Technical University of Denmark

David Møller Hansen

Pairing-based Cryptography

A short signature scheme using the Weil pairing

MSc Master's Thesis, February 2009

DTU Mathematics
Department of Mathematics

Pairing-based Cryptography - A short signature scheme using the Weil pairing

This report was prepared by
David Møller Hansen

Supervisors
Lars Ramkilde Knudsen
Peter Beelen

Department of Mathematics
Technical University of Denmark
Matematiktorvet building 303S
DK-2800 Kgs. Lyngby
Denmark

www.mat.dtu.dk
Tel: (+45) 45 25 30 31
Fax: (+45) 45 88 13 99
E-mail: instadm@mat.dtu.dk

Release date:	February 27, 2009
Category:	1 (public)
Edition:	First
Comments:	This report is part of the requirements to achieve the Master of Science in Mathematical Modelling and Computation (M.Sc.Techn) at the Technical University of Denmark. This report represents 35 ECTS points.

Preface

The contents of the following pages are the result of my increasing interests in cryptography from my final year in high school up till now. The seed for this thesis was placed when I did my highschool graduate paper on the RSA crypto system.

In the spring of 2007 I attendended the cryptology 2 course lectured by one of my thesis advisors Lars Ramkilde Knudsen. I was exposed to Stinson's very comprehensive book and while writing my thesis, I discovered that during the course I had circled the following on page 262:

...there is a method of exploiting an explicit isomorphism between elliptic curves and finite fields that leads to efficient algorithms for certain classes of elliptic curves.

In the fall the same year I attendended a course in applied cryptography lectured by Erik Zenner, who mentioned Pairing-based cryptography. Erik adviced me to talk to Lars. Lars brought Peter Beelen onboard as a co-advisor and presented the very well written article on a short signature scheme by Boneh et al., which this thesis has come to be based upon.

I would like to thank the entire staff at the department of mathematics at DTU for making it such a pleasent place to work on my thesis on a day-to-day basis. I want to give special thanks to my thesis advisors Lars Ramkilde Knudsen and Peter Beelen.

Lars, thank you for all your time and for keeping me focused in the process. Peter, thank you for patiently helping me through a lot of mathematics I had forgotten I knew or not knew at all.

February 27, 2009

David Møller Hansen

Summary

This thesis investigates the BLS short signature scheme from elliptic curve groups using the Weil pairing. Using co-GDH groups, the signature scheme is proved secure in the random oracle model. The Weil pairing is constructed theoretically and implemented in Sage using Miller's algorithm. A reduction of the discrete logarithm problem on an elliptic curve group to the discrete logarithm problem in a finite extension field is derived as a consequence of the Weil pairing. The reduction is showed effective on supersingular elliptic curves over fields of low characteristic. Co-GDH groups is constructed from supersingular elliptic curves and consequences of this is discussed.

The main conclusion is that one should not use supersingular elliptic curves for constructing the co-GDH groups to be used for generating short signatures. The security of the signature scheme will in this case rely on the discrete logarithm problem in a finite extension field and not on the elliptic curve group. This results in signatures of length not much shorter than the length of the equivalent ECDSA signature, which defeats the purpose of using pairings. A sub conclusion of this is that finding elliptic curves that make good candidates for constructing co-GDH groups is a non-trivial task.

Keywords: Cryptography. Elliptic curves. Pairing-based cryptography. Short signature scheme. Weil pairing. MOV reduction. Supersingular elliptic curves.

Dansk Resumé

Dette speciale undersøger BLS metoden til at opnå korte signaturer fra elliptiske kurvegrupper ved brug af Weil pairingen. Ved benyttelse af co-GDH grupper bevises signaturmetoden sikker under random oracle modellen. Weil pairingen konstrueres teoretisk og implementeres i Sage ved at benytte Millers algoritme. En reduktion af det diskrete-logaritme-problem på en elliptisk kurve til det diskrete logaritme problem i et endeligt udvidelseslegeme udledes som en konsekvens af Weil paringen. Reduktionen vises effektiv for supersingulære elliptiske kurver over endelige legemer af lav karakteristik. Co-GDH grupper konstrueres fra supersingulære elliptiske kurver og konsekvenserne af dette diskuteres.

Hovedkonklusionen er, at man ikke bør benytte supersingulære elliptiske kurver til at konstruere co-GDH grupper, som skal benyttes til frembringelse af korte signaturer. Sikkerheden af signatursystemet vil i så fald afhænge af det diskrete logaritme problem i et endeligt udvidelseslegeme og ikke på den elliptiske kurve. Dette resulterer i signaturer med en længde ikke meget kortere end længden af den ækvivalente ECDSA signatur. Dermed ødelægges formålet med at benytte pairings. En delkonklusion af dette er at det er en ikke-triviel opgave at finde elliptiske kurver, som udgør gode kandidater til konstruktion af co-GDH grupper.

Contents

List of Figures x

List of Tables xi

List of Algorithms xiii

1 Introduction 1

1.1 Gap Diffie-Hellman problem 3

1.2 Elliptic curve groups 5

2 The BLS signature scheme 11

2.1 Description of the BLS signature scheme 11

2.2 The MapToGroup hash function 13

2.2.1 Implementation of MapToGroup 16

2.2.2 Security of MapToGroup 17

2.3 Security of the BLS signature scheme 20

3 The Weil pairing 27

3.1 Divisor theory 28

3.2 Constructing the Weil pairing 34

3.3 Properties of the Weil pairing 36

3.4 Calculating the Weil pairing 41

3.4.1 Implementation of the Weil pairing 47

4 The Menezes, Okamoto, Vanstone reduction **49**

4.1 Supersingular elliptic curves 50

4.2 Embedding of points 53

4.3 Reduction in the supersingular curve case 54

5 co-GDH groups from the Weil pairing **57**

5.1 Efficiently computable group isomorphism 58

5.2 Tractability of DDH problem 60

5.3 Intractability of CDH problem 60

5.3.1 Generic discrete logarithm algorithms 61

5.3.2 The Index Calculus method 64

5.3.3 A small experiment 67

5.3.4 Lower bounds on curve parameters 72

6 BLS scheme using the Weil Pairing **75**

6.1 BLS with elliptic curve groups 75

6.1.1 Implementation of the BLS scheme 77

6.2 Selecting an appropriate curve 79

6.2.1 Scalability in general 82

6.2.2 Performance 82

7 Conclusion **85**

References **87**

Appendix **90**

A Sage **91**

B Projective geometry **95**

C Another example 97

D Supersingular curves 99

E BLS Signature System Guide 101

E.1 Installation 101
E.2 Weil pairing function 102
E.3 MapToGroup function 102
E.4 BLSSignatureScheme class 103
E.4.1 Parameters 103
E.4.2 Functions 103
E.4.3 BLS outside Sage - almost 104
E.4.4 Attached examples 105

F Code 107

F.1 Sage interact: Point addition on elliptic curve 107
F.2 Sage patch: Map to group 110
F.3 Sage patch: Weil pairing 113
F.4 Sage sample: Weil pairing example 118
F.5 Sage sample: MNT curve 118
F.6 Sage sample: MOV reduction example 119
F.7 Magma script: Timing of logarithm computations 120
F.8 Sage plot: Plot of time complexity for logarithm computations 123
F.9 Sage patch: BLS signature scheme 125
F.10 Sage sample: BLS signature example 131
F.11 Sage script: BLS CLI 132
F.12 Sage interact: Weil Optimisations 134

List of Figures

1.1 The curve E_a has a singularity, E_b an intersection, while E_c is non-singular. 5

1.2 The curve $E_c : y^2 = x^3 - x$ over prime field $\mathbb{F}_{101}$. 6

1.3 Addition of points P, Q and R on curve $E/\mathbb{R} : y^2 = x^3 - 2x$. 7

5.1 Shanks' baby-step giant-step algorithm graphically 61

5.2 Pollard's rho method graphically 64

5.3 Plot of CPU timing results for curve group $E_{2,1}$. 71

5.4 Plot of CPU timing results for curve group $E_{2,2}$. 71

5.5 Log-plot of t_{rho} and t_{IC} wrt. to the base field extension degree m and elliptic curves $E_{2,1}$ and $E_{2,2}$ 72

A.1 Sage interact: adding points on an elliptic curve graphically. . 94

List of Tables

3.1 Timing of Weil pairing for different sized subgroups of elliptic curve group $E_{3,2}(3^{42}) : y^2 = x^3 + x + 2$. 47

5.1 Time complexity for discrete logarithm algorithms measured in group size n or finite field size q 66

5.2 Magma MOV reduction cpu(s) timings in curve $E_{2,1}(\mathbb{F}_{2^m})$. . 69

5.3 Magma MOV reduction cpu(s) timings in curve $E_{2,2}(\mathbb{F}_{2^m})$. . 70

6.1 Timing (s) of BLS implementation in Sage for different curves 79

6.2 Bitsizes of supersingular curve groups $E_{3,2}(\mathbb{F}_{3^m})$ and $E_{3,2}(\mathbb{F}_{3^m})$. 80

6.3 Security properties of candidate curves. 81

6.4 82 bit security comparisson of BLS and ECDSA 82

6.5 Comparison of signing and verification times (in ms) on a PIII 1 GHz. [BKLS02, Table 4] 83

D.1 Structure in supersingular curves 100

List of Algorithms

2.1.1 KeyGen 12

2.1.2 Sign 12

2.1.3 Verify 12

2.2.1 MapToGroup 13

2.2.2 UpdateTable 18

2.3.1 SimulateSignatureOracle 21

2.3.2 UpdateHList 23

3.4.1 Millers algorithm using double-and-add 45

4.3.1 MOV reduction for supersingular curves 55

6.1.1 ECKeyGen 76

6.1.2 ECSign 76

6.1.3 ECVerify 76

Chapter 1

Introduction

Several modern asymmetric cryptographic schemes build on the discrete logarithm problem in finite fields. Today there exist sub-exponential methods of solving the discrete logarithm problem in finite fields. This has made elliptic curve groups appealing since these sub-exponential methods do not apply here. This makes it possible to keep group sizes smaller and as a result of that, we can use smaller keys while still keeping the same bit security.

Besides encrypting data with asymmetric cryptography the paradigm also provides the possibility of signing data. In applications where data bandwidth is expensive, we would like the length of a signature to be as short as possible while maintaining a required bit security. The current elliptic curve based standard for digital signatures ECDSA does not provide any shorter signature lengths than the non-elliptic curve based standard DSA using prime fields. The DSA signature consists of two field elements of each size q, i.e. a signature length $2q$. The equally secure ECDSA signature consists of one point coordinate of size q and an extra value of size q and thus also a signature length of size $2q$.

Is it possible to do better?

Yes it is. Boneh, Lynn and Schacham [BLS04] propose a signature scheme using a special pair of groups called gap groups. The groups they use for gap groups are elliptic curve groups and they show, that by choosing curves *wisely* you can get the same bit security on a signature with only length q. Elliptic curve groups only work as gap groups because we are able to define a bilinear map on elliptic curve groups, one such map is called the

Weil pairing.

In this thesis I will take a practical approach on constructing the BLS short signature scheme by using the Sage open source mathematical software package [Ste09] for examples and implementations. I will investigate how to choose the elliptic curve *wisely* by choosing my elliptic curves unwisely and show what the consequences of this choice is.

The BLS scheme requires a hash function to map the data into an element of the one gap group. This can be done on elliptic curve groups by constructing a hash function from a random oracle and prove that security is not compromised. The hash function will be implemented in Sage.

Given gap groups, we get the BLS signature scheme and prove it is in the random oracle model.

We will construct the Weil pairing and show that it is a bilinear map on an elliptic curve. We show how to compute the pairing efficiently using Miller's algorithm and implement the algorithm in Sage.

An application of the Weil pairing is the Menezes, Okamoto and Vanstone (MOV) reduction of the discrete logarithm problem in a curve group to a finite field. We perform this reduction and show it is effective on the elliptic curve groups we look at.

We will then show that given the Weil pairing you can use elliptic curve groups as gap groups. I will do a small experiment in Magma with supersingular curves to see the consequences of the MOV reduction when using elliptic curve groups for gap groups.

Finally we will construct the BLS scheme using elliptic curve groups and the Weil pairing. The system is implemented in Sage. We then choose a supersingular curve such that we get a gap group from it and argue why using supersingular curves is not *wise* to do and discuss how we can do better.

I have attached appendices on Sage syntax and commands, Elliptic curves in projective geometry, Supersingular curve results, A guide to installing and using the included BLS implementation and all code referenced in this thesis.

In the rest of the introduction gap groups and the gap group problem which the signature scheme is build on is introduced along with elliptic curve groups.

1.1 Gap Diffie-Hellman problem

We will in this section define the co-Gap Diffie-Hellman problem from two known problems already widely used in cryptography. We will start by defining the Discrete Logarithm problem and then the regular Diffie-Hellman problems. Asymmetric cryptography builds on different computationally hard problems, such as computing the discrete logarithm of an element in a large group with respect to a generator. We call this the Discrete Logarithm (DLog) Problem. Formally we define it as [Sti05, p.234].

Definition 1.1 (Discrete Logarithm Problem)**.** *Given a group G of order n with a generator g and an element $h \in G$.*

$$\textit{Compute } a \in \mathbb{Z}_n : g^a = h.$$

We now look at some similar problems originally stated and used in the key agreement protocol by Whitfield Diffie and Martin Hellman [DH76]. The first one is the Computational Diffie-Hellman (CDH) Problem.

Definition 1.2 (Computational Diffie-Hellman Problem)**.** *Given a group G of order n with a generator g and two elements g^a and g^b for unknown $a, b \in \mathbb{Z}_n$.*

$$\textit{Compute the element } g^{ab}.$$

The CDH problem can be polynomially reduced to the DLog problem [Sti05, p.273] proving that the DLog problem is at least as hard as CDH problem, i.e. if you can solve the DLog problem efficiently then you can solve the CDH problem efficiently. The other Diffie-Hellman problem is the Decision Diffie-Hellman (DDH) Problem.

Definition 1.3 (Decision Diffie-Hellman Problem)**.** *Given a group G of prime order n with a generator g and three elements g^a, g^b and g^c for unknown $a, b, c \in \mathbb{Z}_n$.*

$$\textit{Decide whether } c \equiv ab \pmod{n}.$$

You can show that the DDH problem can be polynomially reduced to the CDH problem [Sti05, p.273]. While both the CDH and DDH problem are interesting problems already widely used in cryptography we will work with slight variants of the two: The Computational co-Diffie-Hellman (co-CDH) Problem and Decision co-Diffie-Hellman (co-DDH) Problem [BLS04]. These problem instances are defined over a group pair (G_1, G_2) instead of a single group.

Definition 1.4 (Computational co-Diffie-Hellman Problem)**.** *Given a pair of groups (G_1, G_2) of prime order n with generators g_1, g_2 and two elements $h = g_1^b$ and g_2^a for $a, b \in \mathbb{Z}_n$.*

$$\textit{Compute the element } h^a = g_1^{ab}.$$

Definition 1.5 (Decision co-Diffie-Hellman Problem)**.** *Given a group pair (G_1, G_2) of prime order n with generator $g_2 \in G_2$, an element $h \in G_1$, g_2^a and h^d for $a, d \in \mathbb{Z}_n$.*

$$\textit{Decide whether } a \equiv d \pmod{n}.$$

Note when $G_1 = G_2$ then the above co-CDH and co-DDH problems become the CDH and DDH problems defined on a single group. In this case the above definition are equivalent to the normal DDH problem if we write $h = g_1^b$ then we can always write $d = c/b$ for some $b, c \in \mathbb{Z}_n$. The tuple (g_2, g_2^a, h, h^d) is called a co-Diffie-Hellman tuple.

We will need to refer to the hardness of the co-CDH problem later on. A measure of hardness of the co-CDH problem, can be chosen as the probability of solving the problem within a given time frame.

Definition 1.6. *An algorithm $\mathcal{A}$ is said to (τ, ε)-break co-CDH on (G_1, G_2) if the probability of success in time at most τ of $\mathcal{A}$ solving co-CDH on (G_1, G_2) satisfies:*

$$P\left(\mathcal{A}(g_2, g_2^a, h) = h^a : a \stackrel{R}{\leftarrow} \mathbb{Z}_n, h \stackrel{R}{\leftarrow} G_1\right) \geq \varepsilon.$$

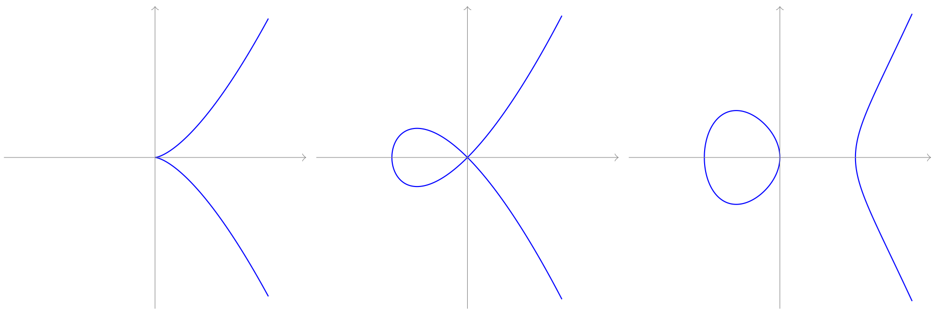
Now we are ready to define the co-Gap Diffie-Hellman (co-GDH) Problem. We first look at Gap Diffie-Hellman group pairs. These group pairs have the special property of the co-DDH problem being easy while the co-CDH problem remains hard.

Definition 1.7 (Gap Diffie-Hellman group pair)**.** *A group pair (G_1, G_2) is said to be a (τ, t, ε)-co-GDH group pair if:*

- *Group operations in G_1 and G_2 and the isomorphism $\psi : G_2 \to G_1$ can be computed in time at most τ.*
- *The co-DDH problem on (G_1, G_2) can be solved in time at most τ.*
- *No algorithm (t, ε)-breaks co-CDH on (G_1, G_2).*

The co-GDH problem thus becomes the problem of solving co-CDH given a co-DDH oracle[1] [OP01]. In the last defining property of co-GDH we will

[1] The co-DDH oracle can be thought of as a machine able to answer co-DDH problem in a single operation.



$E_a : y^2 = x^3$ $E_b : y^2 = x^3 + x^2$ $E_c : y^2 = x^3 - x$

Figure 1.1: The curve E_a has a singularity, E_b an intersection, while E_c is non-singular.

assume that the only way of breaking co-CDH, even given a co-DDH oracle is to solve the DLog problem in some form. This is not proved in any way and there might be another way of solving the co-CDH problem, given a co-DDH oracle without having to solve the DLog problem. We only note this, in the rest of the thesis we will implicitly use the above assumption.

1.2 Elliptic curve groups

In this section elliptic curve groups will be introduced. These are the groups we will use to obtain a co-GDH group pair. We begin by defining an elliptic curve in general.

Definition 1.8. *Define an elliptic curve E over a field K as a non-singular curve given by the general Weierstrass equation*

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

with $a_1, a_2, a_3, a_4, a_6 \in K$.

The requirement of the curve being non-singular ensures that the graph of the curve has no singularities and no self-intersections as the curve E_a in Figure 1.1.

We have defined elliptic curves over arbitrary fields K in general, so also over finite fields e.g. as the prime field $\mathbb{F}_{101}$ in Figure 1.2.

The general Weierstrass form can be reduced to a more compact form. If we distinguish in cases of the characteristic $p = 2$ and $p \neq 2$ of the field K [Kim08]. In this thesis we will only look at curves with $a_1 = 0$ in the case $p = 2$.

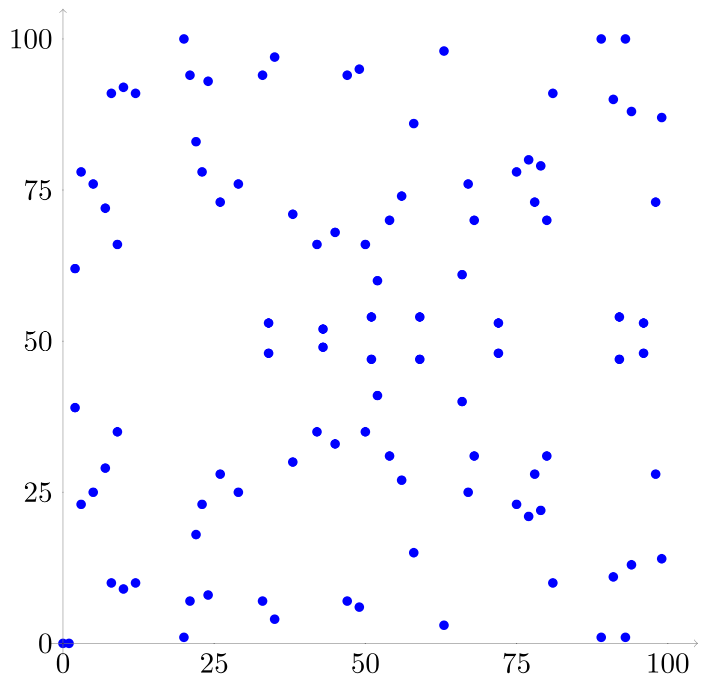


Figure 1.2: The curve $E_c : y^2 = x^3 - x$ over prime field $\mathbb{F}_{101}$.

Theorem 1.9. *If K's characteristic $p = 2$ and $a_1 = 0$ then the general Weierstrass form can be put on the form:*

$$E/K : y^2 + a_3 y = x^3 + ax^2 + bx + c, \;\; a_3 \neq 0 \tag{1.2}$$

with $a_3, a, b, c \in K$.

If K's characteristic $p \neq 2$ then the general Weierstrass form can be put on the form:

$$E/K : y^2 = x^3 + ax^2 + bx + c, \tag{1.3}$$

with $a, b, c \in K$.

Remark 1.10. *Form 1.2 always defines an elliptic curve. The form 1.3 defines an elliptic curve if and only if the polynomial $f(x) = x^3 + ax^2 + bx + c$ has distinct roots.*

If we look at the set of points on E, then we can define a composition of the point set.

Definition 1.11. *Define the composition '+' of two points P and Q on an elliptic curve in the following way. The line intersecting both P and Q will*

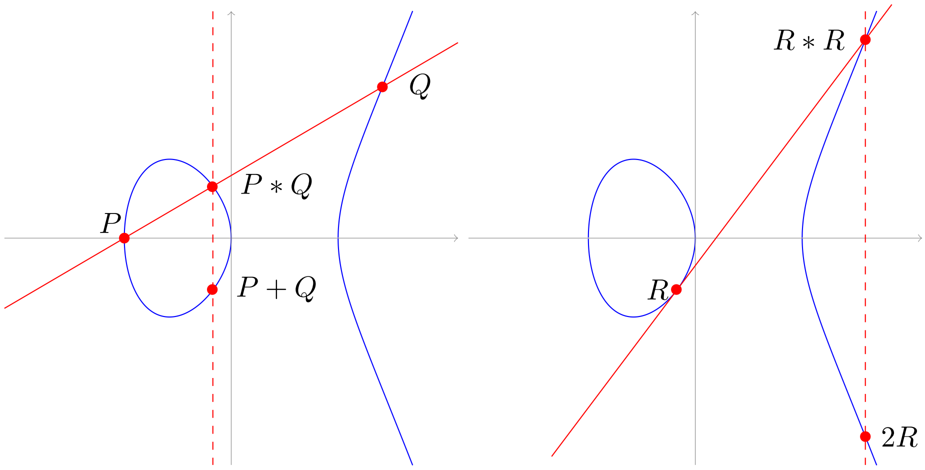


Figure 1.3: Addition of points P, Q and R on curve $E/\mathbb{R} : y^2 = x^3 - 2x$

always intersect the curve in a third point $P*Q$ of the projective plane[2]*. Let $\mathcal{O}$ be the point at infinity*[3] *on the curve, then the composition $P+Q$ is given as*

$$P + Q = (P * Q) * \mathcal{O}.$$

If we choose the field to be $\mathbb{R}$, then the composition can be explained graphically which is depicted in Figure 1.3. Notice how the line intersecting the elliptic curve in the points P and Q, intersects the curve in a third point $P * Q$. The line intersecting $P * Q$ and $\mathcal{O}$ is the vertical dashed line which intersects the curve in the third point $P+Q = (P*Q)*\mathcal{O} = R$. In Appendix F.1 I have appended the source code for a Sage interact with the graphical point addition.

Theorem 1.12. *Points on an elliptic curve E/K form an abelian group with the defined composition '+' and the point at infinity $\mathcal{O}$ as the neutral element and the inverse to a point $P = (x_1, y_1)$ as $-P = (x_1, -y_1 - a_1x_1 - a_3)$.*

The proof of this theorem can be found in several varieties in several textbooks on elliptic curves [ST92], [Was08], so we will not prove it. We will instead in Section 3.1 on divisor theory sketch an alternative way to proving the group law using divisors.

From this point on we will denote the abelian group of points with coordinates over a field extension $K_1 \supseteq K_0$ on the curve E/K_0 as $E(K_1)$. The composition '+' will be referred to as addition and written as $+$. Since

[2] See Appendix B.
[3] The point at infinity is defined as the point $[0 : 1 : 0]$ in projective coordinates.

addition is defined by line intersections, we can provide explicit formulas [Kim08] for adding two points $P = (x_1, y_1)$ and $Q = (x_2, y_2)$, when neither is the point at infinity. Let us look at the following two cases.

Case I: For $Q = -P$ we will have that $Q + P = \mathcal{O}$. Note that in this case $y_2 = y_1$ or $y_2 = -y_1 - a_1x_1 - a_3$. Graphically these are the points that produce vertical lines in the addition process. In Figure 1.3 we have examples of this situation where $P = -P$ and $R = -(P * Q)$.

Case IIa: For $Q \neq -P$ define for $x_1 \neq x_2$

$$\alpha := \frac{y_2 - y_1}{x_2 - x_1} \text{ and } \beta := \frac{y_1x_2 - y_2x_1}{x_2 - x_1}$$

In this case we will have two distinct points, as with P and Q on Figure 1.3. α is the slope of the line and β is the intersection with the y-axis.

Case IIb: For $Q \neq -P$ define for $x_1 = x_2$

$$\alpha := \frac{3x_1^2 + 2a_2x_1 + a_4 - a_1y_1}{2y_1 + a_1x_1 + a_3} \text{ and } \beta := \frac{-x_1^3 + a_4x_1 + 2a_6 - a_3y_1}{2y_1 + a_1x_1 + a_3}.$$

Here the point is the same and you use the tangent in the point instead of the line through two distinct points.

The point $P + Q = (x_3, y_3)$ can be computed in both cases IIa and IIb as

$$x_3 = \alpha^2 + a_1\alpha - a_2 - x_1 - x_2, \; y_3 = -(\alpha + a_1)x_3 - \beta - a_3.$$

Example 1.13. *In this example we will look at the curve shown in Figure 1.3. We recognize two points as*

$$P = (-\sqrt{2}, 0) \text{ and } Q = (2, 2)$$

on the curve. We want to compute $P + Q = (x_3, y_3)$ notice that $P \neq -Q$ and $P \neq Q$ so we are in addition case IIa. We compute the slope α and y-axis intersection β:

$$\alpha = \frac{2}{2 + \sqrt{2}} = 2 - \sqrt{2}, \quad \beta = \frac{2\sqrt{2}}{2 + \sqrt{2}} = 2\sqrt{2} - 2.$$

We can then compute coordinates (x_3, y_3):

$$x_3 = \alpha^2 + \sqrt{2} - 2 = 4 - 3\sqrt{2}$$
$$y_3 = -\alpha x_3 - \beta + 2 = -12 + 8\sqrt{2}.$$

Let $R = P + Q$. We next want to compute the doubling $2R$ we will be in case IIb. We again compute the slope α and y-axis intersection β:

$$\alpha = \frac{3x_3^2 - 2}{2y} = \frac{-3 + 4\sqrt{2}}{2}, \quad \beta = \frac{-x^3 - 2x}{2y} = \frac{12 - 9\sqrt{2}}{2}.$$

We can then compute the coordinates of the doubling $2R = (x_4, y_4)$:

$$x_4 = \alpha^2 - 2x_3 = \frac{9}{4}$$
$$y_4 = -\alpha x_4 - \beta = -\frac{21}{8}.$$

Another example with elliptic curve E_c on Figure 1.1 the can be found in Appendix C. The following theorems concern the abelian group $E(\mathbb{F}_q)$ of points on the elliptic curve E over the finite field $\mathbb{F}_q$ with $q = p^e$ for a prime p and an integer e. First we will state the somewhat famous bound on the number of elliptic curve group elements proved by Helmut Hasse in the 1930's.

Theorem 1.14 (Hasse's bound)**.** *Let E be a curve with points defined over the finite field $\mathbb{F}_q$ then the order of $E(\mathbb{F}_q)$ is bounded in the following way*

$$||E(\mathbb{F}_q)| - (q+1)| \leq 2\sqrt{q}.$$

For a proof see Washington [Was08, p.100]. The theorem states that over a finite field $\mathbb{F}_q$ the number of points on the curve does not stray more than two times the squareroot of q. This can be in bitresrepresentation be seen as a single bit. We can even say something about the structure of the elliptic curve group.

Theorem 1.15. *Let E be a curve with points defined over the finite field $\mathbb{F}_q$ then*

$$E(\mathbb{F}_q) \simeq \mathbb{Z}_{n_1} \times \mathbb{Z}_{n_2},$$

for natural numbers $n_1, n_2 \in \mathbb{N}$ with $n_1 | n_2$.

This theorem tells us that an elliptic curve group over a finite field is isomorphic to a cyclic group or a product of cyclic groups. Next we define the n-torsion group of a curve to be the group containing all points that have order n.

Definition 1.16. *Let E/K be an elliptic curve defined over a field K. Define the n'th torsion of E as the set $E[n]$ of points in the algebraic closure of K:*

$$E[n] = \{P \in E(\bar{K}) \mid nP = \mathcal{O}\}.$$

Note that it's only since we are in the algebraic closure $\bar{K}$ we can be sure to have all points of order n. If the n-torsion points is in a smaller field K' than the algebraic closure of K we will call it $E(K')[n]$, else it will implicitly be in $\bar{K}$. We will later on see how we can choose the field and curve such that we may restrict this for practicality. The last theorem and corollary in this section tells us what kind of group structures we get from the set of n-torsion points.

Theorem 1.17. *Let E/K be an elliptic curve over a field K and let $n > 0$. If the characteristic $p = 0$ or $p \nmid n$ then*

$$E[n] \simeq \mathbb{Z}_n \times \mathbb{Z}_n$$

else you can write $n = p^r n'$ such that $p \nmid n'$ and then

$$E[n] \simeq \mathbb{Z}_{n'} \times \mathbb{Z}_{n'} \text{ or } E[n] \simeq \mathbb{Z}_n \times \mathbb{Z}_{n'}.$$

Proof for the above theorem can be found in Washington [Was08, p.81]. We will need the following corollary later on, which states that if we choose an extension field large enough then we can be sure to obtain all points of order n.

Corollary 1.18. *Let E be an elliptic curve with points over a finite field $\mathbb{F}_q$. Let $n \mid |E(\mathbb{F}_q)|$ then there exists an extension degree r for which*

$$E(\mathbb{F}_{q^r})[n] \simeq \mathbb{Z}_n \times \mathbb{Z}_n$$

Proof for the above corollary can be found in Silverman [Sil86, p.89].

