series*. Cambridge University Press, Cambridge, 1999.
- [Car04] Robert Carls. *A Generalized Arithmetic Geometric Mean*. PhD thesis, Rijksuniversiteit Groningen, November 2004.
- [CM94] Jean-Marc Couveignes and François Morain. Schoof’s algorithm and isogeny cycles. In *Algorithmic number theory (Ithaca, NY, 1994)*, volume 877 of *Lecture Notes in Comput. Sci.*, pages 43–58. Springer, Berlin, 1994.
- [Deu41] Max Deuring. Die Typen der Multiplikatorenringe elliptischer Funktionenkörper. *Abh. Math. Sem. Hansischen Univ.*, 14:197–272, 1941.
- [Elk98] Noam D. Elkies. Elliptic and modular curves over finite fields and related computational issues. In *Computational perspectives on number theory (Chicago, IL, 1995)*, volume 7 of *AMS/IP Stud. Adv. Math.*, pages 21–76. Amer. Math. Soc., Providence, RI, 1998.
- [FGH00] Mireille Fouquet, Pierrick Gaudry, and Robert Harley. An extension of Satoh’s algorithm and its implementation. *J. Ramanujan Math. Soc.*, 15(4):281–318, 2000.
- [FGH01] Mireille Fouquet, Pierrick Gaudry, and Robert Harley. Finding secure curves with the Satoh-FGH algorithm and an early-abort strategy. In *Advances in cryptology—EUROCRYPT 2001 (Innsbruck)*, volume 2045 of *Lecture Notes in Comput. Sci.*, pages 14–29. Springer, Berlin, 2001.
- [FM02] Mireille Fouquet and François Morain. Isogeny volcanoes and the SEA algorithm. In *Algorithmic number theory (Sydney, 2002)*, volume 2369 of *Lecture Notes in Comput. Sci.*, pages 276–291. Springer, Berlin, 2002.
- [Gau02] Pierrick Gaudry. A comparison and a combination of SST and AGM algorithms for counting points of elliptic curves in characteristic 2. In *Advances in cryptology—ASIACRYPT 2002*, volume 2501 of *Lecture Notes in Comput. Sci.*, pages 311–327. Springer, Berlin, 2002.
- [Har02] Robert Harley. Asymptotically optimal point counting, December 2002. email to NMBRTHRY list.
- [Hus87] Dale Husemöller. *Elliptic curves*, volume 111 of *Graduate Texts in Mathematics*. Springer-Verlag, New York, 1987.

- [KPC<sup>+</sup>02] Hae Young Kim, Jung Youl Park, Jung Hee Cheon, Je Hong Park, Jae Heon Kim, and Sang Geun Hahn. Fast elliptic curve point counting using Gaussian normal basis. In *Algorithmic number theory (Sydney, 2002)*, volume 2369 of *Lecture Notes in Comput. Sci.*, pages 292–307. Springer, Berlin, 2002.
- [Lan02] Serge Lang. *Algebra*, volume 211 of *Graduate Texts in Mathematics*. Springer-Verlag, New York, third edition, 2002.
- [Ler97] R. Lercier. Algorithmique des courbes elliptiques dans les corps finis, June 1997.
- [LL03] Reynald Lercier and David Lubicz. Counting points in elliptic curves over finite fields of small characteristic in quasi quadratic time. In *Advances in cryptology—EUROCRYPT 2003*, volume 2656 of *Lecture Notes in Comput. Sci.*, pages 360–373. Springer, Berlin, 2003.
- [Mes72] William Messing. *The crystals associated to Barsotti-Tate groups: with applications to abelian schemes*. Springer-Verlag, Berlin, 1972. Lecture Notes in Mathematics, Vol. 264.
- [Mes00] Jean-Francois Mestre. Lettre adressée a Gaudry et Harley, December 2000. Available at <http://www.math.jussieu.fr/~mestre>.
- [Pil90] Jonathan Pila. Frobenius maps of abelian varieties and finding roots of unity in finite fields. *Mathematics of Computation*, 55(192):745–763, 1990.
- [Sat00] Takakazu Satoh. The canonical lift of an ordinary elliptic curve over a finite field and its point counting. *Journal of the Ramanujan Mathematical Society*, 15(4):247–270, 2000.
- [Sch95] René Schoof. Counting points on elliptic curves over finite fields. *Journal de Théorie des Nombres de Bordeaux*, 7(1):219–254, 1995. Les Dix-huitièmes Journées Arithmétiques (Bordeaux, 1993).
- [Ser79] Jean-Pierre Serre. *Local fields*, volume 67 of *Graduate Texts in Mathematics*. Springer-Verlag, New York, 1979.
- [Sil92] Joseph H. Silverman. *The arithmetic of elliptic curves*, volume 106 of *Graduate Texts in Mathematics*. Springer-Verlag, New York, 1992. Corrected reprint of the 1986 original.
- [Sil94] Joseph H. Silverman. *Advanced topics in the arithmetic of elliptic curves*, volume 151 of *Graduate Texts in Mathematics*. Springer-Verlag, New York, 1994.
- [Skj03] Berit Skjærnaa. Satoh’s algorithm in characteristic 2. *Mathematics of Computation*, 72(241):477–487 (electronic), 2003.

- [SST03] Takakazu Satoh, Berit Skjerna, and Yuichiro Taguchi. Fast computation of canonical lifts of elliptic curves and its application to point counting. *Finite Fields and their Applications*, 9(1):89–101, 2003.
- [Vél71] Jacques Vélou. Isogénies entre courbes elliptiques. *C. R. Acad. Sci. Paris Sér. A-B*, 273:A238–A241, 1971.
- [Ver03] Frederik Vercauteren. *Computing Zeta Functions of Curves over Finite Fields*. PhD thesis, Katholieke Universiteit Leuven, November 2003.
- [VPV01] Frederik Vercauteren, Bart Preneel, and Joos Vandewalle. A memory efficient version of Satoh’s algorithm. In *Advances in cryptology—EUROCRYPT 2001 (Innsbruck)*, volume 2045 of *Lecture Notes in Comput. Sci.*, pages 1–13. Springer, Berlin, 2001.