Chapter 2

The BLS signature scheme

In this section the Boneh, Lynn, Shacham short signature scheme will be described and security proofs from the original article by Boneh et al. [BLS04] will be worked through. In the article the authors only look at the case where the base field characteristic is strictly greater than two when they construct a hash function onto an elliptic curve group. The characteristic two case is nevertheless an important case, since practical implementations often will take advantage of computers being able to do fast finite field arithmetic over a binary base field. We will in the following treat this case to some extent.

2.1 Description of the BLS signature scheme

The BLS signature scheme is described as follows:

Let (G_1, G_2) be a (τ, t, ε)-co-GDH group pair with group orders equal to n. The signature scheme is then given as the set of algorithms

$$\{KeyGen, Sign, Verify\}.$$

Algorithm 2.1.1 generates an asymmetric key pair $(x, v) \in \mathbb{Z}_n \times G_n$ with private key x and public key v.

Algorithm 2.1.2 is used when signing a message M with the private key x. This algorithm requires a hash function H that can hash the message to an element $h \in G_1$. We will assume that H is a random oracle hash function. We will in Section 2.2 describe this hash function in detail for the case where G_1 and G_2 are elliptic curve groups.

Algorithm 2.1.1: KeyGen

Data: generator g_2 for G_2, prime number p
Result: private key $x \in \mathbb{Z}_n$, public key $v \in G_2$
Choose random $x \in \mathbb{Z}_n$
$v \leftarrow g_2^x$
return (x, v)

Algorithm 2.1.2: Sign

Data: private key $x \in \mathbb{Z}_n$, message $M \in \{0,1\}^*$
Result: signature $\sigma \in G_1$
$h \leftarrow H(M) \in G_1$
$\sigma \leftarrow h^x$
return σ

We check that a message M signed using the public key v has a valid signature σ using Algorithm 2.1.3. We again use the hash function H to hash the message to an element of G_1.

Theorem 2.1. *The signature scheme $\{KeyGen, Sign, Verify\}$ is well defined.*

Proof. We check that a message M signed with Algorithm 2.1.2 using the public key v can be validated with Algorithm 2.1.3 using the private key x where v and x are the key pair generated in Algorithm 2.1.1. Let the key pair (v, x) be generated as described with parameters $\{g_2, n\}$. Let σ_M be the signature produced on message M using the private key x. Let the message hash $H(M) = h$. Then the tuple

$$(g_2, v, h, \sigma_M) = (g_2, g_2^x, h, h^x),\ \ g_2 \in G_2,\ \ h \in G_1$$

is a valid co-Diffie-Hellman tuple by Definition 1.5. □

Algorithm 2.1.3: Verify

Data: public key $v \in G_2$, message $M \in \{0,1\}^*$, signature $\sigma \in G_1$
Result: boolean value
$h \leftarrow H(M) \in G_1$
return `Test`((g_2, v, h, σ) *is a valid co-Diffie-Hellman tuple*)

2.2 The MapToGroup hash function

Later when we want to use elliptic curve groups as our co-GDH group we will need a way of hashing onto an elliptic curve subgroup G_1. We want to do this without it compromising the security of the signature scheme, for this purpose we construct the MapToGroup hash function.

We will construct a more general MapToGroup hash function than the one described by Boneh et al.[BLS04] since the one they give only holds for elliptic curves of Form 1.3, i.e. elliptic curves over fields of characteristic $\neq$ 2.

Algorithm 2.2.1: MapToGroup

Data: message $M \in \{0,1\}^*$, hash function H', parameter I, curve $f(x,y)=0$
Result: $P_M \in G_1$ or Failure
$i \leftarrow 0$
while $i \leq 2^I$ **do**
  $(x_0, b) \leftarrow H'(i||M) \in \mathbb{F}_q \times \{0,1\}$
  if $f(x_0, y) = 0$ *has solutions* (y_0, y_1) **then**
    Let y_0, y_1 be indexed such that $y_1 \geq y_0$
    $\tilde{P}_M \leftarrow (x_0, y_b)$
    $P_M \leftarrow (m/n)\tilde{P}_M \in G_1$
    if $P_M \neq \mathcal{O}$ **then**
      return P_M
  else
    $i \leftarrow i+1$
return Failure: M is unhashable

We will next look at the cases where there are solutions to the equation $f(x_0, y) = 0$ in Algorithm 2.2.1. Note that I've written the elliptic curve E as $f(x,y) = 0$, this should not be confused with the right hand side of the short Weierstrass forms which I will write as $f(x)$. In the following $QR(\mathbb{F}_q)$ will denote the set of quadratic residues in $\mathbb{F}_q$.

Theorem 2.2. *For an elliptic curve $E : f(x,y) = 0$ over a field $\mathbb{F}_q$ of characteristic $p \neq 2$ the equation $f(x_0, y) = 0$ has solutions if and only if $f(x_0) \in QR(\mathbb{F}_q)$. The solutions are*

$$y_0 = -\sqrt{f(x_0)} \text{ and } y_1 = \sqrt{f(x_0)}.$$

Proof. For characteristic $p \neq 2$ we can write the curve $E : f(x,y) = 0$ on the reduced Form 1.3: $y^2 = f(x) = x^3 + ax^2 + bx + c$, and check for solutions to $f(x_0, y) = 0$ by checking if $f(x_0)$ is a quadratic residue. □

To check for solutions in the case of characteristic $p = 2$ we will need the trace map.

Definition 2.3 (Trace)**.** *Let all $\sigma \in Gal(\mathbb{F}_{p^e}/\mathbb{F}_p)$ be indexed $\sigma_i(x) = x^{p^i}$ for $i = 0, \ldots, e-1$. Let $x \in \mathbb{F}_{p^e}$ and define the trace*

$$tr : \mathbb{F}_{p^e} \to \mathbb{F}_p, \text{ where } x \mapsto \sum_{i=0,\ldots,e-1} \sigma_i(x).$$

We prove that over over characteristic $p = 2$ fields the trace maps to 1 and 2 with equal probability.

Lemma 2.4. *The trace $tr(\theta) = 1$ with probability $\frac{1}{2}$ for a randomly chosen $\theta \in \mathbb{F}_{2^e}$.*

Proof. The image of trace of a $\theta \in \mathbb{F}_{2^e}$ is $Im(tr) = \mathbb{F}_2$ so we get the two possibilities

$$tr(\theta) = 0 \Leftrightarrow \theta \text{ is a solution to } x + \ldots + x^{2^{e-1}} = 0$$
$$tr(\theta) = 1 \Leftrightarrow \theta \text{ is a solution to } x + \ldots + x^{2^{e-1}} = 1.$$

The number of possible solutions is in both cases less than or equal to the degree 2^{e-1}, but since the collective number of solutions has to sum to 2^e, we must require equality in both cases thus making the probability

$$P\left(tr(\theta) = 0\right) = \frac{1}{2}$$

for a randomly chosen element $\theta \in \mathbb{F}_{2^e}$. □

For characteristic $p = 2$ we will only look at curves which in the general Weierstrass form have $a_1 = 0$. In this case we can determine if there is a solution to the equation $f(x_0, y)$ by using the following lemma.

Lemma 2.5 (Beelen's lemma)**.** *The equation $y^2 + y = f(x)$ has a solution (x, y) over $\mathbb{F}_{2^e}$ if and only if $tr(f(x)) = 0$.*

Proof. If there exists a solution over $\mathbb{F}_{2^e}$, then

$$\begin{aligned} tr(f(x)) &= tr(y^2 + y) \\ &= (y^2 + y) + \ldots + (y^2 + y)^{2^{e-1}} \\ &= y + y^{2^{e-1}} \\ &= 0. \quad (\text{since } y \in \mathbb{F}_{2^e}) \end{aligned}$$

If the trace $tr(f(x)) = 0$, then choose an element $\theta \in \mathbb{F}_{2^e}$ such that the trace $tr(\theta) = 1$. We can do this since half the elements has trace 1 by lemma 2.4. Now choose

$$y = f(x)\theta^2 + \left(f(x) + f(x)^2\right)\theta^4 + \ldots + \left(f(x) + \ldots + f(x)^{2^{e-2}}\right)\theta^{2^{e-1}}.$$

Notice that when squaring the freshman's dream apply since we're in characteristic 2 and we get:

$$y^2 = f(x)^2\theta^4 + \left(f(x)^2 + f(x)^4\right)\theta^8 + \ldots + \left(f(x)^2 + \ldots + f(x)^{2^{e-1}}\right)\theta.$$

Then plug y into the equation and check that the above is in fact a solution.

$$\begin{aligned}
y^2 + y &= f(x)\left(\theta^2 + \ldots + \theta^{2^{e-1}}\right) + \left(f(x)^2 + \ldots + f(x)^{2^{e-1}}\right)\theta \\
&= f(x)\left(\theta + \theta^2 + \ldots + \theta^{2^{e-1}}\right) + \left(f(x) + f(x)^2 + \ldots + f(x)^{2^{e-1}}\right)\theta \\
&= f(x)tr(\theta) + tr(f(x))\theta \\
&= f(x).
\end{aligned}$$

The idea of the proof was taken from Hilbert 90, additive form [Lan93, p.290]. □

Theorem 2.6. *For an elliptic curve $E : f(x_0, y) = 0$ over the finite field $\mathbb{F}_{2^e}$, the equation $f(x, y) = 0$ has solutions if and only if $tr(f(x_0)) = 0$. The solutions are:*

$$y_0 \text{ and } y_1 = y_0 + 1,$$

where

$$y_0 = f(x_0)\theta^2 + \left(f(x_0) + f(x_0)^2\right)\theta^4 + \ldots + \left(f(x_0) + \ldots + f(x_0)^{2^{e-2}}\right)\theta^{2^{e-1}}$$

an element $\theta \in \mathbb{F}_{2^e}$ such that the trace $tr(\theta) = 1$.

Proof. Assume that $a_1 = 0$ and $a_3 = 1$ in the general Weierstrass form of the curve. We may then write the curve on the Form 1.2:

$$E/\mathbb{F}_{2^e} : y^2 + y = f(x) = x^3 + ax^2 + bx + c$$

By Lemma 2.5 we have that there exists a solution to the equation $f(x_0, y) = 0$ if and only if $tr(f(x_0)) = 0$. If we choose a random $\theta \in \mathbb{F}_{2^e}$ we will with probability $\frac{1}{2}$ have that $tr(\theta) = 1$ and then by the proof of the lemma

$$y_0 = f(x_0)\theta^2 + \left(f(x_0) + f(x_0)^2\right)\theta^4 + \ldots + \left(f(x_0) + \ldots + f(x_0)^{2^{e-2}}\right)\theta^{2^{e-1}}.$$

The other solution is $y_1 = y_0 + 1$, since if you plug y_1 into the left hand side equation and use the freshman's dream you see that:

$$(y_0 + 1)^2 + (y_0 + 1) = y_0^2 + y_0.$$

□

2.2.1 Implementation of MapToGroup

The MapToGroup function has been implemented in sage on the `EllipticCurve_finite_field` curve class. So it can be called from here.

Example 2.7 (MapToGroup)**.** *This short example is included to demonstrate the function of MapToGroup in Sage. To simplify it we just map into a point of order equal to the elliptic curve group order.*

```
sage: E2=EllipticCurve(GF(2^7,'a'),[0,0,1,1,1])
sage: E2
Elliptic Curve defined by y^2 + y = x^3 + x +1 over Finite
Field in a of size 2^7
sage: m=E2.cardinality()
sage: P=E2.map_to_group(m,m,'test',17)
sage: P
(a^6 + a^5 + a^4 + a^3 + a^2 + a + 1 : a^4 + a^3 + a^2 + a : 1)
sage: P in E2
True
sage: Q=E2.map_to_group(m,m,'test',13)
sage: P==Q
True
```

Notice that the parameter I can be set to both 13 or 17 and we will still get the same point. This is because the parameter only controls how many times the algorithm should keep trying to find points with solutions of right order. When the first point is found the algorithm returns. So if P did not equal Q in the above, then MapToGroup had to have failed. Which would have raised a warning in Sage and then Q would never have been assigned the point object.

The MapToGroup implementation uses Python's hashlib library to do the initial SHA-1 hash that returns 160 bits. We take the first bit away, save it, and then use what we need of the remaining 159 bits. What we need is essentially the lowest number of bits to represent every element in $\mathbb{F}_q$. Thus, in the current implementation with SHA-1, $\mathbb{F}_q$ should not be larger than 159 bits because otherwise we do not hit every element. The rationale for throwing away bits is to keep the distribution of the probability that an element hit is uniform.

The representation of the element in $\mathbb{F}_q$ is done by translating from base-2 to base p, where p is the characteristic of $\mathbb{F}_q$. We then use the base-p representation to represent coefficients of an element in $\mathbb{F}_q$. This is fast when p is low, as it will be in our case. We will need about $\log q$ bits to hit every element in $\mathbb{F}_q$, so again this implementation is limited in what size fields $\mathbb{F}_q$ it can handle. The implementation can be inspected in Appendix F.2

2.2.2 Security of MapToGroup

When discussing the security of the signature scheme we want to work in the random oracle security model, and assume we have access to a random oracle hash function

$$H' : \{0,1\}^* \to \mathbb{F}_q \times \{0,1\}.$$

We need to show that it is enough to have this random oracle hash function $H' : \{0,1\}^* \to \mathbb{F}_q \times \{0,1\}$. This is important since we have seen that we can build this from existing cryptographic hash functions.

When we're working in elliptic curve groups we showed that we can use the constructed hash function MapToGroup. So First we need to prove that the signature scheme will still be secure if we use our constructed hash function mapping onto a subgroup of $E(\mathbb{F}_q)$. First we need to define what we mean when we say *secure*.

Definition 2.8. *A signature scheme is $(t, q_H, q_S, \varepsilon)$-existentially unforgeable under an adaptive chosen-message attack if no attacker can $(t, q_H, q_S, \varepsilon)$-break it. The attacker $(t, q_H, q_S, \varepsilon)$-break the signature scheme if he wins the following game in time t with probability at least ε only using q_H hash function queries:*

1. *The challenger gives the attacker a valid public key.*
2. *The attacker can adaptively request at most q_S signatures σ_i created from the private key and messages M_i of his choice.*
3. *The attacker outputs a signature pair M, σ and wins if $M \notin \{M_i, i = 1, \ldots, q_S\}$ and σ is a valid signature under the public key.*

We will show that using the MapToGroup hash function do not compromise the security of our signature scheme, by showing that the security parameters when using MapToGroup can be controlled.

Theorem 2.9. *Let $E/\mathbb{F}_q$ be an elliptic curve and let $|E(\mathbb{F}_q)| = m$. Let G_1 be a subgroup of $E(\mathbb{F}_q)$ with order n such that $n^2 \nmid m$. Suppose the co-GDH signature scheme is $(t, q_H, q_S, \varepsilon)$-secure on (G_1, G_2) when a random hash function*

$$H : \{0,1\}^* \to G_1$$

is used. Then it is $(t - 2^I C_{G_1}(q_H + q_S + 1), q_H - q_S - 1, q_S, \varepsilon)$*-secure when the hash function used is computed with the MapToGroup algorithm 2.2.1 that uses* H' *which is a random oracle hash function*

$$H' : \{0,1\}^* \to \mathbb{F}_q \times \{0,1\}.$$

C_{G_1} *is the constant time it takes to do an exponentiation in* G_1 *and* I *is a stop parameter in the Algorithm 2.2.1.*

Proof. We prove the negated expression: Let the hash function used in the game described in Definition 2.8 be the one in Algorithm 2.2.1 with the random oracle hash function

$$H' : \{0,1\}^* \to \mathbb{F}_q \times \{0,1\}$$

as input function. If an algorithm $\mathcal{F}_{slave}$ that can $(t - 2^I C_{G_1}(q_H + q_S + 1), q_H - q_S - 1, q_S, \varepsilon)$-break signature scheme on (G_1, G_2). Then we can construct an algorithm $\mathcal{F}$ that $(t, q_H, q_S, \varepsilon)$-breaks the signature scheme on (G_1, G_2) when an arbitrary hash function

$$H : \{0,1\}^* \to G_1$$

is used.

$\mathcal{F}$ will need to maintain a $q_H \times 2^I$ table $[s_{ij}]$ where $s_{ij} \in \mathbb{F}_q \times \{0,1\}$ for $i = 1, \ldots, q_H$ and $j = 1, \ldots, 2^I$. $\mathcal{F}$ starts by filling the table with uniformly randomly distributed values. $\mathcal{F}$ we will use algorithm 2.2.2 to maintain the table. Algorithm $\mathcal{F}$ runs algorithm $\mathcal{F}_{slave}$ as a slave algorithm feeding it

Algorithm 2.2.2: UpdateTable

Data: table $[s_{ij}]$, I-bit string w, message M_i
Result: updated table $[s_{ij}]$
foreach $j = 1, \ldots, 2^I$ **do**
  if $s_{ij} \overset{MapToGroup}{\longmapsto} G_1 \setminus \{\mathcal{O}\}$ **then**
    if $H(M_i) = Q_i = \mathcal{O}$ **then**
      Break "trivial forgery found"
  else
    choose $T_i \in E(\mathbb{F}_q)$ randomly
    $\tilde{Q}_i = nT_i + zQ_i$ where $n = |G_1| = |G_2|$ and $z = \left(\frac{m}{n}\right)^{-1} \pmod{n}$
    $s_{ij} \leftarrow (x(\tilde{Q}_i), b_i)$ s.t. $y_{b_i} = y(\tilde{Q}_i)$

information for doing computations needed to break the signature scheme on (G_1, G_2). $\mathcal{F}_{slave}$ can request the following information: H' hashed values and signatures of messages M_i, algorithm $\mathcal{F}$ will act as the gamekeeper and respond to these queries as described in 3) and 4) in the following scenario:

1. $\mathcal{F}$ fills the table $[s_{ij}]$ with uniformly randomly distributed values.

2. $\mathcal{F}$ inputs a public key v into the $\mathcal{F}_{slave}$ algorithm.

3. If $\mathcal{F}_{slave}$ requests a hash of $w||M_i$ and the message M_i is previously unseen then $\mathcal{F}$ will first use the Algorithm 2.2.2 to update the table $[s_{ij}]$. $\mathcal{F}$ returns the value s_{iw} to $\mathcal{F}_{slave}$. If it discovers a trivial forgery then $\mathcal{F}$ halts and returns the trivial forgery $(M_i, \mathcal{O})$.

4. If $\mathcal{F}_{slave}$ requests a signature σ_i of M_i then $\mathcal{F}$ will first check and update the table entry $[s_{ij}]$ corresponding to M_i using Algorithm 2.2.2. If it discovers a trivial forgery then $\mathcal{F}$ halts and returns the trivial forgery $(M_i, \mathcal{O})$. If not so, $\mathcal{F}$ will query its own game master for a signature on M_i and forward this to $\mathcal{F}'$ as σ_i.

5. If $\mathcal{F}_{slave}$ returns with failure to produce a forgery then $\mathcal{F}$ will report failure as well.

6. If $\mathcal{F}_{slave}$ returns a forgery signature pair (M_k, σ_k) and $\mathcal{F}$ runs Algorithm 2.2.2 to update row k. If it discovers a trivial forgery then $\mathcal{F}$ halts and returns the trivial forgery $(M_k, \mathcal{O})$.

7. $\mathcal{F}$ returns the forgery signature pair (M_k, σ_k).

Lemma 2.10. *The output forgery (M_k, σ_k) produced by $\mathcal{F}_{slave}$ is a valid forgery under the arbitrary hash function H used by $\mathcal{F}$.*

Proof. We want to show that the forged signature σ_k is valid in a scheme using hash function H. The signature σ_k is valid in a scheme using hash function $\text{MapToGroup}_{H'}$, thus we only need to show that the above construction of $\mathcal{F}$ ensures that

$$\text{MapToGroup}_{H'}(M_k) = H(M_k).$$

Given that algorithm $\mathcal{F}$ does not produce a trivial forgery, we have:

$$s_{kj} = (x_k, b_k) = (x(\tilde{Q}_k), b_k) \text{ s.t. } y_{b_i} = y(\tilde{Q}_i).$$

Let $m = |E(\mathbb{F}_q)|$ and $n = |G_1|$ then n divide m and thus $\frac{m}{n} \in \mathbb{Z}$. Next by the assumption $n^2 \nmid m$ we have that $(n, \frac{m}{n}) = 1$ and thus the inverse $z = \left(\frac{m}{n}\right)^{-1} \pmod{n}$ exists. When we map (x_k, b_k) to G_1 using MapToGroup we get:

$$(x_k, b_k) \mapsto \text{MapToGroup}_{H'}(M_k) = \frac{m}{n}\tilde{Q}_k = \frac{m}{n}(nT_k + zQ_k) = mT_k + Q_k = Q_k.$$

The point $Q_k = H(M_k)$ thereby proving the lemma. $\square$

By Lemma 2.10 the probability $Succ_{\mathcal{F}} \geq \varepsilon$. We assumed $\mathcal{F}_{slave}$ to run in time $t' = t - 2^I C_{G_1}(q_H + q_S + 1)$. Algorithm $\mathcal{F}$ will run in time t' plus the time it takes to update all the row entries in the table $[s_{ij}]$ for every hash and signature query and the last table look up, i.e

$$t' + 2^I C_{G_1}(q_H + q_S + 1) = t,$$

where C_{G_1} is the constant time it takes to run Algorithm 2.2.2. We assumed $\mathcal{F}_{slave}$ to make at most $q'_H = q_H - q_S - 1$ hash queries. Algorithm $\mathcal{F}$ will potentially do a H hash query for each hash and signature requests made by the slave algorithm and before terminating, i.e. $q'_H + q_S + 1 = q_H$ hash queries. Since the signature queries σ_i is just passed on by the master algorithm $\mathcal{F}$ it will also at most do q_S signature queries. I have now shown that algorithm $\mathcal{F}$ $(t, q_H, q_S, \varepsilon)$-breaks co-GDH on (G_1, G_2) when an arbitrary hash function $H : \{0,1\}^* \to G_1$ is used. $\square$

The stop parameter I in Theorem 2.9 is chosen in the following way, given the failure probability δ. We will divide the possibility of finding a solution x into the two cases. If characteristic $p \neq 2$ then the probability of $H'(i \mid\mid M)$ producing an x value such that $f(x)$ is a quadratic residue is approximatly $\frac{1}{2}$. This is because there are $(q+1)/2$ quadratic residues (including 0) and $(q-1)/2$ quadratic non-residues modulo an odd prime n. If the characteristic $p = 2$ then the probability of $H'(i \mid\mid M)$ producing a x value s.t. $tr(f(x)) = 0$ is $\frac{1}{2}$ by Lemma 2.4.

In each each case the algorithm will run 2^I iterations if the message is to be found unhashable. So the failure probability will be bounded by

$$\frac{1}{2^{2^I}} \leq \delta, \text{ i.e } I \geq \log\log\frac{1}{\delta}.$$

So when choosing $I = \lceil \log\log\frac{1}{\delta} \rceil$ you can force the failure probability to get smaller than δ. So you want a low value I and q_H much larger than q_S which seems to be a fair requirement to make.

2.3 Security of the BLS signature scheme

We are now ready to prove a theorem on the security of the BLS signature scheme.

The following theorem tells how the security of the signature scheme is bounded from below by the co-GDH parameters. In this way reducing the security to the hardness of the co-GDH problem on (G_1, G_2).

Theorem 2.11. *Let (G_1, G_2) be a (τ, t', ε')-co-GDH pair with $|G_1| = |G_2| = p$. Then the signature scheme on (G_1, G_2) is $(t, q_H, q_S, \varepsilon)$-existentially unforgeable under an adaptive chosen-message attack for all t and ε where*

$$\varepsilon \geq e \cdot (q_S + 1) \cdot \varepsilon' \quad \textit{and } t \leq t' - c_{G_1} \cdot (q_H + 2q_s),$$

c_{G_1} is the constant time it takes to do an exponentiation in G_1.

Proof. Assume for the purpose of contradiction that there exists an algorithm $\mathcal{A}$ that $(t, q_H, q_S, \varepsilon)$-breaks the signature scheme based on a co-GDH group pair given the bounds on ε and t. I want to construct an algorithm $\mathcal{B}$ that by help of algorithm $\mathcal{A}$ can (τ, t', ε')-break the co-GDH property on (G_1, G_2) and thus get a contradiction with the assumption of (G_1, G_2) being a (τ, t', ε')-co-GDH pair. Let g_2 generate G_2 and $h \in G_1$, algorithm $\mathcal{B}$ will get the input (g_2, g_2^a, h) and it will with some probability produce the output $h^a \in G_1$. Algorithm $\mathcal{B}$ uses $\mathcal{A}$ in the following way:

Algorithm 2.3.1: SimulateSignatureOracle

Data: message M_i, set H
Result: valid signature σ_i
$T_i \leftarrow$ `UpdateHList(`M_i`,`H`)`
if $T_i(c_i) = 0$ **then**
  return Failure: $c_i = 0$
else
  return $\sigma_i \leftarrow \psi(g_2^a)^{b_i} \cdot \psi(g_2)^{rb_i}$

1. $\mathcal{B}$ inputs into $\mathcal{A}$ $(g_2, g_2^a \cdot g_2^r)$, where r is assumed to be randomly chosen in $\mathbb{Z}_p$.

2. When $\mathcal{A}$ queries its random oracle H, then $\mathcal{B}$ will simulate H and provide $\mathcal{A}$ with a hash value by maintaining a H-list and if necessary updating it by using Algorithm 2.3.2.

3. When $\mathcal{A}$ queries for a signature σ_i on a message M_i then $\mathcal{B}$ will use Algorithm 2.3.1 to construct a valid signature and return it to $\mathcal{A}$.

4. In the end $\mathcal{A}$ will output a forgery (M_k, σ_k) such that $M_k \neq M_i \ \forall\ i$ in step 3. $\mathcal{B}$ checks whether its H-list contains an entry for message M_k. If not, $\mathcal{B}$ will update the list using Algorithm 2.3.2.

5. $\mathcal{B}$ checks if the outputted forgery (M_k, σ_k) is valid. If not, $\mathcal{B}$ returns failure.

6. $\mathcal{B}$ checks if $c_k \in T_k$ equal 1. If so $\mathcal{B}$ returns failure.

7. Otherwise $c_k = 0$ and then $\mathcal{B}$ returns the value
$$\frac{\sigma_k}{\psi(g_2^a)\psi(g_2)^{b_k r}}$$

Lemma 2.12. *The signature σ_i on M_i generated by using Algorithm 2.3.1 is valid under the public key g_2^{a+r}.*

Proof. If Algorithm 2.3.1 succeeds in generating a signature in step 3 then for the corresponding tuple T_i, it must be the case that $c_i = 1$ and thus
$$w_i = h^0 \cdot \psi(g_2)^{b_i} = \psi(g_2)^{b_i}.$$
Since $\psi : \ G_2 \to G_1$ is an isomorphism we can write σ_i as
$$\sigma_i = \psi(g_2^a)^{b_i} \cdot \psi(g_2)^{rb_i} = \psi(g_2)^{ab_i + rb_i} = (\psi(g_2)^{b_i})^{a+r} = w_i^{a+r},$$
and verify that σ_i is a valid signature on M_i under the public key g_2^{a+r}. $\square$

Lemma 2.13. *The value produced by algorithm $\mathcal{B}$*
$$\frac{\sigma_k}{\psi(g_2^a)\psi(g_2)^{b_k r}} = h^a.$$

Proof. If $\mathcal{B}$ produces a result in step 7 then $c_k = 0$ and thus
$$w_k = h \cdot \psi(g_2)^{b_k},$$
so we can write
$$\sigma_k = \left(h \cdot \psi(g_2)^{b_k}\right)^{a+r} = h^{a+r} \cdot \psi(g_2)^{b_k(a+r)}.$$
By calculating
$$\psi(g_2)^{b_k(a+r)} = \psi(g_2)^{b_k a}\psi(g_2)^{b_k r} = \psi(g_2^a)^{b_k}\psi(g_2)^{b_k r}$$
and inserting into the above expression for σ_k we get that
$$\frac{\sigma_k}{\psi(g^a)\psi(f_2)^{b_k r}} = h^a.$$
$\square$

We have constructed $\mathcal{B}$ and now we need to show that the probability
$$\text{Succ co-CDH}_{\mathcal{B}} \geq \varepsilon', \text{ when } \varepsilon \geq e \cdot (q_S + 1) \cdot \varepsilon'.$$

Algorithm 2.3.2: UpdateHList

Data: message M_i, set H
Result: tuple $T = \langle M_i, w_i, b_i, c_i \rangle$
foreach $T \in H$ **do**
  if $M_i \in T$ **then**
    return T
  else
    $c_i \overset{R}{\leftarrow} \{0,1\}$ with probability $p(c_i = 0) = \frac{1}{q_s+1}$
    $b_i \overset{R}{\leftarrow} \mathbb{Z}_p$ uniformly
    $w_i \leftarrow h^{1-c_i} \cdot \psi(g_2)^{b_i} \in G_1$
    $H \leftarrow H \cup \{\langle M_i, w_i, b_i, c_i \rangle\}$
    return $\langle M_i, w_i, b_i, c_i \rangle$

The following conditions must all be true for $\mathcal{B}$ to succeed:

C_1 : Every call to Algorithm 2.3.1 is successful, i.e. $c_i = 1$.

C_2 : σ_k is a valid signature on message M_k.

C_3 : In the tuple $T_k = \langle M_k, w_k, b_k, c_k \rangle c_k = 0$.

So we can write

$$\begin{aligned}
\text{Succ co-CDH}_{\mathcal{B}} &= P(C_1 \cap C_2 \cap C_3) \\
&= P(C_2 \cap C_3 \mid C_1)P(C_1) \\
&= P((C_2 \mid C_1) \cap (C_3 \mid C_1)P(C_1) \\
&= P((C_3 \mid C_1) \mid (C_2 \mid C_1))P(C_2 \mid C_1)P(C_1) \\
&= P(C_3 \mid C_1 \cap C_2)P(C_2 \mid C_1)P(C_1)
\end{aligned}$$

Claim 2.14. $P(C_1) \geq \frac{1}{e}$.

Proof. Assume without loss of generality that $\mathcal{A}$ does not query for a signature on a message M_i more than once. If $\mathcal{A}$ did make multiple queries on the same message then the probability for success would only be higher since fewer updates in Algorithm 2.3.2 would be required. Use the principle of induction on the number of queries i made to the Algorithm 2.3.1 to show that

$$p(C_{1i}) \geq \left(1 - \frac{1}{q_S + 1}\right)^i$$

Induction start: $i = 0$. No queries have yet been made and thus the probability for failure is zero. Induction hypothesis: Assume that the claim is true for all $j < i$. Inductive step: In the i'th signature query c_i will be set independently of the previous H-list queries to the Algorithm 2.3.2 made by the Algorithm 2.3.1. Thus the probability for failure is in signature query i less than or equal $\frac{1}{q_S+1}$ (since $\mathcal{A}$ could ask a signature on a M_i already present in a tuple in the H-list). If we then calculate the probability of the i'th query to return without failure we get

$$p(C_{1i}) \geq \left(1 - \frac{1}{q_S+1}\right)^{i-1} \left(1 - \frac{1}{q_S+1}\right) = \left(1 - \frac{1}{q_S+1}\right)^{i}$$

By the principle of induction we have shown the above statement. After q_S signature queries we will have

$$p(C_1) \geq \left(1 - \frac{1}{q_S+1}\right)^{q_S} \geq \frac{1}{e},$$

by noting that

$$\left(1 - \frac{1}{x+1}\right)^{x} \geq \frac{1}{e}.$$

holds for $x \geq 0$. □

Claim 2.15. $P(C_2 \mid C_1) \geq \varepsilon$.

Proof. Given that the condition C_1 is true, then algorithm $\mathcal{A}$ will terminate. By the assumption that algorithm $\mathcal{A}$ $(t, q_H, q_S, \varepsilon)$-breaks the signature, we know by our definition of algorithm $\mathcal{A}$ that $\mathcal{A}$ returns a valid (M_k, σ_k) signature pair with probability at least ε, so

$$P(C_2 \mid C_1) = \text{Succ forgery}_{\mathcal{A}} \geq \varepsilon.$$

□

Claim 2.16. $P(C_3 \mid C_1 \cap C_2) = \frac{1}{q_S+1}$.

Proof. First let us look at the dependence of event $C_1 \cap C_2$ and the value of $c_k = 0$. When $c_k = 0$, the prior signature queries made by $\mathcal{A}$ only gives information on those c_i for which the signature query on related M_i was made. We know that $\mathcal{A}$ has not made a signature query on M_k and so the only information available about c_k will be $H(M_k)$, but the distribution of values on H is uniform. We can therefore assume that the probability $P(C_3 \mid C_1 \cap C_2)$ is independent of the prior signature queries made by $\mathcal{A}$ and the queries to the Algorithm 2.3.2, so we may write

$$\begin{aligned} P(c_k = 0 \mid C_1 \cap C_2) &= \frac{P\left((c_k = 0) \cap (C_1 \cap C_2)\right)}{P(C_1 \cap C_2)} \\ &= \frac{P(c_k = 0) P(C_1 \cap C_2)}{P(C_1 \cap C_2)} \\ &= P(c_k = 0) \\ &= \frac{1}{q_S+1}. \end{aligned}$$

□

From the above proved three claims we now see that

$$\text{Succ co-CDH}_{\mathcal{B}} \geq \frac{1}{e}\varepsilon\frac{1}{q_S+1} \geq \varepsilon'$$

since we asserted that $\varepsilon \geq e \cdot (q_S + 1) \cdot \varepsilon'$. The running time of $\mathcal{B}$ can be summed up in the following way

$$\begin{aligned} \text{Running time}_{\mathcal{B}} = & \text{Running time } t \text{ for } \mathcal{A} \\ & + \text{time to answer } (q_H + q_S) \ H\text{-queries and} \\ & \quad q_S \text{ signature queries} \\ = & t + c_{G_1}(q_H + 2q_S), \end{aligned}$$

here c_{G_1} is the constant amount of time it takes to run the Algorithms 2.3.1 and Algorithm 2.3.2. By the assumption $t \leq t' - c_{G_1} \cdot (q_H + 2q_s)$ we see that

$$\text{Run time}_{\mathcal{B}} = t + c_{G_1}(q_H + 2q_S) \leq t'.$$

So by Definition 1.6 algorithm $\mathcal{B}$ (t', ε')-breaks co-GDH on (G_1, G_2), thus yielding a contradiction. It is now proved that the signature scheme based on the co-GDH pair (G_1, G_2) is $(t, q_H, q_S, \varepsilon)$-existentially unforgeable under an adaptive chosen-message attack. □

Chapter 3

The Weil pairing

The Weil pairing is named after André Weil (1906-1998) even though it has been around since Karl Wilhelm Theodor Weierstrass (1815-1897) introduced it as the sigma function on elliptic curves. André Weil (1906-1998) gave a more abstract definition of this mapping in his first proof of the Riemann hypothesis for arbitrary genus curves over finite fields [Sur les fonctions algébriques à corps de constantes finis, C.R. Académie des Sciences, 1940]. The definition is also refered to and restated in the article [On the Riemann hypothesis in function-fields, New School for social research, 1941].

In the following theorem the existence of the Weil pairing is stated along with some of its properties. First we will introduce divisors, then the Weil pairing is constructed on elliptic curves and some of the properties of the Weil pairing are proved. Then we will use Victor Miller's algorithm for efficiently computing the Weil pairing. We will implement the Weil pairing in Sage with Miller's algorithm and show that it runs linearly in the number of bits of its input points order n.

Theorem 3.1. *Let E be an elliptic curve defined over a field K, let n be a positive integer and let μ_n be the set of n'th roots of unity. Assume that K's characteristic $p \nmid n$ then there exists a pairing*

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

such that:

a. $e_n(P_1 + P_2, Q) = e_n(P_1, Q)e_n(P_2, Q)$ *and*

$e_n(P, Q_1 + Q_2) = e_n(P, Q_1)e_n(P, Q_2)$ *(bilinearity).*

b. *If* $e_n(P, Q) = 1$ *for all* $Q \in E[n]$ *then* $P = \mathcal{O}$ *and*

if $e_n(P, Q) = 1$ *for all* $P \in E[n]$ *then* $Q = \mathcal{O}$ *(non-degeneracy).*

c. $e_n(P, P) = 1$ *for all* $P \in E[n]$ *(alternating) .*

d. $e_n(P, Q) = e_n(Q, P)^{-1}$ *for all* $P, Q \in E[n]$ *(skew symmetry).*

e. $e_n(\sigma P, \sigma Q) = \sigma(e_n(P, Q))$ *for all* $\sigma \in Gal(\bar{K}/K)$ *(Galois action).*

Two apparently important properties with respect to our signature scheme are bilinearity and non-degeneracy. It will later on be explained how bilinearity makes it easy to solve the co-DDH problem. The property of non-degeneracy is important to ensure that the kernel of the map $P \mapsto e_n(P, Q)$ is trivial, which we will need to check that a tuple is a co-DDH tuple in the verification step in the BLS signature scheme. Besides the trivial pairings with $\mathcal{O}$, pairings of linear dependent points should also be noted.

Remark 3.2. *Note that given two points $P, Q \in E[n]$ where $Q = kP$, i.e. Q and P are linearly dependent, we have that $e_n(P, Q) = 1$ by properties a and c.*

First we will need some theory on divisors before we will be able to prove the existence and the properties of the Weil pairing in the following sections.

3.1 Divisor theory

Let us define what we mean when we say divisors, sum and degree in respect to divisor theory.

Definition 3.3. *Let K be a field. A divisor D on an elliptic curve E is a formal sum of symbols $[P_i]$ representing each point P_i in the curve group $E(\bar{K})$*

$$D = \sum_i a_i[P_i], \quad a_i \in \mathbb{Z}$$

The set of all divisors is denoted by $Div(E)$.

Definition 3.4. *The degree of a divisor D is a map*

$$deg : Div(E) \to \mathbb{Z}$$

where

$$\deg(D) = \deg\left(\sum_i a_i[P_i]\right) = \sum_i a_i \in \mathbb{Z}.$$

Remark 3.5. *The kernel of the degree function is the set of divisors of degree 0:*

$$Div^0(E) := \{D | deg(D) = 0\}.$$

Definition 3.6. *The sum of a divisor D is*

$$sum(D) = sum\left(\sum_i a_i[P_i]\right) = \sum_i a_i P_i \in E(\bar{K}).$$

When we look at functions on an elliptic curve $E(K)$ and count zeros and poles of the function we can define divisors of functions. We use the following theorem to count zeros and poles.

Theorem 3.7. *There exists a function u_P called the uniformizer at a point P s.t. for every function f there exists $r \in \mathbb{Z}$ and a function g satisfying $g(P) \neq 0, \infty$ such that*

$$f = u_P^r g.$$

Definition 3.8. *The order of a function at point P is given as the exponent r of the uniformizer u_P in the above expression and is written $ord_P(f)$.*

Definition 3.9. *The divisor of a function f not identically 0 is defined as*

$$div(f) = \sum_{P \in E(\bar{K})} ord_P(f)[P]$$

The divisor of a function is called a principal divisor.

An immediate consequence of this definition is the rules

$$div(f/g) = div(f) - div(g),$$
$$div(fg) = div(f) + div(g).$$

The principal divisors turns out to be a subset of the subgroup of divisors of degree 0, We can define an equivalence relation on Div^0 using principal divisors.

Definition 3.10. *We define an equivalence relation $\sim$ on the set of divisors on E by saying that two divisors D and D' are equivalent if $D - D'$ is principal i.e. $D' = D + div(f)$ for a principal divisor $div(f)$.*

This gives us a set of divisor classes w.r.t. the relation $\sim$:

$$Div^0(E)/\sim$$

which is a group.

Next we want to prove an important theorem by Abel and Jacobi. We will need a couple of lemmas. We will not prove these lemmas, here but the proof of Lemma 3.11 can be found in ”Algebraic Curves: An Introduction to Algebraic Geometry” [Ful89, chap. 8] and the proof of Lemma 3.13 can be found in Washington [Was08, p.345].

Lemma 3.11. *Let E be an elliptic curve and $f \neq 0$ a function on E, then the following holds:*

1. *f has only a finite number of zeroes*
2. *$deg(div(f)) = 0$.*
3. *If $div(f) = 0$ then f is a constant.*

The following is an example using the above stated theorem.

Example 3.12. *Let E be an elliptic curve over a field K and let $P, Q \in E(K)$. Let $\ell_{P,Q}$ be the equation of the line passing through P and Q as for defining the point composition $P * Q$ in Definition 1.11.*

$$\ell_{P,Q}: \ ax + by + c = 0, \ a, b, c \in K.$$

If $P = Q$ then $\ell_{P,Q}$ is taken to be the tangent at P. Define

$$g_{P,Q} := \frac{L_{P,Q}}{L_{(P+Q),-(P+Q)}}.$$

Let us call the function defined by the left hand side of line equation $\ell_{P,Q}$ for $L_{P,Q}$. Let us try to determine the divisor for $g_{P,Q}$. First look at the divisor for $L_{P,Q}$. For $b \neq 0$ the line defined by $\ell_{P,Q}$ will intersect E in precisely 3 points $P, Q, -(P+Q) \neq \mathcal{O}$. By Lemma 3.11 the degree has to add up to 0 since $L_{P,Q}$ is a function, we must necessarily have all 3 poles at $\mathcal{O}$. Therefore the divisor of $L_{P,Q}$ is given as

$$div(L_{P,Q}) = [P] + [Q] + [-(P+Q)] - 3[\mathcal{O}]$$

and the divisor for $L_{(P+Q),-(P+Q)}$

$$div(L_{(P+Q),-(P+Q)}) = [P+Q] + [-(P+Q)] + [\mathcal{O}] - 3[\mathcal{O}].$$

Compute then

$$div(g_{P,Q}) = div(L_{P,Q}) - div(L_{(P+Q),-(P+Q)}) = [P] + [Q] - [P+Q] - [\mathcal{O}].$$

Lemma 3.13. *Let $P, Q \in E(\bar{K})$, if there exists a function h on E with divisor*

$$div(h) = [P] - [Q].$$

Then $P = Q$

Theorem 3.14 (Abel-Jacobi)**.** *Let E be an elliptic curve. Let D be a divisor on E with $\deg(D) = 0$. Then*

$$\text{there exists a function } f \text{ on } E \text{ such that } div(f) = D,$$

if and only if

$$sum(D) = \mathcal{O}$$

Proof. We will start by showing the following claim:

Claim 3.15. *The divisor D can be written in the convenient way*

$$D = [P] - [Q] + div(g) \text{ and } sum(D) = P - Q.$$

Proof. In Example 3.12 we showed for points P_1 and P_2 on E that

$$[P_1] + [P_2] = [P_1 + P_2] + [\mathcal{O}] + div(g_{P_1,P_2}),$$

if $P_1 + P_2 = \mathcal{O}$ the above expression can be simplified further

$$[P_1] + [P_2] = 2[\mathcal{O}] + div(g_{P_1,P_2}). \tag{3.1}$$

Also note that the sum

$$sum(div(g_{P_1,P_2})) = \mathcal{O}.$$

The divisor D is defined as the formal sum of elements (points) with signs. There will be some positive terms and some negative terms. Using the above expression (3.1), the positive and the negative parts of the sum can each be summed up to

$$\begin{aligned} D_+ =& [P] + n_1[\mathcal{O}] + div(g_1), \\ D_- =& -\left([Q] + n_2[\mathcal{O}] + div(g_2)\right). \end{aligned}$$

Note that the divisors $div(g_i)$ is a result of the divisors summing the negative and positive parts pairwise and can be written like

$$\begin{aligned} div(g_1) &= \sum div(g_{P_i,P_j}) \text{ and} \\ div(g_2) &= \sum div(g_{Q_i,Q_j}). \end{aligned}$$

Looking at the divisor D in this way we can write

$$\begin{aligned} D = D_+ + D_- &= [P] - [Q] + (n_1 - n_2)[\mathcal{O}] + div(g_1) - div(g_2) \\ &= [P] - [Q] + n[\mathcal{O}] + div(g). \end{aligned}$$

Observation 3.16. *The sum of the divisor of g in the above is*

$$\begin{aligned} sum(div(g)) &= sum(div(g_1) - div(g_2)) \\ &= sum\left(\sum div(g_{P_i,P_j}) - \sum div(g_{Q_i,Q_j})\right) \\ &= \sum sum(div(g_{P_i,P_j})) - \sum sum(div(g_{Q_i,Q_j})) \\ &= \sum \mathcal{O} - \sum \mathcal{O} \\ &= \mathcal{O}. \end{aligned}$$

By Lemma 3.11 the degree $deg(div(g)) = 0$ and using the assumption $deg(D) = 0$:

$$deg(D) = 1 - 1 + n + 0 = n \Rightarrow n = 0 \Rightarrow D = [P] - [Q] + div(g) \text{ and}$$
$$sum(D) = P - Q + sum(div(g)) = P - Q - \mathcal{O} = P - Q.$$

□

Now we're ready to prove the if and only if statement. First assume that $sum(D) = \mathcal{O}$. From the claim

$$sum(D) = P - Q, \text{ i.e. } P = Q,$$

so the divisor $D = div(g)$ and we can choose $f = g$.

Next the *only if* part, now assume that $D = div(f)$. From the claim we write

$$div(f) = D = [P] - [Q] + div(g), \text{ i.e. } [P] - [Q] = div\left(\frac{f}{g}\right).$$

By Lemma 3.13 where we choose $h = \frac{f}{g}$, we see that $P = Q$ and thus

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

□

Corollary 3.17. *There exists an one-to-one correspondence between the divisor classes of degree 0 and points on the elliptic curve $E(\bar{K})$.*

Proof. Define the map $sum : Div^0 \to E(\bar{K})$ by

$$sum : D \mapsto sum(D).$$

The map sum is a homomorphism, since it is defined by the sum of a divisor. It is surjective since $[P] - [\mathcal{O}] \in Div^0(E)$ for all $P \in E(\bar{K})$. The kernel of the map *sum* is the set of all the principal divisors by Abel-Jacobi. The equivalence relation determines divisors up to a principal divisor. So by Noether's first isomorphism theorem:

$$Div^0/\sim\ \simeq\ E(\bar{K}).$$

□

We can now give an alternative proof of the group laws on the set of points on E.

Theorem 3.18. *Points on an elliptic curve E/K form an abelian group $E(\bar{K})$.*

Proof. We saw in the above corollary that

$$E(\bar{K}) \simeq Div^0/\sim .$$

So it's enough to show that $Div^0/\sim$ is abelian. Look at two elements D_P and D_Q and note that the composition in this group is addition of the class representitives

$$D_P + D_Q = [P] - [\mathcal{O}] + ([P] - [\mathcal{O}])\,.$$

We want to check that they commute

$$D_P + D_Q \sim D_Q + D_P.$$

This is clear when we compute

$$sum\,(([P] - [\mathcal{O}]) + ([Q] - [\mathcal{O}]) - ([Q] - [\mathcal{O}]) - ([P] - [\mathcal{O}])) = \mathcal{O}$$

and again since

$$deg\,(([P] - [\mathcal{O}]) + ([Q] - [\mathcal{O}]) - ([Q] - [\mathcal{O}]) - ([P] - [\mathcal{O}])) = 0$$

we can use Abel-Jacobi to see that the difference is principal and thus

$$D_P + D_Q \sim D_Q + D_P.$$

□

We would like to be able to evaluate functions of divisors. We do this as stated in the following definition.

Definition 3.19. *For any function f with a divisor $div(f) = D$ that share no points with the divisor $D' = \sum_i a_i[P_i]$ we define*

$$f(D') = \prod_i f(P_i)^{a_i}.$$

3.2 Constructing the Weil pairing

In this section the existence of the Weil pairing will be proven by constructing it.

Proof of existence in Theorem 3.1. This proof follows the approach of Washington [Was08]. Let T be a point of order n, i.e. $T \in E[n]$ and look at the divisor $D = n[T] - n[\mathcal{O}]$ then

$$sum(D) = nT - n\mathcal{O} = \mathcal{O},$$

and thus we can apply Theorem 3.14; and see that there exists a function f on E such that

$$div(f) = n[T] - n[\mathcal{O}]. \tag{3.2}$$

We now choose T' such that $T' \in E[n^2]$ this is done by choosing T' so $T = nT'$ and therefore $n^2T' = nT = \mathcal{O}$.

Observation 3.20. *Choose arbitrarily two different $T', T'' \in E[n^2]$ in the above way and observe that*

$$nT' - nT'' = T - T = \mathcal{O}, \;\; i.e. \; n(T' - T'') = \mathcal{O}$$

and so the difference $(T' - T'') \in E[n]$.

Now consider the divisor

$$D' = \sum_{R \in E[n]} \left([T' + R] - [R]\right).$$

Note that the sum is over n^2 different points $R \in E[n]$ so one can write the sum of D' as

$$sum(D') = \sum_{R \in E[n]} T' + R - R = \sum_{R \in E[n]} T' = n^2T' = \mathcal{O}.$$

Also apply Theorem 3.14 on the divisor D' to see that there exist a function g on E such that

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

Using Observation 3.20 rewrite the above sum defining the divisor

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

Now define the map $\tau_n : P \mapsto nP$ for points $P \in E$ and a positive integer n. Look at the map $f \circ \tau_n$ which first multiplies a point by n and then applies f on the multiplum. Let $P = T' + R$ with $R \in E[n]$ then it holds for this P that $nP = nT' + nR = nT' = T$. And since $R \in E[n]$, i.e. $nR = \mathcal{O}$ we may write

$$div(f) = n[nP] - n[nR].$$

So the divisor of $f \circ \tau_n$ can be written as

$$\begin{aligned} div(f \circ \tau_n) &= n[P] - n[R] \\ &= n[T' + R] - n[R] \text{ for all } R \in E[n] \\ &= n \left(\sum_{R \in E[n]} [T' + R] - \sum_{R \in E[n]} [R] \right) \\ &= n \cdot div(g) = div(g^n). \end{aligned}$$

Let us look at the expression

$$div(f \circ \tau_n) = div(g^n) \Leftrightarrow div(f \circ \tau_n) - div(g^n) = 0 \Leftrightarrow div\left(\frac{f \circ \tau_n}{g^n}\right) = 0,$$

so $\frac{f \circ \tau_n}{g^n}$ does not have any zeroes or poles, i.e. it must be a constant function different from 0 by Lemma 3.11. So we're able to multiply with a suitable constant $c \neq 0$ and get $f \circ \tau_n = c \cdot g^n$.

Let $S \in E[n]$ and let $P \in E(\bar{K})$ then

$$\begin{aligned} c \cdot g(P + S)^n &= (f \circ \tau_n)(P + S) \\ &= f(n(P + S)) \\ &= f(nP + \mathcal{O}) \\ &= f(nP) = (f \circ \tau_n)(P) = c \cdot g(P)^n. \end{aligned}$$

Rewrite the discovered identity

$$c \cdot g(P + S)^n = c \cdot g(P)^n \Leftrightarrow \frac{g(P + S)^n}{g(P)} = \left(\frac{g(P + S)}{g(P)}\right)^n = 1,$$

to see that $\frac{g(P+S)}{g(P)}$ is an n'th root of unity in $\bar{K}$. □

We define the map

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

as the Weil pairing. The next result shows that the map is unique with respect to points T and S.

Theorem 3.21. *The function $\frac{g(P+S)}{g(P)}$ is independent of the choice of P.*

Proof of this theorem is sketched in Washington [Was08, p.350]. The last theorem of this section is a technical result needed to construct the Weil pairing, it will only be stated here. Proof can be found in Washington [Was08, p.300].

Theorem 3.22. *Let E be an elliptic curve over the field K, and let g be a function on E and n a natural number such that the characteristic of K $p \nmid n$. If $g(P+T) = g(P)$ for all $P \in E(\bar{K})$ and all $T \in E[n]$. Then there exists a function h on E such that $g(P) = h(nP)$.*

3.3 Properties of the Weil pairing

In this section I will prove the properties of the Weil pairing given in Theorem 3.1.

Property e will not be proved, the proof consists of going through the construction of the Weil pairing again and checking that the automorphism $\sigma \in Gal(\bar{K}/K)$ can be carried through the whole construction providing us property e.

Proof of the properties a. - d. in Theorem 3.1. This proof follow Washington [Was08]. We prove the first four properties in the order: a.,c.,d., b.

a.

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

is bilinear. We saw in Theorem 3.21 that the pairing value is independent of the choice of the point P. Choose points P and $P+S_1$ to define the value of the pairings in the product

$$\begin{aligned} e_n(S_1,T)e_n(S_2,T) &= \frac{g(P+S_1)}{g(P)}\frac{g(P+S_1+S_2)}{g(P+S_1)} = \frac{g(P+S_1+S_2)}{g(P)} \\ &= e_n(S_1+S_2,T). \end{aligned}$$

This shows bilinearity in the first variable. Choose $T_i \in E[n]$, $i = 1,2,3$ such that $T_1 + T_2 = T_3$, then it follows from Theorem 3.14 that there exists a function h on E such that

$$div(h) = [T_3] - [T1] - [T_2] + [\mathcal{O}].$$

Let f_i and g_i be the functions defining the pairing $e_n(S,T_i)$ in the construction, then from Equation 3.2

$$div(f_i) = n[T_i] - n[\mathcal{O}],$$

and we can write

$$div\left(\frac{f_3}{f_1 f_2}\right) = div(f_3) - div(f_1) - div(f_2) = n \cdot div(h) = div(h^n).$$

So by Lemma 3.11 there exists a constant $c \neq 0$ such that $f_3 = c \cdot f_1 f_2 h^n$. If we apply the map $\tau_n : P \mapsto nP$ we get

$$\begin{aligned} f_3 &= c \cdot f_1 f_2 h^n \\ f_3 \circ \tau_n &= c \cdot (f_1 \circ \tau_n)(f_2 \circ \tau_n)(h \circ \tau_n)^n \ (\tau_n \text{ is applied to all } n \text{ copies of } h) \\ g_3^n &= c \cdot g_1^n g_2^n (h \circ \tau_n)^n \ (g_i^n = f_i \circ \tau_n) \\ g_3 &= c^{\frac{1}{n}} g_1 g_2 (h \circ \tau_n). \end{aligned}$$

This makes it possible to calculate

$$\begin{aligned} e_n(S, T_1 + T_2) &= e_n(S, T_3) = \frac{g_3(P+S)}{g_3(P)} \\ &= \frac{g_1(P+S)}{g_1(P)} \frac{g_2(P+S)}{g_2(P)} \frac{h(n(P+S)}{h(nP)} \\ &= \frac{g_1(P+S)}{g_1(P)} \frac{g_2(P+S)}{g_2(P)} \frac{h(nP)}{h(nP)} \\ &= \frac{g_1(P+S)}{g_1(P)} \frac{g_2(P+S)}{g_2(P)} \\ &= e_n(S, T_1) e_n(S, T_2). \end{aligned}$$

This shows bilinearity in the second variable.

c. The pairing is alternating in its variables:

$$\forall T \in E[n] : e_n(T, T) = 1.$$

Let $\tau_{jT} : P \mapsto P + jT$ be the map that translates a point $P \in E$ by a multiple of another point T. From the mapping where you first apply τ_{jT} and next the f from the construction of the pairing you get that the divisor

$$div(f \circ \tau_{jT}) = n[T - jT] - n[-jT] = n[(1-j)T] - n[-jT].$$

We recognize the above as something similar to a term in a telescoping sum and therefore write up the divisor

$$\begin{aligned} div\left(\prod_{j=0}^{n-1} f \circ \tau_{jT}\right) &= \sum_{j=0}^{n-1} (n[(1-j)T] - n[-jT]) \\ &= n \sum_{j=0}^{n-1} ([(1-j)T] - [-jT]) \\ &= n([T] - [(-n+1)T]) = n([T] - [T]) = 0. \end{aligned}$$

So from Lemma 3.11 we have that $\prod_{j=0}^{n-1} f\circ\tau_{jT}$ must be constant. Therefore, when $nT' = T$ we can write

$$\begin{aligned}\left(\prod_{j=0}^{n-1} g\circ\tau_{jT'}\right)^n &= \prod_{j=0}^{n-1} g^n\circ\tau_{jT'}\\ &= \prod_{j=0}^{n-1}(f\circ\tau_n)\circ\tau_{jT'}\\ &= \prod_{j=0}^{n-1} f\circ\tau_{jT}\circ\tau_n,\end{aligned}$$

so $\left(\prod_{j=0}^{n-1} g\circ\tau_{jT'}\right)^n$ is also constant. Then we'll gat that also $\prod_{j=0}^{n-1} g\circ\tau_{jT'}$ is constant[1]. So the value of the function $\prod_{j=0}^{n-1} g\circ\tau_{jT'}$ is the same in the different points P and $P' = P + T'$ and we may write

$$\prod_{j=0}^{n-1} g(P+jT') = \prod_{j=0}^{n-1} g(P+T'+jT')$$

Dividing out the common factors on both sides of the equation leaves

$$g(P) = g(P+nT'),\text{ i.e. } g(P) = g(P+T).$$

Note that in the division we have chosen P such that we do not divide with zero. We can do this since the pairing value was independent of the choice of point P by Theorem 3.21. But then from the construction of the Weil pairing we get that:

$$e_n(T,T) = \frac{g(P+T)}{g(T)} = 1.$$

d. e_n is skew symmetric in its variables, i.e.

$$\forall S,T\in E[n] : e_n(T,S) = e_n(S,T)^{-1}.$$

This is the same as saying

$$\forall S,T\in E[n] : e_n(T,S)e_n(S,T) = 1.$$

Using properties a. and c. we get that

$$\begin{aligned}1 = e_n(S+T,S+T) &= e_n(S,S)e_n(T,T)e_n(T,S)e_n(S,T)\\ &= e_n(T,S)e_n(S,T),\end{aligned}$$

which proves the above statement.

[1] This is taken as a fact. It comes from a deeper topological result on the connectedness of E in Zariski topology.

b. e_n is non-degenerate in each variable. We start by showing the non-degeneracy for the second variable T:

$$e_n(S,T) = 1 \ \forall S \in E[n] \Rightarrow T = \mathcal{O}.$$

Rewrite the hypothesis in the above implication to

$$g(P+S) = g(P) \ \forall P \in E(\bar{K}), \ \forall S \in E[n].$$

It follows from Theorem 3.22 that there exists a function h such that $g = (h \circ \tau_n)$ where $\tau_n : P \mapsto nP$ for $P \in E$ and $n \in \mathbb{N}$. So now write

$$h^n \circ \tau_n = (h \circ \tau_n)^n = g^n = f \circ \tau_n$$

This means that $h^n = f$ since τ_n is a surjective mapping. Thus

$$n \cdot div(h) = div(f) = n[T] - n[\mathcal{O}] \text{ i.e. } div(h) = [T] - [\mathcal{O}],$$

and then it follows from Lemma 3.13 that $T = \mathcal{O}$.

Next show non-degeneracy in the first variable S:

$$e_n(S,T) = 1 \ \forall T \in E[n] \Rightarrow S = \mathcal{O}.$$

First apply skew symmetry property d. in the hypothesis in the above implication and get that

$$e_n(T,S)^{-1} = e_n(S,T) = 1 \Rightarrow e_n(T,S) = 1,$$

which leaves us with the statement for the second variable, which has already been shown. □

The following corollary shows that if all points of order n is in $E(K)$ then the set of roots of unity, which the Weil pairing maps into will be a subset of K and not just $\bar{K}$.

Corollary 3.23. *If $E[n] \subseteq E(K)$ then $\mu_n \subset K$.*

Proof. We saw in Theorem 1.17 that the n-torsion is a product of two cyclic groups. Let the two points (T_1, T_2) generate $E[n]$. First we prove that for generators (T_1, T_2), the pairing value $e_n(T_1, T_2)$ is a primitive n'th root of unity. Suppose first that $e_n(T_1, T_2) = \eta$. Then $\eta^d = 1$ for some $d|n$. Then by a. and c. we get

$$e_n(T_1, dT_2) = e_n(T_2, dT_2) = 1.$$

For all $S \in E[n]$ we can write $S = aT_1 + bT_2$ and

$$\begin{aligned} e_n(S, dT_2) &= e_n(aT_1 + bT_2, dT_2) = e_n(aT_1, dT_2)e_n(bT_2, dT_2) \\ &= e_n(T_1, dT_2)^a e_n(T_2, dT_2)^b = 1. \end{aligned}$$

So from the non-degeneracy we get that $dT_2 = \mathcal{O}$ which imply $n|d$, so η is a primitive n'th root of unity. We use property e to see that all automorphisms $\sigma \in Gal(\bar{K}/K)$ fixes all pairing values η of points in $E[n]$. This means $\mathcal{F}\left(Gal(\bar{K}/K)\right) = K$. So $\eta \in K$. And since η is a primitive root of unity we have the statement. □

From Remark 3.2 we have already seen that we will get some trivial pairings. The next theorem shows that there exist non-trivial pairing values over a finite field $\mathbb{F}_q$. We will need this later on when we construct co-GDH group pairs from elliptic curve groups using the Weil pairing.

Theorem 3.24. *Let E be an elliptic curve defined over $\mathbb{F}_q$ with a point $P \in E(\mathbb{F}_q)$ of prime order n with $n \nmid q$. If the subgroup $\langle P \rangle$ has embedding degree $k > 1$ then $E(\mathbb{F}_q^k)$ contains a point Q of order n that is linearly independent of P.*

We do not prove this theorem, instead we look at another theorem implying that there are in fact $n(n-1)$ of P linearly independent pairs of points of order n in $E(\mathbb{F}_{q^k})$.

Theorem 3.25 (Balasubramanian-Koblitz)**.** *Let E be an elliptic curve defined over $\mathbb{F}_q$ and suppose that n is a prime and that $n \mid |E(\mathbb{F}_q)|$ but also that $n \nmid q-1$. If $n \mid (q^k - 1)$ then $E(\mathbb{F}_{q^k})$ contains n^2 points of order n.*

Proof. Proof is due to Balasubramanian-Koblitz [BK98]. Since $n \mid |E(\mathbb{F}_q)|$, there exist a non-trivial point $P \in E(\mathbb{F}_q)$ of order n. From Corollary 1.18 we know there exists an r such that $E(\mathbb{F}_{q^r}) \supset \mathbb{Z}_n \times \mathbb{Z}_n$. Let Q be a point on $E(\mathbb{F}_{q^r})$ so P and Q make a basis for the vector space $V = \mathbb{Z}_n \times \mathbb{Z}_n$. Look at the map

$$\Phi_q : V \to V,\ \ \Phi_q(x, y) = (x^q, y^q)$$

Φ_q is also known as the Frobenius endomorphism [Was08] and over the vector space V Φ_q is a $\mathbb{Z}_n$-linear mapping of the points of order n in $E(\mathbb{F}_{q^r})$. We know that $\Phi_q(P) = P$, since $x(P), y(P) \in \mathbb{F}_q$. We can therefore write the linear map Φ_q as a matrix in the basis (P, Q):

$$\Phi_q = \begin{pmatrix} 1 & a \\ 0 & b \end{pmatrix}, \text{ for } a, b \in \mathbb{Z}_n.$$

It is known that the determinant of this matrix is q [Was08, prop.4.11] and therefore we have that $b = q$. We assumed that $n \nmid q-1$ i.e. $q \not\equiv 1 \pmod n$ so the matrix has two distinct eigenvalues and can be diagonalized. Note that

$$\Phi_q^2 = \begin{pmatrix} 1 & a \\ 0 & q \end{pmatrix}^2 = \begin{pmatrix} 1 & a + qa \\ 0 & q^2 \end{pmatrix}.$$

Let the above be the induction start. Assume that the following holds for some $j > 1$

$$\Phi_q^j = \begin{pmatrix} 1 & c \\ 0 & q^j \end{pmatrix},$$

where $c \in \mathbb{F}_{q^r}$. Then

$$\Phi_q^{j+1} = \begin{pmatrix} 1 & c \\ 0 & q^j \end{pmatrix} \begin{pmatrix} 1 & a \\ 0 & q \end{pmatrix} = \begin{pmatrix} 1 & a+qc \\ 0 & q^{j+1} \end{pmatrix} = \begin{pmatrix} 1 & c' \\ 0 & q^{j+1} \end{pmatrix}.$$

From the principle of induction we have shown that for some $c \in \mathbb{F}_{q^r}$

$$\Phi_q^k = \begin{pmatrix} 1 & c \\ 0 & q^k \end{pmatrix}.$$

We initially assumed that $q^k \equiv 1 \pmod{n}$ so we may write

$$\Phi_q^k = \begin{pmatrix} 1 & c \\ 0 & 1 \end{pmatrix},$$

this is an upper triangle matrix and it is diagonisable. Since Φ_q is diagonisable, there exists a matrix B such that D is a diagonal matrix,

$$\begin{aligned} \Phi_q &= BDB^{-1} \text{ and} \\ \Phi_q^k &= (BDB^{-1})^k \\ &= BDB^{-1}B \cdots B^{-1}BDB^{-1} \\ &= BD^kB^{-1}. \end{aligned}$$

Then Φ_q^k has two linearly independent eigenvectors. There are already 1's in the diagonal of Φ_q^k so the only possibility for c is 0, i.e. $\Phi_q^k = Id$. Now we have that

$$\Phi_q^k(R) = (x^{q^k}, y^{q^k}) = R \text{ for all } R \in E(\mathbb{F}_{q^r}),$$

i.e. $r \mid k$ and

$$\mathbb{F}_{q^r} \subseteq \mathbb{F}_{q^k} \text{ and thus } E(\mathbb{F}_{q^r}) \subseteq E(\mathbb{F}_{q^k}).$$

We have thereby shown that since $E(\mathbb{F}_{q^r})$ contains n^2 points of order n, then so must $E(\mathbb{F}_{q^k})$. □

3.4 Calculating the Weil pairing

Calculating the Weil pairing can be done efficiently using Victor Miller's algorithm [Mil04], in this section the algorithm is described and proved to run in linear time. The first theorem gives a more convenient form of the Weil pairing when having to evaluate it in specific points.

Theorem 3.26. *Let $P, Q \in E[n]$ and D_P, D_Q be divisors of degree 0 such that*

$$sum(D_P) = P \text{ and } sum(D_Q) = Q,$$

and D_P and D_Q share no points. Let $f_{n,P}$ and $f_{n,Q}$ be functions s.t.

$$div(f_{n,P}) = nD_P \text{ and } div(f_{n,Q}) = nD_Q.$$

Then the Weil pairing can be written on the form

$$e_n(P, Q) = \frac{f_{n,P}(D_Q)}{f_{n,Q}(D_P)}.$$

The proof of this theorem is quite technical and can be found in Washington, [Was08, p. 371]. Note that the form found in Washington yields the inverse of the above fraction [Was08, remark 11.13]. This is not a problem since we map into a set of roots of unity in the same way just hitting the inverse of the roots preserving all structure.

When we want to compute the Weil pairing, we need to have it as an expression in points P and Q.

Corollary 3.27. *Suppose that a point $T \notin \{P, Q, Q - P, \mathcal{O}\}$ is given. Let $D_P = [P + T] - [T]$, $D_Q = [Q] - [\mathcal{O}]$ and let $f_{n,P}$ and $f_{n,Q}$ be functions s.t.*

$$div(f_{n,P}) = nD_P \text{ and } div(f_{n,Q}) = nD_Q.$$

Then

$$e_n(P, Q) = \frac{f_{n,Q}(T)}{f_{n,P}(-T)} \frac{f_{n,P}(Q - T)}{f_{n,Q}(P + T)}.$$

Proof. By Theorem 3.14 there exists a function f_{helper} such that

$$div(f_{helper}) = n[P + T] - n[T].$$

Since we chose $T \neq P, Q, Q - P, \mathcal{O}$ then D_P and D_Q do not share any points and from Theorem 3.26 we may write the Weil pairing

$$e_n(P, Q) = \frac{f_{helper}([Q] - [\mathcal{O}])}{f_{n,Q}([P + T] - [T])}.$$

From Definition 3.19 we may expand this to

$$\begin{aligned} e_n(P, Q) &= \frac{f_{helper}(Q) f_{helper}(\mathcal{O})^{-1}}{f_{n,Q}(P + T) f_{n,Q}(T)^{-1}} \\ &= \frac{f_{helper}(Q) f_{n,Q}(T)}{f_{n,Q}(P + T) f_{helper}(\mathcal{O})}. \end{aligned}$$

Observation 3.28. *Let τ_{-T} be the translation with $-T$ then*

$$div(f_{helper}) = n[P+T] - n[T] = div(f_{n,P} \circ \tau_{-T}),$$

but then we know that for some constant γ:

$$f_{helper} = \gamma \cdot (f_{n,P} \circ \tau_{-T}).$$

When we insert this expression for f_{helper} into the expression for e_n, γ divides out and

$$e_n(P,Q) = \frac{f_{n,Q}(T)}{f_{n,P}(-T)} \frac{f_{n,P}(Q-T)}{f_{n,Q}(P+T)}.$$

□

We now see how the pairing can be evaluated if we have a way of evaluating functions $f_{n,P}$ where $div(f_{n,P}) = n[P] - n[\mathcal{O}]$ in points $R \neq P$.

Miller showed that we actually can evaluate $f_{n,P}$ recursively never having to write up the function itself. Construct a recursive function $f_{j,P}$ such that

$$div(f_{j,P}) = j[P] - [jP] - (j-1)[\mathcal{O}] \text{ for } j < n.$$

We see that when $j = n$ the above form produces the correct divisor $div(f_{n,P}) = n[P] - n[\mathcal{O}]$.

Theorem 3.29 (Miller's formula)**.** *Let $P, Q \in E$ and define for $j > 0$*

$$f_{j+1,P} := f_{j,P} g_{P,jP} \text{ and } f_{0,P} := 1, f_{1,P} := 1, \tag{3.3}$$

where the function $g_{P,Q}$ is the function defined in Example 3.12. Then

$$\begin{aligned} div(f_{j,P}) &= j[P] - [jP] - (j-1)[\mathcal{O}], &(3.4)\\ div(f_{j+k,P}) &= div(f_{j,P} f_{k,P} g_{jP,kP}) &(3.5) \end{aligned}$$

Proof. First use the principle of induction to prove (3.4) for all j:

Induction start: Validate (3.4) for both $j = 0, 1$.

$$\begin{aligned} div(f_{0,P}) &= 0[P] - [0P] - (-1)[\mathcal{O}] = -[\mathcal{O}] + [\mathcal{O}] = 0\\ div(f_{1,P}) &= [P] - [P] - (1-1)[\mathcal{O}] = 0, \end{aligned}$$

which is correct since $f_{0,P} = f_{1,P} = 1$ is constant.

Induction hypothesis: Assume $div(f_{i,P}) = i[P] - [iP] - (i-1)[\mathcal{O}]$ for $i \leq j$.

Induction step: Now show $div(f_{(j+1),P}) = (j+1)[P]-[(j+1)P]-(j+1-1)[\mathcal{O}]$ by direct computation using my induction hypothesis:

$$\begin{aligned}
div(f_{(j+1),P}) &= div(f_{j,P}g_{P,jP}) \\
&= div(f_{j,P}) + div(g_{P,jP}) \\
&= j[P] - (j-1)[\mathcal{O}] - [jP] + [P] + [jP] - [P + jP] - [\mathcal{O}] \\
&= (j+1)[P] - ((j+1) - 1)[\mathcal{O}] - [(j+1)P].
\end{aligned}$$

Next show the identity (3.5) by direct computation starting backwards

$$\begin{aligned}
div(f_{j,P}f_{k,P}g_{jP,kP}) &= div(f_{j,P}) + div(f_{k,P}) \\
&\quad +div(L_{jP,kP}) - div(L_{(j+k)P,-(j+k)P}) \\
&= (j[P] - [jP] - (j-1)[\mathcal{O}]) \\
&\quad +(k[P] - [kP] - (k-1)[\mathcal{O}]) \\
&\quad +([jP] + [kP] + [-(j+k)P] - 3[\mathcal{O}]) \\
&\quad -([(j+k)]P + [-(j+k)P] + [\mathcal{O}] - 3[\mathcal{O}]) \\
&= (j+k)[P] - [(j+k)P] - (j+k-1)[\mathcal{O}] \\
&= div(f_{j+k,P}).
\end{aligned}$$

□

Remark 3.30. *Setting*

$$f_{j+k,P} := f_{j,P}f_{k,P}g_{jP,kP} \tag{3.6}$$

in the above, preserves the divisor, thus in the case where $j = k$ we can write:

$$f_{2j,P} = f_{j,P}^2 g_{jP,kP}. \tag{3.7}$$

We are now ready to present a double and add version of Millers algorithm for calculating the value $f_{n,P}$ as Algorithm 3.4.1.

In Algorithm 3.4.1 we see how we can use the above formulas (3.6) and (3.7) to double and add up to the value $f_{n,P}(Q)$.

The following form of the Weil pairing is good since it saves us half the calculations in the case where P and Q are in the curve group $E(K)[n]$.

Theorem 3.31. *Let E/K be an elliptic curve, let $P, Q \in E(K)[n]$, and let $P \neq Q$. Then*

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

Algorithm 3.4.1: Millers algorithm using double-and-add

Data: elliptic curve E/K, points $P, Q \in E(K) \setminus \{\mathcal{O}\}$, positive integer $n = \sum_{j=0}^{\log n} b_i 2^i$
Result: value $t \in \mathbb{Z}_n$
$t \leftarrow 1$
$V \leftarrow P$
$i \leftarrow \lceil \log n \rceil - 2$
while $i > -1$ **do**
    $t \leftarrow t^2 \cdot g_{V,V}(Q)$
    $V \leftarrow 2V$
    if $b_i = 1$ **then**
        $t \leftarrow t \cdot g_{V,P}(Q)$
        $V \leftarrow V + P$
    $i \leftarrow i - 1$
return t

Intuitively the short form seems correct for $T \to \mathcal{O}$ in the form in Corollary 3.27. This will not be proved rigorously, but a proof can be found in the referenced article by Miller [Mil04]. It should be noted that it is still important that the support of divisors are different i.e. P and Q are linearly independent. In practice if they are not, there will likely be a division with zero in the Algorithm 3.4.1.

Example 3.32 (Weil pairing example)**.** *In this example I will consider the elliptic curve group $E(\mathbb{F}_{2^7})$ where*

$$E : y^2 + y = x^2 + x + 1.$$

We first compute the cardinality of this small curve group.

```
sage: F1.<a>=GF(2^7)
sage: E1=EllipticCurve(F1,[0,0,1,1,1])
sage: E1.cardinality()
113
```

Since 113 is prime then $E(\mathbb{F}_{2^7}) \simeq C_{113}$ is cyclic. So every point in this group is linearly dependent of the other. Thus the Weil pairing of two arbitrary points $P, Q \in E(\mathbb{F}_{2^7})$ will be trivial by Remark 3.2. To get a non-trivial Weil pairing we want to use Theorem 3.24, but then we will need to determine the smallest $k > 1$ (embedding degree) such that $113 \mid (2^{7k} - 1)$, i.e the smallest k such that the whole torsion group $E[113] \subset E(\mathbb{F}_{2^{7k}})$. We try $k = 4$.

```
sage: F2.<b>=GF(2^28)
sage: E2=EllipticCurve(F2,[0,0,1,1,1])
```

```
sage: factor(E2.cardinality())
5^2 * 29^2 * 113^2
```

So by Theorem 3.24 there exist points Q and P in $E(\mathbb{F}_{2^{28}})$ yielding a non-trivial pairing value.

I choose the linear independent points $P, Q \in E(\mathbb{F}_{2^{28}})[113]$, see Appendix F.4 for Sage code containing points P and Q.

```
sage: load weil_pairing_example.sage
sage: P.weil_pairing(Q,113)
b^25 + b^17 + b^14 + b^11 + b^10 + b^4
sage: P.weil_pairing(Q,113)^113
1
```

It is important for the practicality of the signature scheme that the Weil pairing can be computed in reasonable amount of time. The next theorem states that the time it takes to do a pairing is linear in the bit size of the input n.

Theorem 3.33. *Let $P, Q \in E[n]$ then the Weil pairing $e_n(P, Q)$ can be efficiently calculated in linear time*

$$O(C(\mathbb{F}_{q^k})\log(n)),$$

for a constant $C(\mathbb{F}_{q^k})$ dependent on the field operations in $\mathbb{F}_{q^k}$.

Proof. I start by proving the correctness of the algorithm. Algorithm 3.4.1 returns $t = f_{n,P}(Q)$. By Formula 3.5 the divisor is preserved up until you reach n in the double and add process. When n is reached Formula 3.5 gives

$$div(f_{n,P}) = n[P] - [nP] - (n-1)[\mathcal{O}] = n[P] - n[\mathcal{O}],$$

since $P \in E[n]$. We have shown that Algorithm 3.4.1 returns $t = f_{n,P}(Q)$ where $div(f_{n,P}) = n[P] - n[\mathcal{O}]$.

Next we prove that the running time of the algorithm is in $O(C(\mathbb{F}_{q^k})\log(n))$. In the worst case, the algorithm will in each while-loop visit the if-statement and have to evaluate the function g. I may assume that evaluating g takes some constant amount of time $C(\mathbb{F}_{q^k})$ dependent on the field $\mathbb{F}_{q^k}$. So this takes $C(\mathbb{F}_{q^k}) \cdot \log n$ time, and we have to run the algorithm four times to calculate the Weil pairing value, i.e. $4C(\mathbb{F}_{q^k})\log n \in O(C(\mathbb{F}_{q^k})\log n)$. □

$\log_2(n)$ (bits)	$H(n)$	Weil comp. (s)	no. mul.	total (ms)	no. div.	total (ms)
81	45	9.78	6080	2157	989	5031
54	32	6.56	4064	1449	669	3405
40	12	3.92	2780	899	397	1984
30	13	3.20	1338	719	325	1643
20	8	2.0	1388	454	205	1026
158	84	2.4	38287	361	1917	468

Table 3.1: Timing of Weil pairing for different sized subgroups of elliptic curve group $E_{3,2}(3^{42}) : y^2 = x^3 + x + 2$.

3.4.1 Implementation of the Weil pairing

The above Algorithm 3.4.1 has been written into the Sage open source project and released with version 3.3, see Appendix F.3 for code. There is a note to be made as an extension of the above discussion on division with zero in the case of linearly dependent points. Remember that when P, Q are linearly dependent the pairing value $e_n(P, Q) = 1$, so in practice the pairing computation in Theorem 3.31 has been implemented in a try-catch statement. From a performance perspective on general input, this makes us in the worst case run the whole Miller algorithm in cases which just evaluate to 1. In the short signature scheme we will work with linear independent points, so in this context we really don't have to worry about this aspect.

The Weil pairing implementation was profiled (intel core 2.4 dual processor system $\sim$ single 1.2 GHz processor) using the `prun` function in Sage, and some observations is found in Table 3.1.

It should be noted, that the Weil pairing implementation is significantly faster on elliptic curves over large characteristic fields $F(\mathbb{F}_{p^k})$ in Sage[2]. There is included an extra row in the table with timing of a weil pairing of point on an elliptic curve over a large characteristic field extension. The elliptic curve used for the large prime characteristic is included as a Sage sample in Appendix F.5.

We confirm from the times in the table, that number of multiplications and divisions very much depend on the Hamming weight of n (notice the 40 bit and 30 bit cases in the table). This complies with having to do more add

[2]From inspecting the PARI implementation it has since been discovered that the irreducible polynomial produced for defining the finite fields was very dense, which has some impact on the performance. Though it still does not account for the large gap in performance in arithmetic over small and large characteristic fields in Sage.

operations along the way with respect to the higher Hamming weight. Also notice that the time it takes to do divisions in a finite field in Sage is approx. 16 times greater than the time it takes to do finite field multiplications. So it could be worth trying to save divisions in the implementation and if possible use a low hamming weighted n.

The timing for small bitsizes of n seems linearly dependent as the Theorem 3.33 states it should be. We will not go further with this observation, but it could be interesting to verify the linear relation by using linear regression analysis.

Chapter 4

The Menezes, Okamoto, Vanstone reduction

In this section the MOV reduction will be described and implemented. The Menezes, Okamoto and Vanstone [MOV91] reduction is a method of reducing the discrete logarithm problem in elliptic curve groups to the discrete logarithm problem in a finite field. In the finite field there are currently more efficient algorithms for solving the discrete logarithm problem than on the curve.

First we need to show a one-to-one correspondance between points on an elliptic curve and finite field elements.

Theorem 4.1. *Let E be an elliptic curve defined over a finite field $\mathbb{F}_q$. Let P have order n and generate the subgroup $\langle P \rangle$ of $E(\mathbb{F}_q)$. Let Q be a point in $E[n]$ such that $e_n(P, Q)$ is a primitive n'th root of unity.*

Let $\varphi : \langle P \rangle \to \mu_n$ be a function where

$$\varphi : R \mapsto e_n(R, Q).$$

Then φ is an isomorphism.

Proof. By the bilinearity of e_n in the first variable

$$e_n(R_1 + R_2, Q) = e_n(R_1, Q)e_n(R_2, Q),$$

φ is a homomorphism. φ is surjective since $Q \neq \mathcal{O}$ is fixed and for $P_1 \neq P_2$ the pairings $e(P_1, Q) \neq e(P_2, Q)$. Consider the kernel of the map φ, i.e for an $0 \leq l < n$ the points $R' = l \cdot P$ such that $\varphi(R') = 1$.

$$1 = \varphi(R') = e_n(R', Q) = e_n(l \cdot P, Q) = e_n(P, Q)^l = \xi^l,$$

where ξ is a primitive n'th root of unity. So $n \mid l$ but we have chosen $0 \leq l < n$ and thus $l = 0$. The kernel of φ is trivial, so by Noether's first isomorphism theorem and surjectiveness of the φ we have that $\langle P \rangle \simeq \mu_n$, which concludes the proof. □

Let the discrete logarithm problem on the elliptic curve subgroup $\langle P \rangle$ be given as $R \in \langle P \rangle, Q \in E[n]$ and $R = l \cdot P$, for $0 < l < n - 1$ and set $\alpha = e_n(P, Q)$ and $\beta = e_n(R, Q)$. Then from the one-to-one correspondance f in the above theorem there will be exactly one value l' such that $\alpha^{l'} = \beta$. But

$$\alpha^{l'} = \beta = e_n(l \cdot P, Q) = e_n(P, Q)^l = \alpha^l,$$

so $l = l'$.

This shows that we can reduce the problem of finding the discrete logarithm in the elliptic curve group to the problem of finding the discrete logarithm in the group of n'th roots of unity. We will need to determine a linearly independent point $Q \in E[n]$ and thus by Theorem 3.24 the smallest k s.t. $E[n] \subset E(\mathbb{F}_{q^k})$. The value k should be as small as possible such that the field $\mathbb{F}_{q^k}$ does not get bigger than necessary. This k is also known as the embedding degree.

The embedding degree is also referred to as the security multiplier [BLS04] and is defined in the following way.

Definition 4.2. *Let $P \in E(\mathbb{F}_q)$ be a point of prime order n. The subgroup generated by P has embedding degree $k > 0$ if $n \mid q^k - 1$ and $n \nmid q^i - 1$ for $0 < i < k$.*

The embedding degree dictates how large the field extension $\mathbb{F}_{q^k}$ is, where computations for determining the Weil pairing value are performed. Thus to efficiently compute the pairing, k should be controlled. An arbitrary curve has with high probability a large embedding degree $k > (\log p)^2$ [BK98]. So we need to choose the curve such that we can control the embedding degree. For this pupose supersingular curves are considered.

4.1 Supersingular elliptic curves

An elliptic curve is said to be supersingular over a finite field $\mathbb{F}_q$ of characteristic p when the p-torsion group is trivial $E[p] \simeq \{\mathcal{O}\}$ [Was08, p.79]. The following theorem makes it easy to determine whether a curve is supersingular.

Theorem 4.3. *Let E be an elliptic curve over the finite field $\mathbb{F}_q$ with characteristic p. Say that $E(\mathbb{F}_q)| = q + 1 - t$. Then E is supersingular if and only if the cardinality $|E(\mathbb{F}_q)| \equiv 1 \pmod{p}$ or equivalently if $t \equiv 0 \pmod{p}$.*

The proof of the above theorem can be found in Washington p. 130 [Was08].

It can be shown that supersingular elliptic curves can be divided into six classes, see Appendix D, and the embedding degree can be determined for each class. The following shows that the embedding degree for curve classes IV and V is $k = 4$ and $k = 6$ with respect to fields $\mathbb{F}_{2^e}$ and $\mathbb{F}_{3^e}$.

Lemma 4.4. *The embedding degree of subgroups of elliptic curve groups in class IV is $k \leq 4$.*

Proof. We show for cardinality m of $E(\mathbb{F}_q)$, $m \mid q^4 - 1$. Every subgroup, which cardinality is a divisor in m, will have embedding degree $k \leq 4$. We know from Table D.1 that the curve group $E(\mathbb{F}_q)$ has cardinality $m = q + 1 \pm \sqrt{2q}$. We now compute

$$
\begin{aligned}
(q^2 + 1) =& (q + 1 + \sqrt{2q})(q + 1 - \sqrt{2q}) \\
(q^4 - 1) =& (q^2 + 1)(q^2 - 1).
\end{aligned}
$$

The computation shows that m divides $(q^4 - 1)$. □

Lemma 4.5. *The embedding degree of subgroups of elliptic curve groups in class V is $k \leq 6$.*

Proof. Let m be the cardinality of $E(\mathbb{F}_q)$. We want to show that $m \mid q^6 - 1$. Every subgroup, which cardinality is a divisor in m, will have embedding degree $k \leq 6$. we know from Table D.1 that the elliptic curve group $E(\mathbb{F}_q)$ has cardinality $m = q + 1 \pm \sqrt{3q}$. We now compute

$$
\begin{aligned}
(q^2 - q + 1) =& (q + 1 + \sqrt{3q})(q + 1 - \sqrt{3q}) \\
(q^4 + q^2 + 1) =& (q^2 - q + 1)(q^2 + q + 1) \\
q^6 - 1 =& (q^4 + q^2 + 1)(q^2 - 1).
\end{aligned}
$$

The computation shows that m divides $(q^6 - 1)$. □

Theorem 4.6. *The embedding degree for subgroups of a supersingular elliptic curve E*

- *in class IV over a finite field of characteristic 2 is $k_2 = 4$*
- *in class V over a finite field of characteristic 3 is $k_3 = 6$.*

Proof. Using Lemma 4.4 and Lemma 4.5 it's enough to show that $k_2 \geq 4$ and $k_3 \geq 6$. We can do this using Euclid's algorithm.

Claim 4.7. $k_2 \geq 4$.

Proof. The cardinality $|E(\mathbb{F}_{2^e})| = 2^e + 1 \pm \sqrt{2^{e+1}}$ and

$$(2^{2e} + 1) = (2^e + 1 + \sqrt{2^{e+1}})(2^e + 1 - \sqrt{2^{e+1}}).$$

So we check for all divisors d in $(2^{2e} + 1)$ that $d \nmid 2^{ie} - 1$ for $i = 1, 2, 3$ in reverse order.

$$\begin{aligned}\gcd(2^{3e} - 1, 2^{2e} + 1) &= \gcd(2^{2e} + 1, -2^e - 1)\\ &= \gcd(-2^e - 1, 2)\\ &= 1.\end{aligned}$$

$$\begin{aligned}\gcd(2^{2e} - 1, 2^{2e} + 1) &= \gcd(2^{2e} + 1, 2)\\ &= 1.\end{aligned}$$

$$\begin{aligned}\gcd(2^{2e} + 1, 2^e - 1) &= \gcd(2^e - 1, 2)\\ &= 1.\end{aligned}$$

This means that $k_2 \geq 4$. □

Claim 4.8. $k_3 \geq 6$.

Proof. The cardinality $|E(\mathbb{F}_{3^e})| = 3^e + 1 \pm \sqrt{3^{e+1}}$ and

$$(3^{2e} - 3^e + 1) = (3^e + 1 + \sqrt{3^{e+1}})(3^e + 1 - \sqrt{3^{e+1}}).$$

We now check for all divisors d in $(3^{2e} - 3^e + 1)$ that $d \nmid 3^{ie} - 1$ for $i = 1, \ldots, 5$

in reverse order.

$$\begin{aligned}\gcd(3^{5e}-1,3^{2e}-3^e+1) &= \gcd(3^{2e}-3^e+1,-3^e)\\ &= 1.\end{aligned}$$

$$\begin{aligned}\gcd(3^{4e}-1,3^{2e}-3^e+1) &= \gcd(3^{2e}-3^e+1,-3^e-1)\\ &= \gcd(-3^e-1,3)\\ &= \gcd(3,2)\\ &= 1.\end{aligned}$$

$$\begin{aligned}\gcd(3^{3e}-1,3^{2e}-3^e+1) &= \gcd(3^{2e}-3^e+1,-2)\\ &= 1.\end{aligned}$$

$$\begin{aligned}\gcd(3^{2e}-1,3^{2e}-3^e+1) &= \gcd(3^{2e}-3^e+1,3^e-2)\\ &= \gcd(3^e-2,3)\\ &= \gcd(3,1) &= 1.\end{aligned}$$

$$\gcd(3^{2e}-3^e+1,3^e-1) = 1.$$

This means that $k_3 \geq 6$. □

By Lemma 4.4 and Lemma 4.5 together with the above claims the theorem is proved. □

Example 4.9. *Let us shortly discuss the rationale for choosing $k = 4$ in Example 3.32. We can verify that the curve is supersingular using Theorem 4.3 by checking that*

$$|E(\mathbb{F}_q)| \equiv 1 \pmod 2.$$

In fact this curve is a class IV curve by Theorem D.1 and for the curves in this class it was shown in Theorem 4.6 that the embedding degree $k = 4$. So we have now shown $k = 4$ was indeed a rational choice.

4.2 Embedding of points

We need to treat the practical problem of embedding points from $E(\mathbb{F}_q)$ into $E(\mathbb{F}_{q^k})$ when $q = p^e$. Let α generate the field $\mathbb{F}_q$ and let $A(x)$ be the minimal polynomial of α. Let β generate the extension field $\mathbb{F}_{q^k}$ and let $B(x)$ be the minimal polynomial of β. Note that $A(x)$ will have roots (split) in $\mathbb{F}_{q^k}$. Now consider the embedding

$$\Phi : \mathbb{F}_q \to \mathbb{F}_{q^k}, \text{ by } \alpha \mapsto \bar{\alpha},$$

where $\bar{\alpha}$ is a root of $A(x)$ over $\mathbb{F}_{q^k}$. So embedding a point (x, y) from $E(\mathbb{F}_q)$ into $E(\mathbb{F}_{q^k})$ is then done in the straight forward way $(x, y) \mapsto (\Phi(x), \Phi(y))$.

The embedding will preserve group structure on points and given a point P generating a group $\langle P \rangle$ the embedded point $\Phi(P)$ will generate an isomorphic group $\langle \Phi(P) \rangle \simeq \langle P \rangle$.

Note that in $q = p^e$ if $e = 1$ then Φ is just the identity map.

Example 4.10 (Point embedding)**.** *In this example we consider the elliptic curve used in Example 3.32. It was shown in Example 4.9 that the embedding degree is $k = 4$. Sage has a built in function, as many other math software packages have as well, that can define a homomorphism between two objects, in this case for fields:*

```
sage: P1=E1.random_point()
sage: P1.order()
113
sage: aa=F1.modulus().roots(F2)[0][0]
sage: aa in F2
True
sage: phi=Hom(F1,F2)(aa)
sage: phi
Ring morphism:
   From:  Finite Field in a of size 2^7
   To:    Finite Field in b of size 2^28
   Defn:  a |--> b^23 + b^22 + b^20 + b^19 + b^17 + ...
sage: P2=E2(phi(P1.xy()[0]),phi(P1.xy()[1]))
sage: P2 in E2
True
Sage: P2.order()
113
```

4.3 Reduction in the supersingular curve case

In this section we will look at the MOV reduction on supersingular elliptic curves. We will start by showing that the MOV reduction in Algorithm 4.3.1 is effective. Note that for the MOV attack to be effective, we require to know the parameters k, c and n_1, since there is no fast way of directly computing these.

Theorem 4.11. *Let E be a supersingular elliptic curve over the field $\mathbb{F}_q$. Let $P \in E(\mathbb{F}_q)$ with order n, let $R \in \langle P \rangle$ and l be an integer such that $P = l \cdot R$. Let k be the extension degree of $\mathbb{F}_q$ so $E[n] \subset E(\mathbb{F}_{q^k})$. There exist a probabalistic polynomial time reduction of the DLog problem in $E(\mathbb{F}_q)$ to the DLog problem in $\mathbb{F}_{q^k}$.*

Algorithm 4.3.1: MOV reduction for supersingular curves

```
Data: supersingular curve E/F_q, points P ∈ E(F_q) and R ∈ ⟨P⟩
Result: the discrete logarithm l of R to the base P
Look up k, c and n_1 in a table
t ← n
while t > 0 do
    Q' ←R E(F_{q^k});        /* R: random element is assigned */
    Q ← (c n_1 / n) · Q';     /* such that Q get order n */
    α ← e_n(P, Q)
    β ← e_n(R, Q)
    l' ← log_α β
    if l' · P = R
    then
        return l'
        t ← 0
    t ← t − 1
```

Proof. We may assume that arithmetic in $\mathbb{F}_{q^k}$ takes some constant amount of time M if we are given an irreducible polynomial defining the field. We pick the point Q' and calculate Q in polynomial time $O(M \log \frac{cn_1}{n})$.

Elements α and β are computed using Miller's Algorithm 3.4.1 for computing the Weil pairing in time $O(\log n)$.

The probability p to find a $Q \in E[n]$ is the number of elements of order n in $\mathbb{F}_{q^k}$ divided by n

$$p = \frac{\phi(n)}{n}$$

So we expect to iterate $t = \frac{n}{\phi(n)}$ times. It can be shown that $t \leq 6 \ln \ln n$ for $n \geq 5$ [MOV91].

Note that if the order of α is n, then the order of β is a divisor d in n. We may assume that $d = n$, otherwise we can run the algorithm with $n/d \cdot P$ instead of P.

The statement $l' \cdot P = R$ can also be checked in polynomial time $O(M \log l')$ for $l' \leq n$. Summing up we get

$$\left[O(\log \frac{cn_1}{n}) + O(\log n) + O(\log l') \right] O(\ln \ln n) = O(\log n) \sim O(\log q)$$

□

Example 4.12 (MOV reduction)**.** *In this example we consider the elliptic curve used in Example 3.32. It was shown that the embedding degree is $k = 4$. Select points $P, R \in E(\mathbb{F}_{2^7})$ such that $P = l \cdot R$ for some integer $0 < l < 113$. Use an embedding ϕ to map the points into $E(\mathbb{F}_{2^{28}})$. In Example 4.10 there is defined such an embedding in Sage. Start by loading the points appended as Sage code in Appendix F.6.*

```
sage: load mov_reduction_example.sage
sage: P1=E1.random_point()
sage: R1=45*P1
sage: P2=E2(phi(P1.xy()[0]),phi(P1.xy()[1]))
sage: R2=E2(phi(R1.xy()[0]),phi(R1.xy()[1]))
```

Now we choose a random point $Q' \in E(\mathbb{F}_{2^{28}})$. To get a point in $E[113]$ we look up cn_1 in the Table D.1 and multiply Q' with

$$\frac{cn_1}{n} = \frac{q^2+1}{113} = \frac{2^{14}+1}{113} = 145.$$

We can now pair P, Q and R, Q to get the 113'th roots of unity and solve the discrete logarithm in these. We do this using Sage

```
sage: Q=145*E2.random_point()
sage: alpha=P2.weil_pairing(Q)
sage: beta=R2.weil_pairing(Q)
sage: beta.log(alpha)
45
```

CHAPTER 5

co-GDH groups from the Weil pairing

In this section we show how one can use the Weil pairing to obtain a co-GDH group pair from subgroups of elliptic curve groups. The elliptic curves we look at are supersingular curves over a finite field with low characteristic. We will see that this choice will have an effect on how difficult it is to break co-CDH. This will be discussed in the end of this section.

Let $\langle P \rangle \in E(\mathbb{F}_q)$ be the subgroup generated by a point P of prime order n such that $n \nmid q$ and $n^2 \nmid |E(\mathbb{F}_q)|$, i.e. $\langle P \rangle$ is the only order n subgroup in this curve group. Also let the embedding degree of $\langle P \rangle$ be $k > 1$. From Theorem 3.24 we know that there exist a point, linearly independent of P in $E(\mathbb{F}_{q^k})$, which also generates an order n subgroup.

We want to show that $\langle P \rangle$ and $\langle Q \rangle$ make a (τ, t, ε)-co-GDH group pair. By Definition 1.7 we need to show:

- Group operations in $\langle P \rangle$ and $\langle Q \rangle$ are done in time at most τ.
- There exist an isomorphism $\psi : \langle Q \rangle \to \langle P \rangle$ and ψ can be computed in time at most τ.
- The co-DDH problem on $(\langle P \rangle, \langle Q \rangle)$ can be solved in at most time τ.
- No algorithm (t, ε)-breaks co-CDH on $(\langle P \rangle, \langle Q \rangle)$.

5.1 Efficiently computable group isomorphism

Using the double and add formula for points on elliptic curves and assuming that finite field operations in $E(\mathbb{F}_{q^k})$ takes a constant amount of time, then group operations will take time polynomial in $O(k \log q)$.

An efficient computable isomorphism $\psi : G_2 \to G_1$ is required. The following theorem [BLS04] shows that we can extend the trace map to elliptic curve groups and use this as the isomorphism ψ. We define the trace of a point on an elliptic curve $E(\mathbb{F}_{q^k})$ in the following way.

Definition 5.1. *Define the trace on elliptic curve groups in $E(\mathbb{F}_{q^k})$ as the map $tr : E(\mathbb{F}_{q^k}) \to E(\mathbb{F}_q)$,*

$$tr : P \mapsto \sum_{i=0,\ldots,k-1} \sigma_i(P),$$

where $\sigma_i(P) = \left(x(P)^{q^i}, y(P)^{q^i}\right)$ for $P \in E(\mathbb{F}_{q^k})$.

We see from the above definition that the time it takes to compute the trace map on elliptic curves is k times the time it takes to power finite field elements in $\mathbb{F}_{q^k}$. If a square and add algorithm is used, we get a total time $\tau \in O(k^2 \log q)$.

Next we show that the above trace map can be used as an isomorphism between $\langle P \rangle$ and $\langle Q \rangle$.

Theorem 5.2. *Let $P \in E(\mathbb{F}_q)$ be a point of prime order $n \neq q$ and let $\langle P \rangle$ have embedding degree $k > 1$. Let $Q \in E(\mathbb{F}_{q^k})$ also have order p and be linearly independent of the point P. If $tr(Q) \neq \mathcal{O}$ then the map tr is an isomorphism from $\langle Q \rangle$ to $\langle P \rangle$.*

Proof. We begin with a claim on the order n points in $E(\mathbb{F}_q)$.

Claim 5.3. *All points in $E(\mathbb{F}_q)$ of order n are contained in $\langle P \rangle$.*

Proof. Assume for contradiction that an arbitrary point $R \in E(\mathbb{F}_q)$ have order n and $R \notin \langle P \rangle$. Then $\{P, R\}$ spans $E[n]$. Thus the whole of $E[n] \subset \mathbb{F}_q$, but we assumed that the embedding $k > 1$, which gives us the wanted contradiction. □

The σ_i's are automorphisms and thus field homomorphisms. They preserve point additions and scalings, since these consist only of additions and powering of different field elements. So we can derive

$$\begin{aligned} n \cdot tr(Q) &= \sum_{i=1,\dots,k-1} n\sigma_i(Q) \\ &= \sum_{i=1,\dots,k-1} \sigma_i(nQ) \\ &= \sum_{i=1,\dots,k-1} \sigma_i(\mathcal{O}) \\ &= \mathcal{O}, \end{aligned}$$

since we assumed $Q \in E[n]$ and that the automorphisms fix the point at infinity $\mathcal{O} \in E(\mathbb{F}_q)$. From the assumption $tr(Q) \neq \mathcal{O}$ and the above result we have that $tr(Q)$ have order n. By the claim $tr(Q) \in \langle P \rangle$. Next observe that for $Q_1, Q_2 \in E(\mathbb{F}_{q^k})$

$$\begin{aligned} tr(Q_1 + Q_2) &= \sum_{i=1,\dots,k-1} \sigma_i(Q_1 + Q_2) \\ &= \sum_{i=1,\dots,k-1} (\sigma_i(Q_1) + \sigma_i(Q_2)) \\ &= \sum_{i=1,\dots,k-1} \sigma_i(Q_1) + \sum_{i=1,\dots,k-1} \sigma_i(Q_2) \\ &= tr(Q_1) + tr(Q_2), \end{aligned}$$

which shows that the trace map on the elliptic curve is a homomorphism. Now look at the kernel of tr, i.e. the $Q' = l \cdot Q$ for some $0 \leq l < n$ such that $tr(Q') = \mathcal{O}$. We just saw that the trace map was a homomorphism so

$$\mathcal{O} = tr(Q') = tr(l \cdot Q) = ltr(Q)$$

using our assumption. Since $tr(Q) \in \langle P \rangle$, $tr(Q)$ has order n, so $n \mid l$ and thus $l = 0$. We have thereby shown that the kernel $\ker(tr) = \mathcal{O}$ is trivial.

We can now show that the map is injective. Take two points $Q_1, Q_2 \in \langle Q \rangle$ where

$$\begin{aligned} tr(Q_1) &= P_0 \\ tr(Q_2) &= P_0, \end{aligned}$$

for some $P_0 \in \langle P \rangle$. Then $tr(Q_1 - Q_2) = \mathcal{O}$ and $Q_1 - Q_2$ must be in the kernel of tr which we just showed to be trivial. Thus $Q_1 = Q_2$ i.e. the map is injective.

But since there are n elements in both $\langle Q \rangle$ and $\langle P \rangle$ the map is surjective. So in conclusion the trace map is a bijective homomorphism or an isomorphism.

$\square$

5.2 Tractability of DDH problem

Property 3 requires the co-DDH problem to be easy to solve on the group pair $(\langle P\rangle, \langle Q\rangle)$. To show this, use the Weil pairing and the following theorem due to Joux and Nguyen [JN03].

Theorem 5.4 (Joux and Nguyen)**.** *Let the tuple (g_2, g_2^a, h, h^b) be the one given as the premise of the co-DDH problem on an order n group pair $(\langle P\rangle, \langle Q\rangle)$. Let e_n be the Weil pairing then*

$$a \equiv b \pmod{n} \text{ if and only if } e_n(h, g_2^a) = e_n(h^b, g_2).$$

Proof. The theorem follows from the bilinearity of the map e_n. Assume that $a \equiv b \pmod{n}$ then

$$e_n(h, g_2^a) = e_n(h.g2)^a = e_n(h, g_2)^b = e_n(h^b, g_2).$$

Assume that $e_n(h, g_2^a) = e_n(h^b, g_2)$ then

$$e_n(h, g_2)^a = e_n(h, g_2^a) = e_n(h^b, g_2) = e_n(h, g_2)^b,$$

and since $e_n(h, g_2) \in \mu_n$ we have that

$$a \equiv b \pmod{n}.$$

□

We can efficiently compute the two pairings with Miller's algorithm

$$e_n(h, g_2^a) \text{ and } e_n(h^b, g_2)$$

and check whether they are equal in time $O(\log q)$. So in this setting the co-DDH problem is solvable in time $\tau \in O(\log q)$.

5.3 Intractability of CDH problem

The last property, the group pair needs to fulfill, is that no algorithm can (t, ε)-break co-CDH on $(\langle P\rangle, \langle Q\rangle)$. cannot show this explicitly. Instead we will discuss when the co-CDH problem currently thought to be intractable on $(\langle P\rangle, \langle Q\rangle)$.

The co-CDH property can be reduced to the problem of computing the discrete logarithm in $\langle P\rangle$ and $\langle Q\rangle$. We will discuss two ways of computing the discrete logarithm on elliptic curve groups: using generic group algorithms

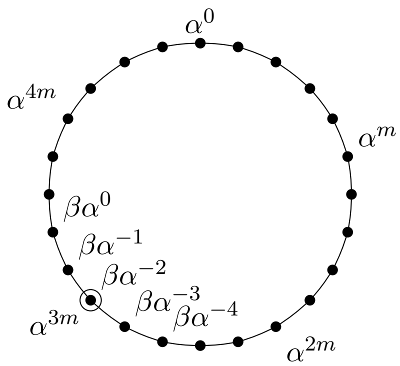


Figure 5.1: Shanks' baby-step giant-step algorithm graphically

or doing a reduction from the curve group to a finite field and then compute the logarithm there.

We should note that in this thesis we will only consider the MOV reduction, while there in reality is other reductions that need to be taken into account such as Weil decent [Fre99]. But that is outside the scope of this thesis.

5.3.1 Generic discrete logarithm algorithms

In this section we review some different non-trivial discrete logarithm algorithms on generic groups. The main reference for this section is Stinson [Sti05].

Shanks' baby-step giant-step method

We look at the discrete logarithm

$$a = \log_\alpha \beta, \text{ for } \alpha, \beta \in G \text{ (cyclic of order } n).$$

Observe that the discrete logarithm $0 \leq a \leq n-1$. Let $m = \lceil \sqrt{n} \rceil$ and write

$$a = mj + i, \quad 0 \leq j, i \leq m - 1.$$

To determine the discrete logarithm a we need to find i, j such that

$$\alpha^{mj+i} = \beta \text{ or } \alpha^{mj} = \beta\alpha^{-i}$$

Then we can compute the discrete logarithm $a = mj + i$. To find the pair i, j we look at a baby-step sequence

$$L_1 = [\beta\alpha^{-i}]_{i=0,\ldots,m-1}$$

and a giant-step sequence

$$L_2 = [\alpha^{mj}]_{j=0,\dots,m-1}$$

and search for the pair i, j that satisfies the above equality.

An example of the algorithm is given graphically in Figure 5.1 with $n = 24$ and $a = 17$.

In practice the sequences is precomputed and presorted in time $O(m)$, which also is the memory needed to store sequences so the search runs in time $O(m)$. Therefore the algorithm computes discrete logarithms in cyclic groups of order n in time $O(\sqrt{n})$ using $O(\sqrt{n})$ amount of memory.

Pohlig-Hellman method

This method uses the Chinese remainder theorem to break up the order of the base point in small prime power factors. Let the discrete logarithm, we look at, continue to be

$$a = \log_\alpha \beta, \text{ for } \alpha, \beta \in G \text{ (cyclic of order } n).$$

The base point in the above setting is α. We factor the order n in k small prime power factors $p_i^{c_i}$

$$n = \prod_{i=1}^{k} p_i^{c_i}$$

and solve the discrete logarithm problem for x_i in these smaller instances where

$$x_i \equiv a \pmod{p_i^{c_i}}.$$

In each of the k small logarithms we will look at the p_i radix representation of x_i

$$x_i = \sum_{j=0}^{c_i-1} a_j p_i^j.$$

Then use the relations

$$\begin{aligned} \beta^{n/p_i} &= \alpha^{a_0 n/p_i}, \\ \beta_j^{n/q^{j+1}} &= \alpha^{a_j n/p_i}, \\ \beta_{j+1} &= \beta_j \alpha^{-a_j p_i^j} \end{aligned}$$

to determine the full p_i-radix representation $x_i = (a_0, \dots, a_{c_i-1})$. This has to be performed k times and then use Gauss's algorithm to obtain the discrete logarithm a from the sub-logarithms.

The running time is $O(c_i p_i)$ for each of the k prime factors, but can be improved using Shanks' baby-step giant-step algorithm for $O(c_i\sqrt{p_i})$. This method is therefore only effective when the base point order n contains a lot of small prime factors. The groups used in practice in our signature scheme will be chosen such that this is not the case. Here n will be a single large prime so the Pohlig-Hellman method is not effective against our signature scheme.

Pollard's rho method

Pollard's rho method is named after the way it searches an element collision in G to compute the discrete logarithm. Let the discrete logarithm, we look at, continue to be

$$a = \log_\alpha \beta, \text{ for } \alpha, \beta \in G \text{ (cyclic of order } n).$$

We divide the group G into equal sized sets $G = S_1 \cup S_2 \cup S_3$ such that $1 \notin S_2$. The idea is to look for tuples (x, a, b) where $x = \alpha^a \beta^b$.

Define a looking function $f : \langle\alpha\rangle \times \mathbb{Z}_n \times \mathbb{Z}_n \to \langle\alpha\rangle \times \mathbb{Z}_n \times \mathbb{Z}_n$ by

$$f(x, a, b) = \begin{cases} (\beta x, a, b+1) & \text{for } x \in S_1 \\ (x^2, 2a, 2b) & \text{for } x \in S_2 \\ (\alpha x, a+1, b) & \text{for} x \in S_3 \end{cases}$$

The function f preserves the relation $x = \alpha^a \beta^b$ and in this way traverses tuples where the relation holds. We begin in $(x, a, b) = (1, 0, 0)$ and index the tuples:

$$(x_i, a_i, b_i) = f(x_{i-1}, a_{i-1}, b_{i-1}) \text{ for } i \geq 1.$$

We stop looking when we discover a collision $x_i = x_{2i}$ in the tuples (x_i, a_i, b_i) and (x_{2i}, a_{2i}, b_{2i}). On Figure 5.2 this can be understood graphically as the points where $s = t$.

Then it can be shown that

$$a \equiv (a_i - a_{2i})(b_{2i} - b_i)^{-1} \pmod{n}.$$

This algorithm computes discrete logarithms in cyclic groups of order n in time $O(\sqrt{n})$ using a constant $O(1)$ amount of memory. Pollard's rho method is therefore more effective than Shanks baby-step giant-step method wrt. memory consumption, while time complexity is the same as Shanks' method. In practice we will use Pollard's rho method for large n.

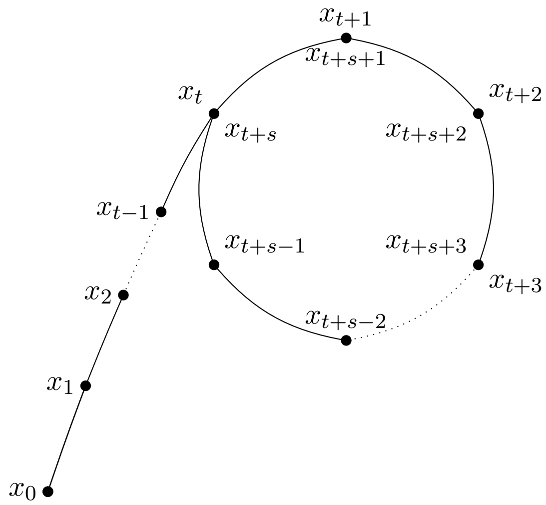


Figure 5.2: Pollard's rho method graphically

5.3.2 The Index Calculus method

Since we have discovered that you can do a MOV reduction with the Weil pairing, we should also note the Index Calculus method on finite fields. This method works for finite fields $\mathbb{F}_q$ by computing the logarithm using a factor base of elements and their logarithms. We look at the discrete logarithm

$$a = \log_\alpha \beta, \text{ for } \alpha, \beta \in \mathbb{F}_q \text{ (cyclic of order } n).$$

A factor base is simply a predetermined set $\mathcal{B}$ of primes we want to factor n over.

$$\mathcal{B} = \{\pi_1, \ldots, \pi_b\}$$

If n can be completely split over a base with a biggest prime b, we say that n is smooth with respect to b. The concept of a factor base generalizes directly to function fields, here the primes are substituted with irreducible polynomials.

In a preprocessing step a sieve method is used to construct the factor base. We then create a number of relations of powers of α factored over the factor base.

$$\begin{aligned}
\alpha &\equiv \pi_{1_1}^{c_{1_1}} \cdots \pi_{s_1}^{c_{s_1}} \pmod{n} \\
\alpha^2 &\equiv \pi_{1_2}^{c_{1_2}} \cdots \pi_{s_2}^{c_{s_2}} \pmod{n} \\
&\vdots \\
\alpha^t &\equiv \pi_{1_t}^{c_{1_t}} \cdots \pi_{s_t}^{c_{s_t}} \pmod{n}
\end{aligned}$$

with $\pi_{i_j} \in \mathcal{B}$, $i = 1, \ldots, s$, $j = 1 \ldots t$ for $t \geq |\mathcal{B}|$. Taking logarithms on each side we will get a linear system of logarithms

$$\begin{aligned}
1 &\equiv c_{1_1} \log_\alpha \pi_{1_1} + \ldots + c_{s_1} \log_\alpha \pi_{s_1} \pmod{n} - 1 \\
2 &\equiv c_{1_2} \log_\alpha \pi_{1_2} + \ldots + c_{s_2} \log_\alpha \pi_{s_2} \pmod{n} - 1 \\
&\vdots \\
t &\equiv c_{1_t} \log_\alpha \pi_{1_t} + \ldots + c_{s_t} \log_\alpha \pi_{s_t} \pmod{n} - 1
\end{aligned}$$

which we solve. In this way we obtain the logarithm value of the factors in the factor base.

$$\mathcal{B}_{\log_\alpha} = \{\log_\alpha \pi_1, \ldots, \log_\alpha \pi_b\}$$

In the main computation a random number s is chosen and you try to factor $\beta\alpha^s$ over the generated factor base.

$$\beta\alpha^s = \pi_1^{c_1} \cdots \pi_{k_1}^{c_{k_1}} \pmod{n}.$$

If this can be done, you take the logarithms on both sides otherwise pick another random number s and to factor again.

When $\beta\alpha^s$ is succesfully factored over $\mathcal{B}$ you compute

$$\log_\alpha \beta \equiv c_1 \log_\alpha \pi_1 + \ldots + c_{k_1} \log_\alpha \pi_{k_1} \pmod{n-1}$$

from the logarithms of the factors in $\mathcal{B}_{\log_\alpha}$.

Prime field $\mathbb{F}_p$

The complexity of this method when q is a prime p, has sub-exponential in running time [Sti05] in the size of p.

$$\text{Pre-computation: } O\left(e^{(1+o(1))\sqrt{\ln p \ln \ln p}}\right)$$
$$\text{Main computation: } O\left(e^{(1/2+o(1))\sqrt{\ln p \ln \ln p}}\right)$$

If we use the General Number Field Sieve (GNFS) [Sti05, p.200] for the sieving process then the precomputation time have time complexity $L[1/3, (64/9)^{\frac{1}{3}}]$. For simplicity we will refer to the running time of GNFS for high characteristic fields. Note that the right thing to do, would be to use the function field sieve (which is discussed in next section) when we work in extensions of large prime characteristic. To avoid confusion with the small characteristic case we say we use GNFS.

Let $\mathcal{B}$ be the factor base. In the simple case we only store precisely enough data to solve the system of relations. The amount of memory required is

$$O\left(|\mathcal{B}|^2 \log n\right),$$

Algorithm	Complexity
Brute force	$O(n)$
BSGS	$O(\sqrt{n})$
Pohlig-Hellman	$O(c_{max}\sqrt{p_{max}})$
Pollard-Rho	$O(\sqrt{n})$
IC in $\mathbb{F}_p$	$L[\frac{1}{3}, (64/9)^{1/3}]$
IC in $\mathbb{F}_{p^m}$	$L[\frac{1}{3}, (32/9)^{1/3}]$

Table 5.1: Time complexity for discrete logarithm algorithms measured in group size n or finite field size q

where $|\mathcal{B}| = \frac{2^{b+1}}{b}$ for the factor polynomials π_i degree bound b [Cop84]. Thus the algorithm takes up a lot of memory resources. For simplicity we will only note this, but in practice it also takes noticeable time to handle memory resources of this size. Looking aside from memory costs, choosing a higher bound b makes the pre-computation faster since it is easier to produce the relations required. The larger your factor base is, the easier it is to choose an s such that $\beta\alpha^s$ factors over the base.

Low characteristic function field $\mathbb{F}_{p^m}$

For fields $\mathbb{F}_{2^m}$ Coppersmith [Cop84] has refined the index calculus algorithm. The time complexity when $q = 2^m$ becomes[1]

$$\text{Precomputation: } O\left(e^{(c+o(1))(m^{1/3}\ln^{2/3} m)}\right)$$
$$\text{Computation: } O\left(e^{(\ln 3+o(1))(m^{1/3}\ln^{2/3} m)}\right).$$

Here the constant c depends on the complexity of solving the linear system of relations. If this complexity is assumed quadratic in number of relations, then $c \simeq 1,405$ [Cop84]. For function fields of small characteristic $p \leq m^{o\sqrt{m}}$ with a carefull choice of input more the Function Field Sieve (FFS) will have running time $L[1/3, (32/9)^{1/3}]$ [JL02]. For simplicity we shall just refer to the running time of the FFS for low characteristic fields. The time complexity of all the above described discrete logarithm algorithms is summed up in Table 5.1.

[1] Note that in the main computation stage the term $\ln 3$ arises from the number of trials needed when you set $b = n^{\frac{2}{3}} \ln^{\frac{1}{3}} n$ [Odl85]

5.3.3 A small experiment

To illustrate the effectiveness of the MOV reduction, I have used the Coppersmith Index Calculus implementation in Magma mathematics software package [BCP97]. I have created a Magma script (see Appendix F.7) that

1. Computes the discrete logarithms in supersingular curve groups over fields $\mathbb{F}_{2^m}$ of characteristic 2.
2. Does a MOV reduction.
3. Computes the logarithm in finite field extensions $\mathbb{F}_{2^{4m}}$.

The curves we will look at is $E_{2,1}$ and $E_{2,2}$ given in Example D.3. They both have embedding degree $k = 4$. Magma hash a discrete logarithm function for both elements in curve groups and elements in finite fields.

For elliptic curve groups with high prime factor subgroups, Magma uses Pollard's rho methodand for characteristic 2 fields Magma uses Emmanuel Thomé's implementationof Coppersmith's Index Calculus algorithm [Tho01].

A bug in the Magma implementation, preventing me from setting any parameters in the Index Calculus algorithm was discovered[2], so the following experiments have only been performed with Magmas default Index Calculus parameters. Note that the parameter `RelationsRatio`, which is the number of relations over the number of elements in your factor base, defaults to 1.2. This has the implication in the pre-processing step of making the linear system of relations faster to solve than for smaller values. This also makes the demand for memory higher and reading and writing to memory takes time. This will in fact turn out to be a limiting factor in the experiment.

Setup

The tests made was done on DTU's Sun Fire E6900 server with 4 x 1 GHz processors. Magma does not multi-thread its processes, so the CPU time measurements is based on a single 1 GHz processor.

In the Magma script we vary the base field $\mathbb{F}_{2^m}$ extension $m = 1, \ldots, 67$. For each curve we can use the formula from Example D.3 to compute the curve group order and find the largest prime order subgroup to test on. The test consists of computing different discrete logarithms $n = 10$ times over the curve group, doing the MOV reduction into field $\mathbb{F}_{2^{4m}}$ and then using the index calculus algorithm in this field. I've implemented the Magma script such that it starts by running the index calculus algorithm one time, where

[2] The bug have since been fixed in MAGMA V2.15-2

the pre-computation is preformed together with the main computation. In the n following computations the pre-computation is not performed. In this way the performance is improved, but it also gives us a way to see the main computation separate from the precomputation.

Results

The results produced from the Magma script is found in Table 5.2 and Table 5.3. If we plot the CPU timings, it's easy to see in Figure 5.3 and Figure 5.4 that the time it takes to do logarithms in the curve subgroup $\langle P \rangle$ comes in spikes. The spikes represent the cases where the prime factorisation of $E_{2,i}(\mathbb{F}_{2^m})$ contains a large prime factor, which is the order of the subgroup $\langle P \rangle$.

The result in Table 5.2 and Table 5.2 contain cases $m = 33, 35, 39, 45$ where the time for both main computations and pre-computations vary significantly from the strictly increasing behaviour you would expect. The reason for this could be some undocumented shortcut from Magma, but from the documentation, it is not apparent why these should be faster to do the Index Calculus logarithm on.

Notice in the case of curve $E_{2,1}$, that for $m = 53$ the Index Calculus computation is faster than the generic discrete logarithm computation for the large subgroups. This important observation tells us that, in this case, the Index Calculus method is more effective than the generic algorithm.

Limitations

The reason for not going higher than extension degree $m = 65$ is the issue with Index Calculus implementation in Magma. The default settings make the Index Calculus algorithm too slow for the computer system used in the experiment. What we can do is to use our algorithms theoretical time complexities to plot the development of required number of operations using Pollard's rho method and the Coppersmith Index Calculus method for the curves $E_{2,1}$ and $E_{2,2}$. This will give a more clear picture of what we saw in the experimental results.

Let the subgroup order, which we use for input in the generic algorithms time complexity, be the largest order subgroup calculated over both curves and only store the strictly growing group orders, for details see Appendix F.8. From Table 5.1 we see that Pollard's rho method takes time $O(\sqrt{p})$ in our prime subgroup $\langle P \rangle$. We ignore the constant in the big-O notation and set $t_{rho}(p) = \sqrt{p}$. For the IC algorithm we disregard the little-o weight. We

m	Dlog in $\langle P\rangle$	Reduction	IC precomp.	IC main comp.
3	0.000	0.001	0.000	0.000
5	0.000	0.001	0.000	0.000
7	0.000	0.005	0.000	0.001
9	0.001	0.009	0.009	0.001
11	0.001	0.010	0.000	0.001
13	0.006	0.0130	0.000	0.002
15	0.005	0.015	0.007	0.003
17	0.009	0.023	0.007	0.003
19	0.001	0.016	0.017	0.003
21	0.021	0.025	0.012	0.008
23	0.013	0.024	0.024	0.006
25	0.035	0.047	0.015	0.015
27	0.039	0.048	0.023	0.017
29	0.297	0.059	25.672	2.068
31	0.026	0.032	29.565	4.725
33	0.092	0.054	0.031	0.019
35	0.270	0.077	0.079	0.041
37	0.504	0.089	37.859	3.921
39	0.032	0.045	0.067	0.013
41	0.245	0.094	44.914	7.516
43	44.362	0.194	57.809	17.541
45	0.039	0.056	0.132	0.018
47	3.571	0.128	453.131	61.959
49	6.141	0.142	568.618	69.262
51	0.036	0.066	1460.873	97.877
53	1796.178	0.270	1456.450	128.200
55	79.714	0.248	1756.489	155.571
57	24.216	0.206	2017.111	227.679
59	0.052	0.103	2025.859	234.541
61	27.234	0.274	2896.387	281.013
63	10.452	0.260	3782.372	391.738
65	0.370	0.158	5989.431	450.829

Table 5.2: Magma MOV reduction cpu(s) timings in curve $E_{2,1}(\mathbb{F}_{2^m})$.

m	Dlog in $\langle P \rangle$	Reduction	IC precomp.	IC main comp.
3	0.000	0.001	0.000	0.000
5	0.000	0.002	0.000	0.000
7	0.000	0.007	0.000	0.001
9	0.001	0.007	0.000	0.002
11	0.005	0.011	0.000	0.002
13	0.000	0.011	0.008	0.002
15	0.001	0.009	0.007	0.003
17	0.002	0.016	0.007	0.003
19	0.013	0.030	0.014	0.006
21	0.012	0.022	0.001	0.009
23	0.023	0.031	0.024	0.006
25	0.02	0.032	0.007	0.013
27	0.039	0.046	0.022	0.018
29	0.640	0.048	20.825	5.725
31	0.041	0.052	28.132	4.068
33	0.045	0.054	0.038	0.012
35	0.127	0.063	0.097	0.023
37	0.500	0.080	37.133	3.707
39	0.018	0.035	0.068	0.012
41	0.155	0.082	46.020	8.760
43	0.054	0.072	79.803	16.457
45	0.275	0.098	0.102	0.048
47	449.059	0.224	459.043	60.437
49	97.750	0.242	516.434	74.726
51	1.693	0.133	1551.807	96.833
53	1.014	0.131	1481.625	128.525
55	26.037	0.257	1713.037	154.383
57	0.074	0.084	2082.369	221.991
59	15.083	0.249	2084.376	244.894
61	35.502	0.295	2912.785	286.945
63	16.467	0.289	3491.499	350.731
65	518.930	0.362	5771.977	396.313

Table 5.3: Magma MOV reduction cpu(s) timings in curve $E_{2,2}(\mathbb{F}_{2^m})$.

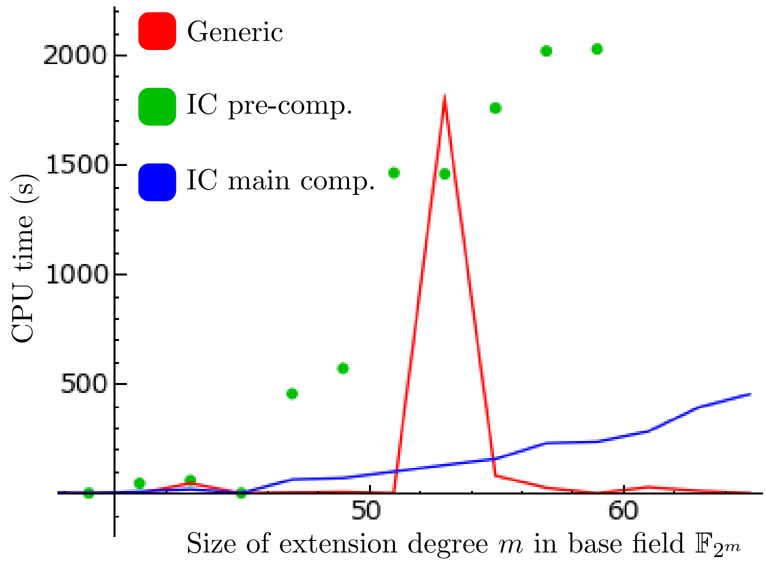


Figure 5.3: Plot of CPU timing results for curve group $E_{2,1}$.

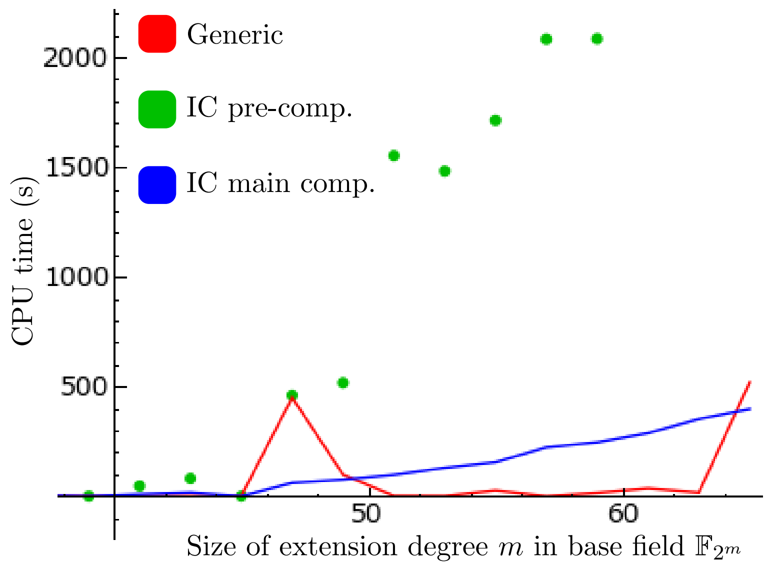


Figure 5.4: Plot of CPU timing results for curve group $E_{2,2}$.

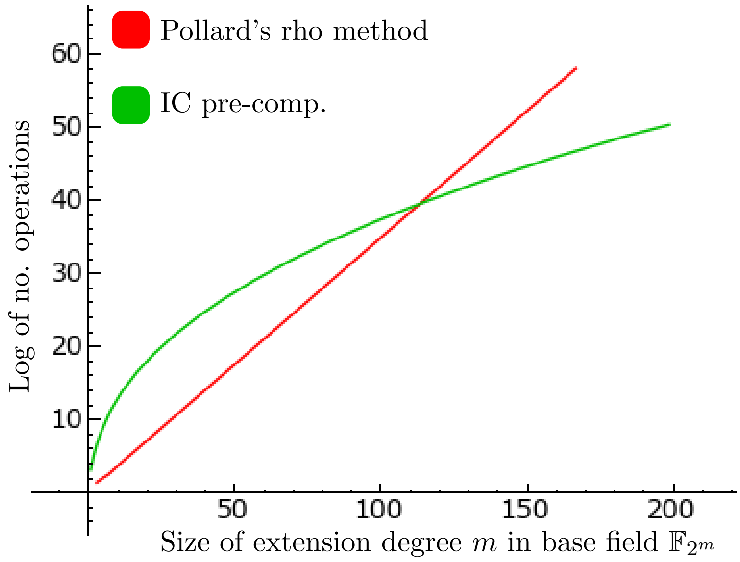


Figure 5.5: Log-plot of t_{rho} and t_{IC} wrt. to the base field extension degree m and elliptic curves $E_{2,1}$ and $E_{2,2}$

then set for field $\mathbb{F}_{2^m}$ and embedding degree k

$$t_{IC}(m) = \exp(c \cdot (mk)^{1/3} \ln(mk)^{2/3}).$$

Choose $c = 1.405$ since we could use Coppersmith in characteristic 2 fields. Then we can do a log-plot of t_{rho} and t_{IC} with respect to the base field extension degree m. This gives us the plot in Figure 5.5.

With the assumptions we have made, the information in the plots should be taken lightly. We see on the figure that the lines cross at a much higher m than was the case in the experiment. An explanation could be that the implementation in Magma maybe does some things faster and we ignore the constant in the time complexity. What we can see with certainty is that the sub-exponential time complexity of the Index Calculus method will make the MOV reduction more efficient to use than a generic algorithm for some large value of m. As our experiment also indicated, this m would for the characteristic 2 case seem to be $m = 53$ in Magma (see table 5.2). So for higher values of m we should base security on the security in the extension field.

5.3.4 Lower bounds on curve parameters

For simplicity we assume that 2^{80} operations is intractable to perform, this is of course relative to the time we are given and the sophistication of the hardware we use, but let us just for now disregard this. In this setting we will try to give a lower bound on the curve parameters for intractability of co-CDH with the Weil pairing when we use (supersingular) elliptic curves over small characteristic fields, i.e. fields of characteristic 2 or 3.

Among known methods for solving discrete logarithms in the generic group case the following non-trivial was described: Shanks baby-step giant-step, Pohlig-Hellman and Pollard's rho method. These generic methods have time complexity in $O(\sqrt{n})$ and the group order n should therefore be chosen large enough to make the methods computational intractable. This means that the group order $n > 2^{160}$, i.e. n must be at least 160 bits long.

In Chapter 4 it was shown how to reduce the problem of finding the discrete logarithm in the curve group $E(\mathbb{F}_q)$ to that of finding the discrete logarithm in the field $\mathbb{F}_{q^k}$, where k is the embedding degree of the group $\langle P \rangle$. You can then solve the discrete logarithm problem in $\mathbb{F}_{q^k}$ with the sub-exponential Index Calculus algorithm.

We saw in the experiment, that for characteristic $p = 2$, this attack would dominate in time complexity for $p \geq 2^{53}$. So the Index Calculus attack is more effective than the generic ones when $n \geq 2^{160}$. In this case it is therefore important to make sure that q^k is sufficiently large. For complexity 2^{80} we take the logarithm of the time complexity of the Index Calculus time complexities and see when it equals 80. For large characteristic p fields:

$$\left(\frac{64}{9}\right)^{\frac{1}{3}} \log(e) \left(\log p \ln(2)\right)^{\frac{1}{3}} \ln^{\frac{2}{3}} \log p \ln(2) > 80$$

for $\log(p) > 850$ and for small characteristic p fields:

$$\left(\frac{32}{9}\right)^{\frac{1}{3}} \log(e) \left(m \ln(2)\right)^{\frac{1}{3}} \ln^{\frac{2}{3}} m \ln(2)$$

for $\log(p^m) > 1448$.

This means, that in the case of small characteristic fields we would need bitsize of the order of the extension field to be greater than 1448 bits, to ensure 80 bits of security. While in the case of a large characteristic field we only need a extension field size greater than 850 bits, to ensure 80 bits of security.

Chapter 6

BLS scheme using the Weil Pairing

In this section we will implement the BLS signature scheme using the Weil pairing together with elliptic curve groups. First the signature scheme will be defined using elliptic curve groups and the Weil pairing without stating anything about the curve. We will then try to select a specific supersingular curve with parameters such that the group pair is a co-GDH pair. When a specific curve is selected, we will discuss how to optimize the Weil pairing implementation for the specific curve.

6.1 BLS with elliptic curve groups

With the elliptic curve co-GDH group pair just defined, the BLS signature scheme described in Section 2.1 can be implemented using elliptic curve groups.

Let $G_1 = \langle P \rangle$ be the prime order n subgroup generated by point $P \in E(\mathbb{F}_q)$ then also $G_1 \in E(\mathbb{F}_{q^k})$ when k is the embedding degree of P and there exists a prime order n subgroup $G_2 \in E(\mathbb{F}_{q^k})$ with linear independent points of the ones in G_1. Let Q generate G_2. The public key will then be a point V in G_2 and the private key is a residue $x \in \mathbb{Z}_n$. We should also ensure that $tr(Q) \neq \mathcal{O}$.

We modify the Algorithms 2.1.1, 2.1.2, 2.1.3 slightly and get Algorithms 6.1.1, 6.1.2. 6.1.3. Key generation in Algorithm 6.1.1 is done by simple point scaling. Signing in Algorithm 6.1.2 uses the MapToGroup algorithm to hash a string into an elliptic curve curve group G_1 and multiplies this with the private key. Verification in Algorithm 6.1.3 uses the Weil pairing to test that (σ, Q, R, V) is a valid co-DDH tuple.

Algorithm 6.1.1: ECKeyGen

Data: point Q generating G_2, prime order p of G_1
Result: private key $x \in \mathbb{Z}_p$, public key $V \in G_2$
Choose random $x \in \mathbb{Z}_p$
$V \leftarrow x \cdot Q$
return (x, V)

Algorithm 6.1.2: ECSign

Data: private key $x \in \mathbb{Z}_p$, message $M \in \{0,1\}^*$
Result: signature $s \in \mathbb{F}_q$
$R \leftarrow MapToGroup'_H(M) \in G_1$
$\sigma \leftarrow x \cdot R$
$s \leftarrow \sigma(x)$
return s

Algorithm 6.1.3: ECVerify

Data: public key $V \in G_2$, message $M \in \{0,1\}^*$, signature $s \in \mathbb{F}_q$
Result: boolean value
if *exists a value* y *such that* $(s, y) \in E(\mathbb{F}_q)$ **then**
    $\sigma \leftarrow (s, y)$
else
    return False
$h \leftarrow H(M) \in G_1$
if $e_n(\sigma, Q) = e_n(h, V)$ *or* $e_n(\sigma, Q)^{-1} = e_n(h, V)$ **then**
    return True
else
    return False

The signature scheme when using elliptic curve groups with the Weil pairing is well defined by Theorem 2.1. The signature scheme is secure by Theorem 2.11 if we choose our elliptic curve groups in respect to the previous section such that they are co-GDH groups. The signature size in the signature scheme is $\log q$, since $s \in \mathbb{F}_q$.

6.1.1 Implementation of the BLS scheme

The described signature scheme is implemented using elliptic curve groups in Sage. The implementation found in Appendix F.9 is implemented as a full `BLSSignatureScheme` class. The `BLSSignatureScheme` object is initialised with parameters :

- g_1: The generator (a point) of curve subgroup G_1.
- g_2: The generator (a point) of curve subgroup G_2.
- m: The base curve order $m = |E(\mathbb{F}_q)|$.
- n: The subgroup prime order $n = |G_1| = |G_2|$.

When the signature object is instanciated the embedding Φ is instanciated and stored on the signature object. The generator g_1 is then mapped into the curve $E(\mathbb{F}_{q^k})$. There is also created prime field object, used to select the private key in. These things should be noted to be possibly significantly time consuming, so saving the scheme object to file and then loading it, is much better in stead of instanciating it over and over again.

The signature scheme can sign large text files in Sage. But you can also use the included Sage script found in Appendix F.11 to start the signature scheme in a simple command line interface outside the sage CLI. The signature could be used in practice with email using a Sage script. See Appendix E for more detail on how to operate the scheme in the text interface and scripting to Sage. A small example of the siganture scheme in Sage follows here.

Example 6.1 (BLS signature)**.** *In this example we again look at the elliptic curve used in Example 3.32. First we need some generators for G_1 and G_2, respectively P and Q. We will just produce these the same way as we did in Example 4.12 (see Appendix F.10) and check that they are both of order 113 and not linearly dependent.*

```
sage: load BLS_example.sage
sage: (113*P1).is_zero()
True
sage: (113*Q).is_zero()
True
sage: P2.weil_pairing(Q,113)!=F2.one_element()
True
```

The independent pair now generates the co-GDH pair (G_1, G_2) as required. We are ready to generate a key pair and ensure it is in $E(\mathbb{F}_{q^4}) \times \mathbb{Z}_{113}$.

```
sage: BLS = BLSSignatureScheme(P1,Q,m,n)
sage: BLS.generate_key_pair()
sage: pub = BLS.public_key()
sage: priv = BLS.private_key()
sage: type(pub)
<class 'sage.schemes.elliptic_curves.ell_point.
EllipticCurvePoint_finite_field'>
sage: type(priv)
<type 'sage.rings.integer_mod.IntegerMod_int'>
```

The produced key pair can now be used to sign the following message.

```
sage: msg="Hello World"
sage: BLS.sign(msg, priv)
sage: BLS.signature in F1
True
```

Now we will verify the signature using the generated public key.

```
sage: BLS.validate(msg, sig, pub)
True
sage: BLS.generate_key_pair()
sage: BLS.validate(msg, sig, BLS.public_key())
False
```

The example is not applicable in practice since the groups are too small for the co-CDH problem to be intractable. In the next section we will try to find a suitable supersingular curve, where this is the case.

Speed

The most expensive feature of the BLS system is the signature verification taking two Weil pairing computations. But signing also takes some time since it's a point scaling in the size of n. The different operations in the scheme is timed (1.2 GHz processor) to see how signing, keygeneration and initialisation of the BLS class performs.

In Table 6.1 I have collected the time it takes to do the BLS operations keygeneration, signing and verification using some different supersingular elliptic curves and an MNT[1] curve with a subgroup size of 158 bits. We verify from the table that signing, which is a point scaling, is very fast in

[1] Curves named after researchers Miyaji, Nakabayashi and Takano where the embedding degree can be controlled.

Curve	subgroup G_2 order n (bits)	Initialise class (s)	Keygen (s)	Signing (s)	Verify (s)
$E_{3,2}(\mathbb{F}_{3^{17}})$	27	0.7	0.37	0.09	7.79
$E_{3,1}(\mathbb{F}_{3^{53}})$	85	13.91	7.46	0.6	168.94
$E_{3,2}(\mathbb{F}_{3^{79}})$	126	48.86	25.64	1.25	561.21
$E_{3,1}(\mathbb{F}_{3^{97}})$	154	64.20	45.49	2.21	1076.65
$E_{MNT}(\mathbb{F}_p)$	158	0.03	0.33	0.02	4.46

Table 6.1: Timing (s) of BLS implementation in Sage for different curves

comparison with the verification. This is due to the implementation. Point scaling is all done within the compiled PARI C code in Sage while verification rely on the efficiency of my pairing implementation in Python, which is not as fast as C. We saw in the Weil pairing performance table, that there is a significant difference in the time Sage uses for finite field computations in low characteristic fields and high characteristic fields. If we assume that Sage is flawed and that it should be faster to work in small characteristic fields than large prime characteristic fields, then from the MNT curve case we have a verification in 4.5 seconds, which is acceptable in a general implementation.

6.2 Selecting an appropriate curve

In this section we will select a supersingular curve and try to see if we can get a real scale system from it.

First some general observations on the parameters of the signature scheme.

1. Signature length $\log q$ depends on the size of the base field $\mathbb{F}_q$.
2. If we want at least 80 bits of security wrt. generic Dlog attacks, then we need $q > n > 2^{160}$.
3. We also need to prevent that the MOV reduction is effective, so we need to have the size of the extension field $\mathbb{F}_{q^k}$ to be large enough to handle the Index Calculus attack. This means that

   $$\log q > \frac{|\mathbb{F}_{q^k}|}{k}.$$

   So to have an effective small signature, a large embedding degree k is good.
4. It should be noted that the arithmetic performed when computing the pairing values for signature validation, is performed in the extended

m	$\log\lvert E_{3,i}\rvert$	$\max_{3,1}\log\lvert G_1\rvert$	$\max_{3,2}\log\lvert G_1\rvert$	$\lceil\log 3^m\rceil$	$\lceil 6\log 3^m\rceil$
$*$ 149	237	220	151	237	1422
$*$ 151	240	104	105	240	1440
155	246	77	116	246	1476
$*$ 157	249	128	180	249	1494
161	256	124	138	256	1536
$*$ 163	259	256	259	259	1554
$*$ 167	265	262	237	265	1590
169	268	107	218	268	1608
$*$ 173	275	145	241	275	1650
175	278	70	191	278	1668
$*$ 179	284	139	193	284	1704
$*$ 181	287	122	198	287	1722
185	294	127	100	294	1764
187	297	245	153	297	1782

Table 6.2: Bitsizes of supersingular curve groups $E_{3,2}(\mathbb{F}_{3^m})$ and $E_{3,2}(\mathbb{F}_{3^m})$.

field. Thus, it is dependent on the extension degree k in terms of speed and memory consumption.

The Weil pairing performance depends on the subgroup order n since the algorithm used was based on double and add of a point up to n times that point. We will in the next section discuss how we can optimize the Weil pairing with respect to the bit representation of n.

What criteria should you look for when selecting a curve to use in the BLS signature scheme?

We need an elliptic curve that induces subgroups large enough for the co-CDH problem to be intractable. We saw in the previous section that this meant for small characteristic supersingular elliptic curves that $\log q^k > 1448$. This makes a good argument for choosing the supersingular elliptic curves over characteristic 3, since they have embedding degree $k = 6$, while in characteristic 2 the embedding degree is $k = 4$.

For supersingular elliptic curves we have explicit formulas for the curve group order with respect to the base field degree e. The security against generic discrete logarithm attacks is based on the size of the prime order subgroup, which we use in the co-GDH group pair in the signature scheme. As we saw in the previous sections experiment, we got peaks in computation time whenever the curve order factorization contained large primes i.e. large prime order subgroups. Let us therefore look at the bitsize of the largest prime subgroups for two supersingular curves over fields of characteristic 3.

Curve	Sig. bitsize $\log q$	gen. DLog. $\log n$	MOV security $6 \log q$
$E_{3,1}(\mathbb{F}_{3^{149}})$	237	220 (110)	1422 (79)
$E_{3,2}(\mathbb{F}_{3^{163}})$	259	259 (129)	1554 (82)
$E_{3,1}(\mathbb{F}_{3^{167}})$	265	262 (131)	1590 (83)

Table 6.3: Security properties of candidate curves.

In Table 6.2 we see the bitsize of the largest prime order subgroups of the two curves.

Besides having

$$|\mathbb{F}_{q^6}| = 6 \log 3^e > 1448$$

we would also like to utilize as much of the curve as possible, by that meaning getting the subgroups (relative to the curves size) biggest possible.

Possible candidates could be $m = 149, 163, 167$.

Table 6.3 shows our candidates' security properties with equivalent [Len01] bit security in trailing parentheses. To translate the generic security to bit security you just multiply by $1/2$, since the generic attacks work in time complexity square root of the group order, so in the first group we have 110 bits security. Notice we are just below the limit of 1448 bit MOV security. This means that our signature have to be approximatly minimum 237 bits long. This is still better than the equivalent ECDSA length of 320 bits. But it seems that we have some overhead in the extra 30 bits security against generic attacks. Since the Index Calculus attack is subexponential and the group order n is bounded by the curve group order, which is approximatly the same bitsize as q, then this overhead in bits can only grow. So even if we keep the ratio between the curve group order m and the subgroup order low, i.e.

$$\frac{m}{n} \sim 1$$

like in the curve $E_{3,2}(\mathbb{F}_{3^{163}})$ we will still have a gap in the MOV security and the curve generic attack security.

If we want to compare our scheme with the low characteristic curves to the current standard ECDSA, we should compare the bit security with respect to the MOV reduction. Because the MOV reduction turned out to be the most effective method of solving the DLog problem in the case of low characteristic supersingular elliptic curves. I've done this in Table 6.4 using the the results from 2001 in a security article by Lenstra [Len01].

The case where the elliptic curve is an MNT curve over a prime field is included (the elliptic curve is found in the BLS article [BLS04, Table 1]) to

Signature scheme	Sig. size	Pub. key size	Priv. key size
$\mathbb{F}_{2^{164}}$ ECDSA	328	164	164
$\mathbb{F}_{3^{163}}$ $\text{BLS}_{\text{supersingular}}$	259	1554	259
$\mathbb{F}_p$ BLS_{MNT} $\lvert p\rvert = 163$ bits	163	978	163

Table 6.4: 82 bit security comparisson of BLS and ECDSA

illustrate that you can get smaller signature sizes. This happens because the Index Calculus attack in large prime characteristic fields is not as effective and therefore it is the discrete logarithm attack in the curve group that is dominant. This is a much better situation and essentially what we want and what is referred to in the introduction as the *wise* choice.

6.2.1 Scalability in general

The MOV reduction takes us into a field where we can use sub-exponential algorithms for solving the DLog problem. So for a fixed embedding degree we will have scalability issues on any elliptic curve. If we want a higher bit security, then at some point the bit security will be dictated by the MOV security (ext. field size) and not the elliptic curve size, just as the case is for supersingular curves.

The only way to increase the embedding degree is to find new curves and use these. This is a considerable problem with the scheme. It does not scale for fixed curves since you have to select new curves to get higher embedding degrees along scaling.

6.2.2 Performance

Besides security we need to have good performance. We saw the performance relied on the Weil pairing performance. Miller's algorithm for computing the Weil pairing uses double and add, which is very dependent on the Hamming weight of the bit representation of the subgroup order n.

We can use this to tailor our Weil pairing implementation to the specific bit representation of the order n. An article by Blake et al. [BMX06] gives some refinements of Miller's algorithm. The refinements is a general improvement to all cases of n and an improvement in cases of high hamming weight. Thus if we can use a subgroup of high Hamming weighted order, this would increase performance of the pairing computation in that special case.

In the article the author also propose tripple and add algorithms for characteristic 3 fields. By doing this computations in a normal basis of the field

algorithm	signing	verification
RSA, $\lvert n\rvert = 1024$ bits, $\lvert d\rvert = 1007$ bits	7.90	0.40
DSA, $\lvert p\rvert = 1024$ bits, $\lvert q\rvert = 160$ bits	4.09	4.87
$\mathbb{F}_p$ ECDSA, $\lvert p\rvert = 160$ bits	4.00	5.17
$\mathbb{F}_{2^{160}}$ ECDSA	5.77	7.15
$\mathbb{F}_{3^{97}}$ BLS (supersingular)	3.57	53
$\mathbb{F}_p$ BLS (MNT), $\lvert p\rvert = 157$ bits	2.75	81.0

Table 6.5: Comparison of signing and verification times (in ms) on a PIII 1 GHz. [BKLS02, Table 4]

would make the tripling or doubling in (characteristic 2) into a simple shift operation in the computer memory. Since it's to expensive to switch between bases of a field along the way in the computation, you would have to do the whole system in the normal basis of the field. This is beyond the scope of this thesis.

The Sage Interact in Appendix F.12 illustrates the optimizations mentioned by printing the calculated expression for a single call to the Miller's algorithm in the different versions the authors give.

An obvious problem with these optimizations is that you need to take into account the Hamming weight of subgroup order n when searching for elliptic curves to use. Even with the mentioned optimizations we would still have the same time complexity.

The most important part from a performance perspective is that the time complexity is linear in the bitsize of the subgroup order by Theorem 3.33. In the article "Efficient Algorithms for Pairing-Based Cryptosystems" [BKLS02] the authors state some impressive timing results for the pairing-based BLS signature scheme together with timing results for other standard signature schemes with 80 bits of security. The results are shown in Table 6.5. Notice that the supersingular BLS they've timed do actually not provide 80 bits of security due to the Index Calculus attack.

It is be possible, even with tailored pairings, to come much closer than a factor 2 to the performance of ECDSA. Since the verfication in ECDSA is much simpler and the equivalent of having to do two of pairing computations, is here to do two point scalings. Remember a single Weil pairing operation consists of two Miller algorithm calls, which in themself have time complexity at least equal to a point scaling. So in the optimal case of having half the ECDSA signature length using the pairing-based scheme, we will have at least the double verification time.

Chapter 7

Conclusion

In this thesis the BLS scheme has been proved secure for co-GDH groups. I have implemented the MapToGroup function in Sage and shown that, given a random oracle, the MapToGroup function does not compromise the signature schemes security.

The Weil pairing has been constructed and implemented in Sage using Miller's algorithm for efficient computation. As a consequence, we got the MOV reduction of the DLog problem on a supersingular curve to the DLog problem in the field extension. It was showed how to obtain co-GDH groups from elliptic curve groups using the Weil pairing. A small experiment in Magma, underlining the problem of the MOV attack when using elliptic curve groups for co-GDH groups, was discussed.

In the last section the BLS short signature scheme was defined with elliptic curve groups and implemented in Sage. Selecting an appropriate elliptic curve has been discussed. It was argued, that supersingular elliptic curves over small characteristic fields is a bad choice. Because the MOV attack makes the security of the scheme rely on the finite extension field and not the elliptic curve group. Furthermore it was argued that finding good elliptic curves for our purpose is hard. Finally it was discussed how to tailor the Weil pairing to a single curve selection.

So in short, the conclusion of this report is that Boneh et al. is right when mentioning that supersingular elliptic curves over small characteristic fields is a bad idea. We saw that the Index Calculus attack became more effective for these curves than the generic attack on the curve group, forcing us to use longer signatures than the optimal length. We after all still got a shorter signature than the ECDSA with 82 bits of security, but we saw that it didn't scale when the embedding degree was fixed.

A sub consequence of the conclusion is that finding curves, which meet these demands, is not straight forward. It could be a limiting factor in making the short signature scheme popular, since we need curves with controllable embedding degrees in order to scale in bit security.

This naturally leads to the idea of using high prime characteristic fields as base fields for our elliptic curves. This prevents the use of the more efficient Function Field Sieve in the Index Calculus attack. The problem with supersingular curves is that only curves of characteristic 2 and 3 have embedding degree 4 and 6, while in other cases we get embedding degree 1,2,3 as we see from Appendix D. Even with embedding degree 3 you would get a situation where the security would rely on the MOV security instead of the generic attack security, as we want it to.

The search for elliptic curves to use in the BLS scheme should continue in the field of non-supersingular elliptic curves over fields of high prime characteristic, as the case with the non-supersingular MNT curves.

An observation, which is worth concluding on, was the trade off between computational load and signature length of ECDSA and BLS. This should clearly be considered when making the decision of which scheme to use. But it seems that the current development in mobile processors contra the development in bandwidth makes shorter signatures more and more attractive.

Bibliography

[BCP97] Wieb Bosma, John Cannon, and Catherine Playoust. The magma algebra system i: the user language. *J. Symb. Comput.*, 24(3-4):235–265, 1997.

[BK98] R. Balasubramanian and Neil Koblitz. The impropability that an elliptic curve has subexponential discrete log problem under the menezes.okamoto-vanstone algorithm. *J. Cryptol.*, 11(2):141–145, 1998.

[BKLS02] Paulo S. L. M. Barreto, Hae Yong Kim, Ben Lynn, and Michael Scott. Efficient algorithms for pairing-based cryptosystems. *CRYPTO '02: Proceedings of the 22nd Annual International Cryptology Conference on Advances in Cryptology*, pages 354–368, 2002.

[BLS04] Dan Boneh, Ben Lynn, and Hovav Shacham. Short signatures from the weil pairing. *J. Cryptol.*, 17(4):297–319, 2004.

[BMX06] Ian F. Blake, V. Kumar Murty, and Guangwu Xu. Refinements of miller's algorithm for computing the weil/tate pairing. *J. Algorithms*, 58(2):134–149, 2006.

[Cop84] D. Coppersmith. Fast evaluation of logarithms in fields of characteristic two. *IEEE Transactions on Information Theory*, 30(4):587–594, 1984.

[DH76] Whitfield Diffie and Martin E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, IT-22(6):644–654, 1976.

[Fre99] Gerhard Frey. Applications of arithmetical geometry to cryptographic constructions. In *Proceedings of the Fifth International*

Conference on Finite Fields and Applications, pages 128–161. Springer-Verlag, 1999.

[Ful89] William Fulton. *Algebraic Curves: An Introduction to Algebraic Geometry.* Addison Wesley Publishing Company, 1989.

[JL02] A. Joux and R. Lercier. The function field sieve is quite special. *Algorithmic Number Theory. 5th International Symposium, ANTS-V. Proceedings (Lecture Notes in Computer Science Vol.2369)*, pages 431–445, 2002.

[JN03] Antoine Joux and Kim Nguyen. Seperating decision diffie-hellman from diffie-hellman in cryptographic groups. *J. Cryptol.*, 16(4):239–247, 2003.

[Kim08] Ian Kiming. *Elliptic Curves: Various supplements.* Lecture supplement notes, 2008. unpublished.

[Lan93] Serge Lang. *Algebra.* Addison-Wesley, third edt. edition, 1993.

[Len01] Arjen K. Lenstra. Unbelievable security. matching aes security using public key systems. *ASIACRYPT '01: Proceedings of the 7th International Conference on the Theory and Application of Cryptology and Information Security*, pages 67–86, 2001.

[Mil04] Victor S. Miller. The weil pairing, and its efficient calculation. *J. Cryptol.*, 17(4):235–261, 2004.

[MOV91] Alfred Menezes, Tatsuaki Okamoto, and Scott Vanstone. Reducing elliptic curve logarithms to logarithms in a finite field. In *STOC '91: Proceedings of the twenty-third annual ACM symposium on Theory of computing*, pages 80–89, New York, NY, USA, 1991. ACM.

[Odl85] A M Odlyzko. Discrete logarithms in finite fields and their cryptographic significance. In *Proc. of the EUROCRYPT 84 workshop on Advances in cryptology: theory and application of cryptographic techniques*, pages 224–314, New York, NY, USA, 1985. Springer-Verlag New York, Inc.

[OP01] Tatsuaki Okamoto and David Pointcheval. The gap-problems: A new class of problems for the security of cryptographic schemes. In *PKC '01: Proceedings of the 4th International Workshop on Practice and Theory in Public Key Cryptography*, pages 104–118, London, UK, 2001. Springer-Verlag.

[Sil86] Joseph H. Silverman. *The arithmetic of elliptic curves.* Graduate Texts in Mathematics. Springer, 1986.

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

[Ste09] William Stein. *Sage: Open Source Mathematical Software (Version 3.2.3)*. The Sage Group, 2009. `http://www.sagemath.org`.

[Sti05] Douglas R. Stinson. *Cryptography: Theory and Practice, Third Edition (Discrete Mathematics and Its Applications)*. Chapman & Hall/CRC, November 2005.

[Tho01] Emmanuel Thomé. Computation of discrete logarithms in gf(207). In *ASIACRYPT '01: Proceedings of the 7th International Conference on the Theory and Application of Cryptology and Information Security*, pages 107–124. Springer-Verlag, 2001.

[Was08] Lawrence C. Washington. *Elliptic Curves: Number Theory and Cryptography, Second Edition*. Chapman & Hall/CRC, 2008.

Appendix A

Sage

In this thesis Sage was used to develop the BLS short signature scheme. In this appendix a quick introduction to the Sage mathematics software package [Ste09] is given.

The Sage open source project consists a collection of open source licencesed mathematics packages like PARI, NTL, etc... This makes up a toolbox with a common syntax for doing advanced mathmatics proof of concept implementations like the one handled in thi thesis.

Sage is Python based and therefore the syntax in sage is almost the same and Python scripts can be run in the Sage interpreter. In Appendix E I show how to install the Sage patches containing the BLS implementation.

The following is a short list of relevant sage commands.

TAB-complete support

Sage supports TAB-complete so at any time you can postfix a Sage object with a punctuation and followed by TAB it will give a complete list of avilable functions for that Sage object.

Finite fields

You should note tha `FiniteField` is just a synonym for `GF`. Generate a prime field object F1:

```
sage: F = GF(101)
sage: type(F1)
<class 'sage.rings.finite_field_prime_modn.
FiniteField_prime_modn'>
```

Generate a galois field object F2:

```
sage: F.<a> = GF(27)
sage: type(F)
<type 'sage.rings.finite_field_givaro.FiniteField_givaro'>
sage: type(a)
<type 'sage.rings.finite_field_givaro.
FiniteField_givaroElement'>
```

Note that Sage has build in dynamic choice of arithematic packages i.e. it will switch to PARI when operating in large finite fields like we will in this thesis.

Elliptic curves

Defining an elliptic curve object in Sage is done in the following way.

```
sage: E=EllipticCurve(F,[0,0,1,1,1])
sage: E
Elliptic Curve defined by y2 + y = x3 + x +1
over Finite Field in a of size 27
sage: sage: E.a_invariants()
[0, 0, 1, 1, 1]
sage: P = E.random_point()
sage: P
(a5 + a4 + a2 + a + 1 : a6 + a5 + a4 + a3 + a2 : 1)
sage: type(P)
<class 'sage.schemes.elliptic_curves.ell_point.
EllipticCurvePoint_finite_field'>
```

Defining a function, statements, loops, etc..

In Sage and Python the syntax is indent sensitive, you indent with 4 spaces.

```
sage: def hello_world(x):
....:          if x < 3:
....:                  print "Hello world!"
```

```
....:   else:
....:                   print "Oh stop it!"
....:
sage: hello_world(1)
Hello world!
sage: hello_world(2)
Hello world!
sage: hello_world(3)
Oh stop it!
```

Loading .sage and .sobj files

Instead of writing everything in the sage commandline you can save Sage scripts, programs to .sage files and load the using the `load` command.

```
sage: load test.sage
if test.sage contained print and then this string,
sage would print it, like this!
```

If it is a .sobj file you have to load it and assign it to a variable.

```
sage: test = load('test.sobj')
```

Sage also contains a `notebook()` mode, this will launch a web-server based browser interface with possibilty of plotting and doing sage interacts, see Figure A.1. The interacts found in the code appendix can be copy pasted into the notebook environment and run.

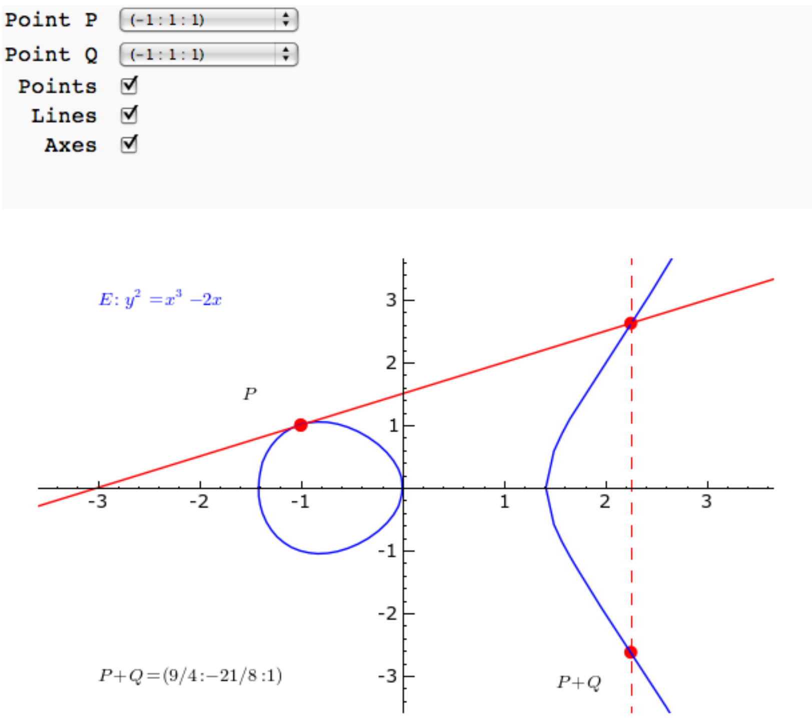


Figure A.1: Sage interact: adding points on an elliptic curve graphically.

Appendix B

Projective geometry

This appendix is a small note on elliptic curves viewed in projective geometry [ST92, p.229] with purpose of explaining the point at infinity $\mathcal{O}$ and that straight lines are well defined with respect to $\mathcal{O}$.

The intuitive idea of projective geometry: if you like you can think of projective spaces as going a dimension up by giving all points an extra coordinate. Let us call this coordinate the direction, if two projective lines are parallel, they may have the same direction z_0 and if you think of the coordinates as of those of planes in $\mathbb{R}^3$ then they would intersect each other in some line $[x, y, z_0]$. Let us try to look at this translation more specific.

Translating from a Euclidian plane into the projective plane you add an extra coordinate and get a set of homogenous coordinates in the following way. A point (x, y) in the Euclidian plane is mapped to the projective point $[x, y, 1]$. Vice versa the projective point $[x, y, z]$ is mapped to the Euclidian point $(x/z, y/z)$ for $z \neq 0$ and $[x, y] \in \mathbb{P}^1$ for $z = 0$. These latter points in $\mathbb{P}^1$ are called the points at infinity, the name arives from the fact of $x/z \to \infty$ and $y/z \to \infty$ for $z \to 0$.

Let us look at the curve $C : f(x, y) = 0$ over a field K, from this you construct a homogenous polynomium $F[x, y, z]$. The points on the curve $\tilde{C} : F[x, y, z] = 0$ can be split into equivalens classes $[a, b, c]$, where a, b, c are not all zero. These will usually be represented by a single point from each class with the equvalens relation $\sim$ defined as

$[a, b, c] \sim [a', b', c']$ if there is a non zero t such that $a = ta',\ b = tb',\ c = tc'$.

Let $\mathbb{P}^2(K)$ be the set of these equivalens classes, then a point $p(x, y) \in \mathbb{P}(K)$

if is can be represented by coordinates $[u, v, w]$ such that $F[u, v, w] = 0$. There is to types of K-rationale points on the curve $\tilde{C}$:

- $Z \neq 0 : [u, v, w] \sim [x/z, y/z, 1]$ is on the curve if $f(x/z, y/z) = 0$.
- $Z = 0$: the points at infinity.

So the K-rationale points on $\tilde{C}$ will be

$$\tilde{C} = \{ \text{ affine points } \} \cup \{ \text{ points at infinty } \}.$$

The general Weierstrass equation in homogenous form:

$$\tilde{E} : \ y^2 z + a_1 xyz + a_3 yz^2 = x^3 + a_2 x^2 z + a_4 xz^2 + a_6 z^3.$$

The affine points is now exactly those $[x/z, y/z, 1]$ where $f(x/z, y/z) = 0$. The points at infinity are those where $F[x, y, 0] = 0$ this yelds in the above equation $-x^3 = 0$ i.e. $x = 0$ so we get the equivalens class $[0, y, 0]$ which we choose to represent by $\mathcal{O} = [0, 1, 0]$.

Since E is an elliptic curve i.e. it is non-singular then it can be shown [Kim08] that E do not contain a whole line $\ell := \alpha x + \beta y + \gamma z = 0$ in the projective plane $\mathbb{P}^2(\bar{K})$.

If we in the natural way define multiplicity of intersections with the line then with respect to multiplicity the line will intersect $\tilde{E}$ exactly three times.

Example B.1. *Let us look at the line $\ell : \ z = 0$ through $\mathcal{O}$. $z = 0$ so intersection points between $\tilde{E}$ and ℓ is $[x, y, 0]$ where $x^3 = 0$ so $x = 0$, so all intersection points are $\mathcal{O}$ with multiplicity 3.*

It can also be shown [Kim08] that if two intersection points are K-rationale then so will the third point be.

Appendix C

Another example

An example based on the elliptic curve E_c on figure 1.1.

Example C.1. *Let us look at the elliptic curve* E_c *on figure 1.1. If we consider the three integral points*

$$(-1, 0), (0, 0), (1, 0).$$

It is clear that they all are of order 2, since doubling them would amount to adding them to them selves by drawing the vertical line as their tangent and getting the point at infinity. Let us show that adding any two of the integral points will produce the third point, this is clear from figure 1.3. So let us try to show this using the above formula. The curve's coefficients in the general weierstrass form are

$$[a_1, a_2, a_3, a_4, a_6] = [0, 0, 0, -1, 0].$$

Let $i, j, k = 1, 2, 3$ *and not pairwise equal, s.t. we may index the point this way. Since* $P_i \neq P_j$ *then since all points had order two,* $P_i \neq -P_j$ *and therefore we have for all integral points in case IIa:*

$$\alpha = \frac{y_j - y_i}{x_j - x_i} = 0 \textit{ and } \beta = \frac{y_i x_j - y_j x_i}{x_j - x_i} \textit{ for } i \neq j, i, j = 1, 2, 3$$

We can then compute the third point R:

$$\begin{aligned} y_k &= -(\alpha + a_1)x_3 - \beta - a_3 = -a_3 = 0 \textit{ and} \\ x_k &= \alpha^2 + a_1\alpha - a_2 - x_i - x_j = -x_i - x_j = -(x_i + x_j). \end{aligned}$$

This clearly shows that adding any two integral points P_i, P_j will produce the third point P_k. We have now shown that

$$\{\mathcal{O}, (-1,0), (0,0), (1,0)\} \simeq \mathbb{Z}_2 \times \mathbb{Z}_2,$$

which is also know as Klein's four group.

Appendix D

Supersingular curves

This appendix section is a sum up of the structure information on supersingular elliptic curves. The following theorem classifies supersingular curves. The proof can be found in the article by Menezes, Okamoto and Vanstone [MOV91].

Theorem D.1. *Let $E(\mathbb{F}_q)$ be a supersingular curve of order $q+1-t$ over $\mathbb{F}_q$ where $q = p^m$ for a prime p. Then E will lie in one of the following six classes*

(I) $t = 0$ *and* $E(\mathbb{F}_q) \simeq \mathbb{Z}_{q+1}$.

(II) $t = 0$, $q \equiv 3 \pmod 4$ *and* $E(\mathbb{F}_q) \simeq \mathbb{Z}_{\frac{q+1}{2}} \times \mathbb{Z}_2$.

(III) $t^2 = q$ *and* m *is even.*

(IV) $t^2 = 2q$, $p = 2$ *and* m *is odd.*

(V) $t^2 = 3q$, $p = 3$ *and* m *is odd.*

(VI) $t^2 = 4q$ *and* m *is even.*

Theorem D.2. *The structure of the curves $E(\mathbb{F}_q) \simeq \mathbb{Z}_{n_1} \times \mathbb{Z}_{n2}$ in each of the described classes can be summarized in table D.1. Here $n_2 = 1$ if $E(\mathbb{F}_q)$ is cyclic and k is the extension degree such that $E[n_1] \subseteq E(\mathbb{F}_{q^k})$ then $E(\mathbb{F}_{q^k}) \simeq \mathbb{Z}_{cn_1} \times \mathbb{Z}_{cn_1}$ for some appropriate c.*

Class	t	$E(\mathbb{F}_q)$	n_1	k	$E(\mathbb{F}_{q^k})$
I	0	cyclic	$q+1$	2	$\mathbb{Z}_{q+1}\times\mathbb{Z}_{q+1}$
II	0	$\mathbb{Z}_{\frac{q+1}{2}}\times\mathbb{Z}_2$	$\frac{q+1}{2}$	2	$\mathbb{Z}_{q+1}\times\mathbb{Z}_{q+1}$
III	$\pm\sqrt{q}$	cyclic	$q+1\mp\sqrt{q}$	3	$\mathbb{Z}_{\sqrt{q^3}\pm1}\times\mathbb{Z}_{\sqrt{q^3}\pm1}$
IV	$\pm\sqrt{2q}$	cyclic	$q+1\mp\sqrt{2q}$	4	$\mathbb{Z}_{q^2+1}\times\mathbb{Z}_{q^2+1}$
V	$\pm\sqrt{3q}$	cyclic	$q+1\mp\sqrt{3q}$	6	$\mathbb{Z}_{q^3+1}\times\mathbb{Z}_{q^3+1}$
VI	$\pm2\sqrt{q}$	$\mathbb{Z}_{\sqrt{q}\mp1}\times\mathbb{Z}_{\sqrt{q}\mp1}$	$\sqrt{q}\mp1$	1	$E(\mathbb{F}_q)$

Table D.1: Structure in supersingular curves

Example D.3. *We look at the curves:*

$$\begin{aligned}
E_{2,1}/\mathbb{F}_2 &: y^2+y=x^3+x+1\\
E_{2,2}/\mathbb{F}_2 &: y^2+y=x^3+x\\
E_{3,1}/\mathbb{F}_3 &: y^2=x^3+2x+1\\
E_{3,2}/\mathbb{F}_3 &: y^2=x^3+2x+2
\end{aligned}$$

Then the curve group orders over finite fields $\mathbb{F}_{2^m}$ and $\mathbb{F}_{3^m}$ satisfies

$$|E_{2,1}(\mathbb{F}_{2^m})| = \begin{cases} 2^m+1-\sqrt{2^{m+1}} & \textit{for } m\equiv\pm1 \pmod 8\\ 2^m+1+\sqrt{2^{m+1}} & \textit{for } m\equiv\pm3 \pmod 8\end{cases}$$

$$|E_{2,2}(\mathbb{F}_{2^m})| = \begin{cases} 2^m+1+\sqrt{2^{m+1}} & \textit{for } m\equiv\pm1 \pmod 8\\ 2^m+1-\sqrt{2^{m+1}} & \textit{for } m\equiv\pm3 \pmod 8\end{cases}$$

$$|E_{3,1}(\mathbb{F}_{3^m})| = \begin{cases} 3^m+1+\sqrt{3^{m+1}} & \textit{for } m\equiv\pm1 \pmod{12}\\ 3^m+1-\sqrt{3^{m+1}} & \textit{for } m\equiv\pm5 \pmod{12}\end{cases}$$

$$|E_{3,2}(\mathbb{F}_{3^m})| = \begin{cases} 3^m+1-\sqrt{3^{m+1}} & \textit{for } m\equiv\pm1 \pmod{12}\\ 3^m+1+\sqrt{3^{m+1}} & \textit{for } m\equiv\pm5 \pmod{12}\end{cases}$$

Note that by theorem 4.6

$$\begin{aligned}
&E_{2,1}(\mathbb{F}_{2^m}) \textit{ and } E_{2,2}(\mathbb{F}_{2^m}) \textit{ have embedding degree } k=4,\\
&E_{3,1}(\mathbb{F}_{3^m}) \textit{ and } E_{3,2}(\mathbb{F}_{3^m}) \textit{ have embedding degree } k=6.
\end{aligned}$$

Appendix E

BLS Signature System Guide

This is a guide on installing and using the BLS signature scheme with Sage.

Sage is available for download at http://sagemath.org. For more information on how to use Sage look in Appendix A or visit the webpage.

E.1 Installation

To install the BLS signature scheme you will need to either apply the sage patch attached on the cd or copy the code from Appendix F into the respective sage source files and recompile Sage.

To install the BLS signature scheme make sure you have Sage version 3.3 or above installed, since then the Weil pairing implementation is already included with your installation.

To apply the patch `bls_scheme.patch` we first create a clone of the main branch, you do not have to do this, it's just to keep your clean installation of sage seperated from a patched one, such that when you later wish to delete the patch you can do it without deleting all of Sage and reinstalling. You can switch between branches using the `hg_sage.swith('branchname')` command.

Start Sage and type in:

```
sage: hg_sage.clone('thesis_branch')
sage: hg_sage.apply('.../.../bls_scheme.patch')
```

What you just installed was actually both the signature system and the MapToGroup hash function so you have access to both functionalities seperatly. The Weil pairing was as mentioned included in the installation og Sage.

E.2 Weil pairing function

The weil pairing is a function defined on elliptic curve point class in Sage, so to access this you need to create an elliptic curve point onject and call it from this.

```
sage: F2=GF(228,'b')
sage: b=F2.gen()
sage: E2=EllipticCurve(F2,[0,0,1,1,1])
sage: m=E2.order()
sage: n = 113
sage: P=int(m/n**2)*E2.random_point()
sage: Q=int(m/n**2)*E2.random_point()
sage: P.order(), Q.order()
(113, 113)
sage: x=P.weil_pairing(Q,n)
sage: x.multiplicative_order()
113
```

E.3 MapToGroup function

MapToGroup is a function defined on the finite field elliptic curve class in Sage, so to access this you just need to create the respective curve object and call it from this. Let us just continue the above Sage session.

```
...
sage: E2=EllipticCurve(F2,[0,0,1,1,1])
sage: type(E2)
<class 'sage.schemes.elliptic_curves.ell_finite_field.
EllipticCurve_finite_field'>
sage: Point = E2.map_to_group(2107,2107,'test',17)
sage: Point in E2
True
```

E.4 BLSSignatureScheme class

The BLSSignatureScheme is implemented as a BLSSignatureCheme object making it easier to store it for later use and define functions on the class.

E.4.1 Parameters

The BLSSignatureScheme object is initailised with parameters :

- g_1: The generator (a point) of curve subgroup G_1.
- g_2: The generator (a point) of curve subgroup G_2.
- m: The base curve order $m = |E(\mathbb{F}_q)|$.
- n: The subgroup prime order $n = |G_1| = |G_2|$.

When the object lives you have the following functions available

E.4.2 Functions

The Object then have the following functions available:

- `generate_key_pair`: Generates and stores a private and public key in object variables
  `self.private_key`, `self.public_key`.
- `sign`: takes a string and returns the signature (an element in $\mathbb{F}_q$), signature is also stored in object variable
  `self.signature`.
- `sign_file`: equivalent of the above, Takes folder paths to a text file to sign and a file to store pickled signature in.
- `verify`: takes a string and a signature (an element in $\mathbb{F}_q$) and returns a boolean.
- `verify_file`: equivalent of the above, Takes folder paths to a text file and a signature file containing pickled signature.
- `export_key_pair_to_files`: takes folder paths to two files for storing pickled public and private key in.

- `set_map_to_group_stop_parameter`: takes an integer. Possibility to change the map to group stop parameter which is initialised default to 17.
- `set_public_key`: takes a point in G_2 and sets the variable `self.public_key`.
- `set_private_key`: takes an element in $\mathbb{Z}_p$ and sets the variable `self.private_key`.
- `set_public_key_from_file`: takes path to file with a pickled public key and sets the variable `self.public_key`.
- `set_private_key_from_file`: takes path to file with a pickled private key and sets the variable `self.private_key`.
- `reset_key_pair`: resets the object variables `self.private_key`, `self.public_key` to the latest generated.

E.4.3 BLS outside Sage - almost

You can of use BLS in the Sage notebook mode but more interesting you can access sage functionality from .sage scripts. Ive attached the script `bls_script.sage`.

Make sure that Sage is in your computer's root path, i.e. in a MAC OS X Terminal write

```
PATH=$PATH:/Applications/sage/
```

Now you can run the script by running the command

```
sage .../bls_script.sage
```

which will present you with a 'nice' BLS UI with some options.

```
-------------------------------------------------------
BLS short signature system
-------------------------------------------------------

please write path to BLSxx.sobj file or press 0 to exit
```

```
:../BLS_objects/BLSMNT.sobj

BLSxx.sobj file loaded!

please select an option (0-7) followed by enter:

0) exit.
1) generate key pair
2) sign message
3) validate signature
4) export key pair
5) set public key
6) set private key
7) reset key pair
```

The possibility of scripting can in fact with MAC OS X folder actions feature make this signature scheme practical applicable between users. The folder action feature makes MAC OSX able to perform a scripted task on a file dropped into a folder e.g. signing it and attaching the file and signature in an email.

E.4.4 Attached examples

I've attached some .sobj files on the cd that can be loaded using the Sage load command discussed in Appendix A such that you do not need to create parameters to instantiate the scheme with.

The examples are:

- BLS17.sobj - Supersingular elliptic curve over $\mathbb{F}_{3^{17}}$
- BLS53.sobj - Supersingular elliptic curve over $\mathbb{F}_{3^{53}}$
- BLS79.sobj - Supersingular elliptic curve over $\mathbb{F}_{3^{79}}$
- BLS97.sobj - Supersingular elliptic curve over $\mathbb{F}_{3^{97}}$
- BLSMNT.sobj - MNT elliptic curve over $\mathbb{F}_p$, large prime p

Appendix F

Code

F.1 Sage interact: Point addition on elliptic curve

```
def point_txt(P,name,rgbcolor):
    if (P.xy()[1]) < 0:
        r = text(name,[float(P.xy()[0])-0.5,float(P.xy()[1])-0.5],rgbcolor=rgbcolor)
    elif P.xy()[1] == 0:
        r = text(name,[float(P.xy()[0])-0.5,float(P.xy()[1])+0.5],rgbcolor=rgbcolor)
    else:
        r = text(name,[float(P.xy()[0])-0.5,float(P.xy()[1])+0.5],rgbcolor=rgbcolor)
    return r

E = EllipticCurve([-2,0])
list_of_points = [E(0,0),E(-1,-1),E(-1,1),E(2,2),E(2,-2),E(9/4,-21/8),E(9/4,21/8),E(-8/9,28/27),E(-8/9,-28/27)]
html("Graphical addition of two points $P$ and $Q$ on the curve $ E: %s $"%latex(E))
@interact
def _(P=selector(list_of_points,default=list_of_points[2],label='Point P'),Q=selector(list_of_points,default=list_of_points[2],label='Point Q'), marked_points = checkbox(default=True,label = 'Points'), lines_on = checkbox(default=True,label = 'Lines'), Axes=True):
    if lines_on:
        Lines = 2
    else:
        Lines = 0
```

```
    curve = E.plot(rgbcolor = (0,0,1),xmin=25,xmax=25,plot_points=300)
    R = P + Q
    Rneg = -R
    if R == E(0):
        l1 = line_from_curve_points(E,P,Q)
        p1 = plot(P,rgbcolor=(1,0,0),pointsize=40)
        p2 = plot(Q,rgbcolor=(1,0,0),pointsize=40)
        textp1 = point_txt(P,"$P$",rgbcolor=(0,0,0))
        textp2 = point_txt(Q,"$Q$",rgbcolor=(0,0,0))
        if Lines==0:
            g=curve
        elif Lines ==1:
            g=curve+l1
        elif Lines == 2:
            g=curve+l1
        if marked_points:
            g=g+p1+p2
        if P != Q:
            g=g+textp1+textp2
        else:
            g=g+textp1
    else:
        l1 = line_from_curve_points(E,P,Q)
        l2 = line_from_curve_points(E,R,Rneg,style='--')
        p1 = plot(P,rgbcolor=(1,0,0),pointsize=40)
        p2 = plot(Q,rgbcolor=(1,0,0),pointsize=40)
        p3 = plot(R,rgbcolor=(1,0,0),pointsize=40)
        p4 = plot(Rneg,rgbcolor=(1,0,0),pointsize=40)
        textp1 = point_txt(P,"$P$",rgbcolor=(0,0,0))
        textp2 = point_txt(Q,"$Q$",rgbcolor=(0,0,0))
        textp3 = point_txt(R,"$P+Q$",rgbcolor=(0,0,0))
        if Lines==0:
            g=curve
        elif Lines ==1:
            g=curve+l1
        elif Lines == 2:
            g=curve+l1+l2
        if marked_points:
            g=g+p1+p2+p3+p4
        if P != Q:
            g=g+textp1+textp2+textp3
        else:
            g=g+textp1+textp3
    g=g+text("$P+Q=%s$"%R,[-3,-3], rgbcolor=(0,0,0),horizontal_alignment="left")
    g=g+text("$E:\ %s$"%latex(E),[-3,3],horizontal_alignment="left")
    g.axes_range(xmin=-3,xmax=3,ymin=-3,ymax=3)
    show(g,axes = Axes)

def line_from_curve_points(E,P,Q,style='-',rgb=(1,0,0),length=25):
    """
```

```
    P,Q two points on an elliptic curve.
    Output is a graphic representation of the straight line intersecting with P,Q.
    """
    # The function tangent to P=Q on E
    if P == Q:
        if P[2]==0:
            return line([(1,-length),(1,length)],linestyle=style,rgbcolor=rgb)
        else:
            # Compute slope of the curve E in P
            [a1, a2, a3, a4, a6] = E.a_invariants()
            numerator = (3*P[0]**2 + 2*a2*P[0] + a4 - a1*P[1])
            denominator = (2*P[1] + a1*P[0] + a3)
            if denominator == 0:
                return line([(P[0],-length),(P[0],length)],linestyle=style,rgbcolor=rgb)
            else:
                l = numerator/denominator
                f(x) = l * (x - P[0]) + P[1]
                return plot(f(x),-length,length,linestyle=style,rgbcolor=rgb)
    # Trivial case of P != R where P=O or R=O then we get the vertical line from the other point
    elif P[2] == 0:
        return line([(Q[0],-length),(Q[0],length)],linestyle=style,rgbcolor=rgb)
    elif Q[2] == 0:
        return line([(P[0],-length),(P[0],length)],linestyle=style,rgbcolor=rgb)
    # Non trivial case where P != R
    else:
        # Case where x_1 = x_2 return vertical line evaluated in Q
        if P[0] == Q[0]:
            return line([(P[0],-length),(P[0],length)],linestyle=style,rgbcolor=rgb)

        #Case where x_1 != x_2 return line trough P,R evaluated in Q"
        l=(Q[1]-P[1])/(Q[0]-P[0])
        f(x) = l * (x - P[0]) + P[1]
        return plot(f(x),-length,length,linestyle=style,rgbcolor=rgb)
```

F.2 Sage patch: Map to group

```
def map_to_group(self,m,n,msg,r):
    r"""
    Hash a message using sha1 and map it onto a point a subgroup of the curve.

    INPUT:
        self -- elliptic curve over finite field.
        m -- self.order(), given as a parameter to reduce computations.
        n -- order subgroup G_1.
        msg -- string to hash.
        r -- stop parameter, upper bound in number of runs.

    OUTPUT:
        PM -- non-trivial point on curve E(F_q) of order p.

    EXAMPLE:
        sage: F.<a>=GF(17^3)
        sage: E = EllipticCurve(F,[0,0,0,2,1])
        sage: n=E.cardinality()
        sage: p=[s for s,e in n.factor()].pop()
        sage: P = E.map_to_group(n,p,'test',17)
        sage: P
        (a^2 + a + 7 : 6*a^2 + 6*a + 13 : 1)
        sage: P.curve() == E
        True

    NOTES:
        Do not work with order n when $E = Z_n \times Z_n$.??
        When over a field of char. p != 2 then the elliptic curve have to be on form $E: y^2 = x^3+a_2x^2+a_4x+a_6$.
        When over a field of char. p = 2 then the elliptic curve have to be on form $E: y^2 + y = x^3+a_2x^2+a_4x+a_6$.
        The string "test1" breaks it.
        REFERENCES:
            [BLS04] Dan Boneh, Ben Lynn, and Hovav Shacham. "Short signatures from the weil pairing". J. Cryptol., 17(4), 2004.


    AUTHOR:
        - David Hansen (2009-01-25)
    """
    F = self.base_field()
    p = F.characteristic()

    # check that curve in on short weierstrass form
    if p == 2:
```

```
            if (not self.a1() == 0 and self.a3 == 1):
                raise Warning, "map_to_group: elliptic curve over field of char. p=2 is not on form y**2 + y"
        else:
            if (not self.a1() == 0 and self.a3 == 0):
                raise Warning, "map_to_group: elliptic curve over field of char. p!=2 is not on form y**2"

        nn = F.cardinality()

        # check that base field is not to large
        if nn.nbits() > 159:
            raise Warning, "map_to_group: base field is to large"

        # character translation from hex to binary form
        convert = {
            "0" : "0000",
            "1" : "0001",
            "2" : "0010",
            "3" : "0011",
            "4" : "0100",
            "5" : "0101",
            "6" : "0110",
            "7" : "0111",
            "8" : "1000",
            "9" : "1001",
            "a" : "1010",
            "b" : "1011",
            "c" : "1100",
            "d" : "1101",
            "e" : "1110",
            "f" : "1111"}
        i=0
        s=r.nbits()
        while i<=s:
            # First we hash the message plus a bit i
            msg_hash_hex_str = hashlib.sha1(str(i)+msg).hexdigest()
            msg_hash_bit_str = ''
            for hexletter in msg_hash_hex_str:
                msg_hash_bit_str += convert[hexletter]
            t = int(msg_hash_bit_str[:-1],2)%nn
            # coerce x into an field element by coercing into coefficients
            x = sum([F.gen()**k*c for k, c in enumerate(t.digits(F.characteristic()))])
            #use last bit for random bit b
            b = Integer(msg_hash_bit_str[159])
            f=x**3+self.a2()*x**2+self.a4()*x+self.a6()
            # In the char p=2 case
            if p == 2:
                if f.trace() == 0:
                    # find a theta with trace 1
                    theta = F.random_element()
                    for i in range(1,20):
```

```
                    if theta.trace() == 1:
                        break
                    else:
                        theta = F.random_element()
                f1 = 0
                f2 = f
                theta1 = theta**2
                sol1 = 0
                for i in range(0,F.degree()-1):
                    f1 += f2
                    sol1 += f1*theta1
                    theta1 = theta1**2
                    f2 = f2**2
                sol= [sol1,sol1+1]
                PMT=self(x,sol[b])
                PM=Integer(m/n)*PMT
                if PM!=self(0):
                    return PM
        else:
            if f.is_square():
                square_roots=f.sqrt(all=True)
                PMT=self(x,square_roots[b])
                PM=Integer(m/n)*PMT
                if PM!=self(0):
                    return PM
        i=i+1
    raise Warning, "map_to_group: unsuccessful"
```

F.3 Sage patch: Weil pairing

This is an excerpt of the sage source code file `ell_point.py`.

```
def _line_(self,R,Q):
    r"""
    Computes a straight line through points self and R evaluated in point Q.

    INPUT:
        R -- a point on self.curve()
        Q -- a point on self.curve()

    OUTPUT:
        An element in the base field self.curve().base_field()

    EXAMPLE:
        sage: F.<a>=GF(2^5)
        sage: E=EllipticCurve(F,[0,0,1,1,1])
        sage: P = E(a^4 + 1, a^3)
        sage: Q = E(a^4, a^4 + a^3)
        sage: O = E(0)
        sage: P._line_(P,-2*P) == 0
        True
        sage: P._line_(Q,-(P+Q)) == 0
        True
        sage: O._line_(O,Q) == F(1)
        True
        sage: P._line_(O,Q) == a^4 - a^4 + 1
        True
        sage: P._line_(13*P,Q) == a^4
        True
        sage: P._line_(P,Q) == a^4 + a^3 + a^2 + 1
        True

    NOTES:
        Cover all posible point combination cases.
        The function is used in _miller_ algorithm.

    AUTHOR:
        - David Hansen (2009-01-25)
    """
    if self.is_zero() or R.is_zero():
        if self == R:
            return self.curve().base_field().one_element()
        if self.is_zero():
            return Q[0] - R[0]
        if R.is_zero():
            return Q[0] - self[0]
    elif self != R:
        if self[0] == R[0]:
            return Q[0] - self[0]
```

```
        else:
            l = (R[1] - self[1])/(R[0] - self[0])
            return Q[1] - self[1] - l * (Q[0] - self[0])
    else:
        [a1, a2, a3, a4, a6] = self.curve().a_invariants()
        numerator = (3*self[0]**2 + 2*a2*self[0] + a4 - a1*self[1])
        denominator = (2*self[1] + a1*self[0] + a3)
        if denominator == 0:
            return Q[0] - self[0]
        else:
            l = numerator/denominator
            return Q[1] - self[1] - l * (Q[0] - self[0])

def _miller_(self,Q,n):
    r"""
    Compute the value of the rational function $f_{n,P}(Q)$, where divisor $div(f_{n,P})=n[P]-n[O]$.

    INPUT:
        Q -- a point on self.curve()
        n -- an integer such that n*self = n*Q = (0:1:0)

    OUTPUT:
        t -- An element in the base field self.curve().base_field()

    EXAMPLE:
        sage: F.<a>=GF(2^5)
        sage: E=EllipticCurve(F,[0,0,1,1,1])
        sage: P = E(a^4 + 1, a^3)
        sage: Fx.<b>=GF(2^(4*5))
        sage: Ex=EllipticCurve(Fx,[0,0,1,1,1])
        sage: phi=Hom(F,Fx)(F.gen().minpoly().roots(Fx)[0][0])
        sage: Px=Ex(phi(P.xy()[0]),phi(P.xy()[1]))
        sage: Qx = Ex(b^19 + b^18 + b^16 + b^12 + b^10 + b^9 + b^8 + b^5 + b^3 + 1, b^18 + b^13 + b^10 + b^8 + b^5 + b^4 + b^3 + b)
        sage: Px._miller_(Qx,41) == b^17 + b^13 + b^12 + b^9 + b^8 + b^6 + b^4 + 1
        True
        sage: Qx._miller_(Px,41) == b^13 + b^10 + b^8 + b^7 + b^6 + b^5
        True

    Example on even order n
        sage: F.<a> = GF(19^4)
        sage: E = EllipticCurve(F,[-1,0])
        sage: P = E(15*a^3 + 17*a^2 + 14*a + 13,16*a^3 + 7*a^2 + a + 18)
        sage: Q = E(10*a^3 + 16*a^2 + 4*a + 2, 6*a^3 + 4*a^2 + 3*a + 2)
        sage: x=P.weil_pairing(Q,360)
        sage: x^360 == F(1)
```

```
        True

    You can use the _miller_ function on lin dep points, but with the risk of a dividing with zero.
        sage: Px._miller_(2*Px,41)
        Traceback (most recent call last):
        ...
        ZeroDivisionError: division by zero in finite field.

    NOTES:
        Implemented with double-and-add.
        The function requires access to the _line_ function.
        REFERENCES:
            [Mil04] Victor S. Miller, "The Weil pairing, and its efficient calculation", J. Cryptol., 17(4):235-261, 2004

    AUTHOR:
        - David Hansen (2009-01-25)

    """
    t = self.curve().base_field().one_element()
    V = self
    S = 2*V
    nbin = n.bits()
    i = n.nbits() - 2
    while i > -1:
        S = 2*V
        t = (t**2)*(V._line_(V,Q)/S._line_(-S,Q))
        V = S
        if nbin[i] == 1:
            S = V+self
            t=t*(V._line_(self,Q)/S._line_(-S,Q))
            V = S
        i=i-1
    return t

def weil_pairing(self, Q, n):
    r"""
    Compute the Weil pairing of self and Q using Miller's algorithm.

    INPUT:
        Q -- a point on self.curve()
        n -- an integer such that n*self = n*Q = (0:1:0)

    OUTPUT:
        An n'th root of unity in the base field self.curve().base_field()

    EXAMPLE:
        sage: F.<a>=GF(2^5)
        sage: E=EllipticCurve(F,[0,0,1,1,1])
        sage: P = E(a^4 + 1, a^3)
```

```
        sage: Fx.<b>=GF(2^(4*5))
        sage: Ex=EllipticCurve(Fx,[0,0,1,1,1])
        sage: phi=Hom(F,Fx)(F.gen().minpoly().roots(Fx)[0][0])
        sage: Px=Ex(phi(P.xy()[0]),phi(P.xy()[1]))
        sage: O = Ex(0)
        sage: Qx = Ex(b^19 + b^18 + b^16 + b^12 + b^10 + b^9 + b^8 + b^5 + b^3 + 1, b^18 + b^13 + b^10 + b^8 + b^5 + b^4 + b^3 + b)
        sage: Px.weil_pairing(Qx,41) == b^19 + b^15 + b^9 + b^8 + b^6 + b^4 + b^3 + b^2 + 1
        True
        sage: Px.weil_pairing(17*Px,41) == Fx(1)
        True
        sage: Px.weil_pairing(O,41) == Fx(1)
        True

    In this simple implementation we only allow points of same order.
        sage: Px.weil_pairing(O,40)
        Traceback (most recent call last):
        ...
        ValueError: P and Q do not both have order n

    NOTES:
        Implemented using proposition 8 in [Mil04].
        The function requires access to the _miller_ function.
        In the case where lin. dep. input leads to division with zero, the error is catched and the 1 is returned.
        Use try-catch instead of doing discrete log test for linear dependence, since this is much to slow for large n.
        REFERENCES:
            [Mil04] Victor S. Miller, "The Weil pairing, and its efficient calculation", J. Cryptol., 17(4):235-261, 2004

    AUTHOR:
        - David Hansen (2009-01-25)
    """
    # Test is both P, Q is in E[n]
    if not ((n*self).is_zero() and (n*Q).is_zero()):
        raise ValueError, "P and Q do not both have order n"

    # Case where P = Q
    if self == Q:
        return self.curve().base_field().one_element()

    # Case where P = O or Q = O
    if self.is_zero() or Q.is_zero():
        return self.curve().base_field().one_element()

    # The non-trivial case P != Q
    try:
```

```
        r = ((-1)**n.test_bit(0))*(self._miller_(Q,n)/Q._miller_(self,n))
        return r
    except ZeroDivisionError, detail:
        return self.curve().base_field().one_element()
```

F.4 Sage sample: Weil pairing example

This code is used in connection with Example 3.32

```
##This is data for an example of a Weil pairing using a supersingular curve over F_{2^7}##
F2=GF(2^28,'b')
b=F2.gen()
E2=EllipticCurve(F2,[0,0,1,1,1])

##Choose points P,Q of torsion 113##
P=E2(b^27 + b^26 + b^25 + b^23 + b^22 + b^18 + b^15 + b^13 + b^12 + b^7 + b^6 + b^3 + 1 , b^25 + b^24 + b^22 + b^19 + b^16 + b^14 + b^13 + b^12 + b^7 + b^4 + b^2 + 1 )
Q=E2(b^26 + b^25 + b^24 + b^22 + b^20 + b^17 + b^16 + b^15 + b^13 + b^11 + b^8 + b^7 + b^6 + b^5 + b^3 + b^2 + b , b^27 + b^25 + b^22 + b^21 + b^20 + b^19 + b^18 + b^16 + b^15 + b^14 + b^13 + b^11 + b^6 + b^3 + b^2 + 1 )

##e_113(P,Q)=b^25 + b^17 + b^14 + b^11 + b^10 + b^4##
```

F.5 Sage sample: MNT curve

```
#This is test data for the BLS signature scheme using the Weil pairing on an MNT curve
#Data is taken from article "Generating more elliptic MNT curves" by Scott and Barreto.
D=62003
q=625852803282871856053922297323874661378036491717
h=3
r=208617601094290618684641029477488665211553761021
B=423976005090848776334332509669574781621802740510
m=625852803282871856053923088432465995634661283063
#Beware line below was manually broken for typesetting reasons.
m2=600942903564084071309841611273100785163600318684
17968262992864809623507269833854678414046779817844
85375702685877496633143419825751245799329327184904
36646551464432290290694633920468378302679942227891
60047337432075266619082657640364986415435746294498
140589844832666082434658532589211525696
F1=FiniteField(q)
k=6
F2=FiniteField(q^k,'b')
E1=EllipticCurve(F1,[0,0,0,-3,B])
E2=EllipticCurve(F2,[0,0,0,-3,B])
n=r
#Since curve order is prime
P1=int(m/n)*E1.random_point()
```

```
if n*P1!=E1(0):
    print "P do not generate G_1, please reload"
phi = Hom(F1,F2)(F1.gen())
P2=E2(P1)
Q=int(m2/(n**2))*E2.random_point()
```

F.6 Sage sample: MOV reduction example

This code is used in connection with Example 4.12

```
#This is data for an example of a mov reduction using a supersingular curve over F_{2^7}
q=2^7
F1=GF(q,'a')
k=4
F2=GF(q^k,'b')
phi=Hom(F1,F2)(F1.gen().minpoly().roots(F2)[0][0])
E1=EllipticCurve(F1,[0,0,1,1,1])
E2=EllipticCurve(F2,[0,0,1,1,1])
```

F.7 Magma script: Timing of logarithm computations

```
// File:          magma_logarithm_timing.m
// Description:       This is magma code for timing logarithms in fields and curve groups over fields of characteristic 2. It looks at EllipticCurve([0,0,1,1,0]) over fields of size 2^m, (m mod 8) odd.
// Note:File is loaded with the cmd: load "E2008/Speciale/Magma/mov_attack_timing.m";
//
// Timing of how long it takes to do discrete log problem in curve group and in extension field efter reduction
// ns runs in each finite field of size 2^(m1)< 2^m < 2^(m2).
//
// E1 : EllipticCurve([0,0,1,1,0])
// E2 : EllipticCurve([0,0,1,1,1])
//
//
timing:= function(ns,m1,m2)
/////////////////////////////////////////////
// Determine the number of points on the elliptic curve E1 / E2 over F_2^m, m odd
/////////////////////////////////////////////
size := function(h)
m:=h mod 8;
if IsEven(m) then
return 0;
end if;
if (m eq 1) or (m eq 7) then
return Floor(2^h+1+Sqrt(2^(h+1))); // switch sign on square when changing curve
end if;
if (m eq 3) or (m eq 5) then
return Floor(2^h+1-Sqrt(2^(h+1))); // switch sign on square when changing curve
end if;
end function;
/////////////////////////////////////////////
// MOV reduction on elliptic curve E on point R=l*P returns extension field elements alpha and beta.
/////////////////////////////////////////////
mov_reduction:=function(E1,n,p,ndp1, R0, P0)
P1 := E1!P0;
R1 := E1!R0;
repeat
Q1 := ndp1*Random(E1);
alpha := WeilPairing(P1,Q1,p);
until Order(alpha) eq p;
beta := WeilPairing(R1,Q1,p);
return [alpha,beta];
```

```
end function;
////////////////////////////////////////////
// Rest of the timing function.
////////////////////////////////////////////
L:=[]; //empty list to store results in
// E := EllipticCurve([0,0,1,1,1]);  // E2
E := EllipticCurve([0,0,1,1,0]);  // E1
for i:= m1 to m2 do
//////////////////////////////////////////
// Setup fields and curves...
//////////////////////////////////////////
F0:=FiniteField(2^i);
F1:=FiniteField(2^(4*i));
n:=size(i);
if n ne 0 then
factors:=Factorization(n);
p:=factors[#factors,1];
ndp0:=Floor(n/p); // We will need |E0| divided by p several ↩
    times
ndp1:=Floor((2^(i*2)+1)/p); // We will need |E1| divided by p ↩
    several times
E0:=ChangeRing(E,F0);
E1:=BaseExtend(E0,F1);
P0:=ndp0*Random(E0);
while P0 eq E0!0 do
P0:=ndp0*Random(E0);
end while;
//////////////////////////////////////////
// coppersmith index calculus precomputation...
//////////////////////////////////////////
w:=PrimitiveElement(F1);
t:=Cputime();
l1:=Log(w,w^(-1));
time_precomp_coppersmith:=Cputime(t);
//////////////////////////////////////////
// Do the logarithms ns times....
//////////////////////////////////////////
total_time_in_E0:=0;
total_time_to_reduce:=0;
total_time_in_F1:=0;
for j:=1 to ns do
repeat
l:=Random(p);
until l ge 1;
R0:=l*P0;
//////////////////////////////////////////
// Logarithm in E0...
//////////////////////////////////////////
t:=Cputime();
l1:=Log(P0,R0);
t:=Cputime(t);
total_time_in_E0:=total_time_in_E0+t;
//////////////////////////////////////////
// Reduction to F1....
```

```
/////////////////////////////////////////
t:=Cputime();
elements:=mov_reduction(E1,n,p,ndp1, R0, P0);
t:=Cputime(t);
total_time_to_reduce:=total_time_to_reduce+t;
/////////////////////////////////////////
// Logarithm in F1....
/////////////////////////////////////////
t:=Cputime();
l2:=Log(elements[1],elements[2]);
t:=Cputime(t);
total_time_in_F1:=total_time_in_F1+t;
end for;
/////////////////////////////////////////
// store results....
/////////////////////////////////////////
x:=Real(total_time_in_E0/ns);
y:=Real(total_time_to_reduce/ns);
z:=Real(total_time_in_F1/ns);
u:=time_precomp_coppersmith;
T:=[i,x,y,z,u];
L:=Append(L,T);
/////////////////////////////////////////
// print results ....
/////////////////////////////////////////
print i;
// printf "time to do log in E0 over F_2^";print i;printf ":";
// print x;
// printf "time to do reduce log in E0 to F_2^";print 4*i;printf ":";
// print y;
// printf "time to do log in finite extension field F_2^";print 4*i;printf ":";
// print z;
end if;
end for;
return L;
end function;
/////////////////////////////////////////
// print function for above list, prints list with either x,y,z,u for n=2,3,4,5
/////////////////////////////////////////
print_lists:=function(L,n)
l:=#L;
printf "L"; print (n-1);printf "="; printf "[";for i:=1 to l do printf "["; print L[i,1]; printf ","; print L[i,n]; printf "]"; if i ne l then printf ","; end if;end for;printf "]";
return 0;
end function;
```

F.8 Sage plot: Plot of time complexity for logarithm computations

```
def plot_approx_graph(p,c=1.4, upper_limit =200):
    if p == 2:
        prime_orders_E1_E2 = []
        large_prime_factor_E1_E2 = []
        group_order = []
        maximum1 = 0
        for i in range(2,upper_limit):
            m = i
            if not is_even(m):
                q = 2^m
                N1 = q + 1 + 2^((m+1)/2)
                N2 = q + 1 - 2^((m+1)/2)
                F1 = factor(N1)
                F1_largest_factor = F1[len(F1)-1][0]
                F2 = factor(N2)
                F2_largest_factor = F2[len(F2)-1][0]
                maximum2 = max(F1_largest_factor,F2_largest_factor)
                if maximum2 > maximum1:
                    maximum1 = maximum2
                if N1.is_prime() :
                    prime_orders_E1_E2.append([m,0])
                elif F1_largest_factor.bits()>40 :
                    large_prime_factor_E1_E2.append([m,0])
                if N2.is_prime() :
                    prime_orders_E1_E2.append([m,0])
                elif F2_largest_factor.bits()>40 :
                    large_prime_factor_E1_E2.append([m,0])
                if N1.is_prime() or N2.is_prime() or maximum1.bits>40:
                    group_order.append([m,maximum1])
        curve_tc = []
        for i in range(0,len(group_order)):
            curve_tc.append([group_order[i][0],log(sqrt(group_order[i][1]))])
        # plot
        field_ext_tc_4 = []
        for i in range(0,upper_limit):
            m=i*1.0
            field_ext_tc_4.append([m,(c*(m*4)^(1/3)*log(m*4)^(2/3))])
        curve_tc_lin =line(curve_tc,rgbcolor=(1,0,0))
        curve_tc_lin_text = text('Pollard\'s rho method in curve group $E(F_{2^m})$',(175,log(sqrt(maximum1))),rgbcolor=(1,0,0))
        field_ext_tc_4_lin=line(field_ext_tc_4,rgbcolor=(0,0.75,0))
```

```
        field_ext_tc_4_lin_text=text('index calculus in field $F_{2^{4m}}$',(275,field_ext_tc_4[upper_limit-3][1]),rgbcolor=(0,0,1))
        g=curve_tc_lin+field_ext_tc_4_lin
        #g=g+field_ext_tc_4_lin_text+curve_tc_lin_text
    elif p == 3:
        prime_orders_E1_E2 = []
        large_prime_factor_E1_E2 = []
        group_order = []
        maximum1 = 0
        for i in range(3,upper_limit):
            m = i
            if not is_even(m):
                q = 3^m
                N1 = q + 1 + 3^((m+1)/2)
                N2 = q + 1 - 3^((m+1)/2)
                F1 = factor(N1)
                F1_largest_factor = F1[len(F1)-1][0]
                F2 = factor(N2)
                F2_largest_factor = F2[len(F2)-1][0]
                maximum2 = max(F1_largest_factor,F2_largest_factor)
                if maximum2 > maximum1:
                    maximum1 = maximum2
                if N1.is_prime() :
                    prime_orders_E1_E2.append([m,0])
                elif F1_largest_factor.bits()>40 :
                    large_prime_factor_E1_E2.append([m,0])
                if N2.is_prime() :
                    prime_orders_E1_E2.append([m,0])
                elif F2_largest_factor.bits()>40 :
                    large_prime_factor_E1_E2.append([m,0])
                if N1.is_prime() or N2.is_prime() or maximum1.bits>40:
                    group_order.append([m,maximum1])
        curve_tc = []
        for i in range(0,len(group_order)):
            curve_tc.append([group_order[i][0],log(sqrt(group_order[i][1]))])
        #plot
        field_ext_tc_6 = []
        for i in range(0,upper_limit):
            m=i*1.0
            field_ext_tc_6.append([m,(c*(m*6)^(1/3)*log(m*6)^(2/3))])
        curve_tc_lin =line(curve_tc,rgbcolor=(1,0,0))
        curve_tc_lin_text = text('Pollard\'s rho method in curve group $E(F_{3^m})$',(175,log(sqrt(maximum1))),rgbcolor=(1,0,0))
        field_ext_tc_6_lin=line(field_ext_tc_6,rgbcolor=(0,0.75,0))
        field_ext_tc_6_lin_text=text('index calculus in field $F_{3^{6m}}$',(275,field_ext_tc_6[upper_limit-3][1]),rgbcolor=(0,0.5,0))
```

```
        g=curve_tc_lin+field_ext_tc_6_lin
        #g=g+field_ext_tc_6_lin_text+curve_tc_lin_text
    else:
        return 0
    #general plot setting
    #g.axes_labels(['m','operations'])
    g.axes_range(xmin = 20,xmax=upper_limit+100,ymin=0,ymax=60)
    g.show()
    return g
```

F.9 Sage patch: BLS signature scheme

```
from sage.categories.homset import Hom
from sage.structure.sage_object import save, load
import sage.rings.all as rings

class BLSSignatureScheme():
    r"""
    The BLS short signature scheme

    EXAMPLE:

    NOTES:
        REFERENCES:
            [BLS04] Dan Boneh, Ben Lynn, and Hovav Shacham. "↩
                Short signatures from the weil pairing". J. ↩
                Cryptol., 17(4), 2004.

    AUTHOR:
       - David Hansen (2009-01-25)
    """

    def __init__(self, g1, g2, m, n):
        r"""
        Constructor for BLSSignatureScheme class

        PARAMETERS:
            g1 -- generator for group $G_1 \in E(F_q)$.
            g2 -- generator for group $G_2 \in E(F_{q^k})$.
            m -- cardinality $m = |E(F_q)|$.
            n -- prime order $n = |G_1| = |G_2|$.

        NOTES:
           Asumes that all parameters are a valid set.
        """

        # TODO: Need to check the given parameters.

        self.g1 = g1

        self.gx2 = g2
```

```
        self.prime_order = n
        self.E_cardinality = m

        self.E = self.g1.curve()
        self.F = self.E.base_field()
        self.Ex = self.gx2.curve()
        self.Fx = self.Ex.base_field()

        # We have to distinguish in how we build phi
        if self.F.order().is_prime():
            self.phi = Hom(self.F,self.Fx)(self.F.gen())
        else:
            self.phi = Hom(self.F,self.Fx)(self.F.gen().minpoly↩
                ().roots(self.Fx)[0][0])
        self.gx1 = self.Ex(self.phi(self.g1.xy()[0]),self.phi(↩
            self.g1.xy()[1]))
        self.prime_field = rings.FiniteField(n)

        self.map_to_group_stop_parameter = rings.Integer(17)
        self.public_key = None
        self.private_key = None
        self.signature = None
        self.point_hash =None

    # Some get methods for the above variables.
    # Or you can just call variables on the class object ↩
        directly.
    def public_key(self):
        return self.public_key
    def private_key(self):
        return self.private_key
    def signature(self):
        return self.signature
    def point_hash(self):
        return self.point_hash

    def generate_key_pair(self):
        r"""
        Generates a private and public key using the given ↩
            parameters.

        EXAMPLE:

        NOTES:
            Set value of public and private key on signaure ↩
                class.
        """

        _x = self.prime_field(0)

        # choose randomly a non-trivial value x in ZZ_p as the ↩
            private key
        while _x == 0 or _x == 1:
            _x = self.prime_field.random_element()
```

```
        self.generated_private_key = _x

        # multiply x with generator g2 to get publick key V in G2
        self.generated_public_key = int(_x)*self.gx2

        # reset key pair to the latest in class generated pair
        self.reset_key_pair()

    def sign(self, msg, priv):
        r"""
        sign a string and return the signature in F

        INPUT:
            msg -- string to sign
            priv -- private key for signing (OPTIONAL)

        OUTPUT:
            signaure -- element in G1 base field F

        EXAMPLE:

        NOTES:
        """

        if priv == None:
            raise Warning, "Please generate or set a private key before signing"
        self.point_hash = self.E.map_to_group(self.E_cardinality, self.prime_order ,msg , self.map_to_group_stop_parameter)
        _sigma = rings.Integer(priv)*self.point_hash
        self.signature = _sigma.xy()[0]
        return self.signature

    def sign_file(self, message_file, signature_file):
        r"""
        sign the message_file with the private_key and store signature in signature file

        INPUT:
            message_file -- string containing path to a textfile.
            signature_file -- string containing path path to a .sobj signature file.

        EXAMPLE:

        NOTES:
        """

        if self.private_key == None:
            raise Warning, "Please generate or set a private key before signing"
```

```

        # load message from file
        fm = open(message_file)
        msg = fm.read()
        fm.close()

        # Hash message to point on curve
        self.point_hash = self.E.map_to_group(self.E_cardinality, self.prime_order ,msg , self.map_to_group_stop_parameter)
        _sigma = rings.Integer(self.private_key)*self.point_hash
        self.signature = _sigma.xy()[0]

        # save signature to file
        save(self.signature, signature_file)

    def validate(self,msg,sig, pub):
        r"""
        validate a message string signature in F

        INPUT:
            msg -- string
            sig -- signature in F
            pub -- public key (OPTIONAL)

        OUTPUT:
            bool

        EXAMPLE:

        NOTES:
        """
        sign = self.phi(sig)
        if self.Ex.is_x_coord(sign):
            _sigma = self.Ex.lift_x(sign)
            if self.prime_order*_sigma == self.Ex(0):
                _R1 = self.E.map_to_group(self.E_cardinality, self.prime_order , msg,self.map_to_group_stop_parameter)
                _R2 = self.Ex(self.phi(_R1.xy()[0]),self.phi(_R1.xy()[1]))
                _e1 = _sigma.weil_pairing(self.gx2, self.prime_order)
                _e2 = _R2.weil_pairing(pub,self.prime_order)
                if _e1==_e2 or _e1**(-1)==_e2:
                    return True
                else:
                    return False
        else:
            return False


    def validate_file(self, message_file, signature_file):
```

```
        r"""
        validate the message_file's signature file

        INPUT:
            message_file -- string containing path to a textfile.
            signature_file -- string containing path path to a .sobj signature file.

        EXAMPLE:

        NOTES:
        """

        if self.public_key == None:
            raise Warning, "Please generate or set a public key before validating"

        # load message and signature from files
        fm = open(message_file)
        msg = fm.read()
        fm.close()

        sig = load(signature_file)

        sign = self.phi(sig)

        # validation
        if self.Ex.is_x_coord(sign):
            _sigma = self.Ex.lift_x(sign)
            if self.prime_order*_sigma == self.Ex(0):
                _R1 = self.E.map_to_group(self.E_cardinality, self.prime_order , msg, self.map_to_group_stop_parameter)
                _R2 = self.Ex(self.phi(_R1.xy()[0]),self.phi(_R1.xy()[1]))
                _e1 = _sigma.weil_pairing(self.gx2, self.prime_order)
                _e2 = _R2.weil_pairing(self.public_key,self.prime_order)
                if _e1==_e2 or _e1**(-1)==_e2:
                    return True
                else:
                    return False
        else:
            return False

    def export_key_pair_to_files(self, private_key_file, public_key_file):
        r"""
        export the key pair to a .sobj private and a .sobj public key file
        """

```

```python
        save(self.private_key, private_key_file)
        save(self.public_key, public_key_file)

    def set_map_to_group_stop_parameter(self,val):
        self.map_to_group_stop_parameter = rings.Integer(val)

    def set_public_key_from_file(self, public_key_file):
        r"""
        set a new public key imported from a file
        """

        self.public_key = load(public_key_file)

    def set_private_key_from_file(self, private_key_file):
        r"""
        set a new private key imported from a file
        """

        self.private_key = load(private_key_file)

    def set_public_key(self, public):
        r"""
        set a new public key
        """

        self.public_key = public

    def set_private_key(self, private):
        r"""
        set a new private key
        """

        self.private_key = private

    def reset_key_pair(self):
        r"""
        reset key pair to the latest in class generated pair
        """

        self.public_key = self.generated_public_key
        self.private_key = self.generated_private_key
```

F.10 Sage sample: BLS signature example

This code is used in connection with Example 6.1

```
#This is test data for the BLS signature scheme using the Weil pairing
e=7
q=2^e
F1=FiniteField(q,'a')
k=4
t=cputime()
F2=FiniteField(q^k,'b')
phi=Hom(F1,F2)(F1.gen().minpoly().roots(F2)[0][0])
E1=EllipticCurve(F1,[0,0,1,1,1])
E2=EllipticCurve(F2,[0,0,1,1,1])
m=E1.cardinality()
n=[s for s,e in m.factor()].pop()
P1=int(m/n)*E1.random_point()
if P1==E1(0):
    print "P do not generate G_1, please reload"
P2=E2(phi(P1.xy()[0]),phi(P1.xy()[1]))
Q=145*E2.random_point()
```

F.11 Sage script: BLS CLI

```
#!/usr/bin/env sage -python
from sage.crypto.all import *
from sage.structure.sage_object import save, load
import os
import sys
from sage.all import *
header = "--------------------------------------------------------\n BLS short signature system\n--------------------------------------------------------\n\n"
# Ask the user what next - use while loop
program_lives = True
while program_lives:
    question0 = header+"please write path to BLSxx.sobj file or press 0 to exit\n\n:"
    command0 = raw_input(question0)
    if command0 == "0":
        sys.exit(1)
    else:
            BLS = load(command0)
            print "\nBLSxx.sobj file loaded!\n"
            program_lives = False




program_lives = True

while program_lives:
    question1 = "please select an option (0-7) followed by enter:\n\n 0) exit. \n 1) generate key pair \n 2) sign message \n 3) validate signature \n 4) export key pair \n 5) set public key \n 6) set private key \n 7) reset key pair \n\n:"
    question2 = "please enter path to message file: \n\n:"
    question3 = "please enter path to signature file: \n\n:"
    question4 = "please enter path to private key file: \n\n:"
    question5 = "please enter path to public key file: \n\n:"
    command1 = raw_input(question1)
    if command1 == "0":
        program_lives = False
    if command1 == "1":
        BLS.generate_key_pair()
        print "\n key pair was generated, remember to export keys\n"
    if command1 == "2":
        command2 = raw_input(question2)
        command3 = raw_input(question3)
        BLS.sign_file(command2,command3)
    if command1 == "3":
        command2 = raw_input(question2)
```

```
        command3 = raw_input(question3)
        r = BLS.validate_file(command2,command3)
        if r == True:
            print "\n signature is valid\n"
        if r == False:
            print "\n signature is invalid\n"
    if command1 == "4":
        command4 = raw_input(question4)
        command5 = raw_input(question5)
        BLS.export_key_pair_to_files(command4,command5)
        print "key pair stored to key files"
    if command1 == "5":
        command5 = raw_input(question5)
        BLS.set_public_key(command5)
        print "\n public key loaded\n"
    if command1 == "6":
        command4 = raw_input(question4)
        BLS.set_private_key_from_file(command4)
        print "\n private key loaded\n"
    if command1 == "7":
        BLS.reset_key_pair()
        print "\n Key pair was reset to last generated pair\n"

#TODO: Do some checks on inputs
sys.exit(1)
```

F.12 Sage interact: Weil Optimisations

```
# Latex representations of algoprithm 1-5 in article
# "Refinements of Miller's algorithm for computing the Weil/Tate pairing" by Blake et al.

def f1_print(nn):
    """
    returns string of LaTeX code
    Miller function calculated with algorithm 1
    """
    tl = [['f_{1}','1',1]]
    V = 1
    n=nn.bits()
    b=nn.nbits()
    i=b-2
    while i > -1:
        tl.append(['g_{'+str(V)+'P\,'+str(V)+'P}','g_{'+str(2*V)+'P}',1])
        V = 2*V
        s = len(tl)
        for k in range(0,s-1):
            tl[k][2] = 2*(tl[k][2])
        if n[i] == 1:
            tl.append(['g_{'+str(V)+'P\,P}','g_{'+str(V+1)+'P}',1])
            V = V+1
            tl[0][2] += 1
        i=i-1
    #t_tex = tl[0][0]+'^{'+str(tl[0][2])+'}'
    #s = len(tl)
    #for j in range(1,s):
    #    t_tex += '\\frac{'+tl[j][0]+'^{'+str(tl[j][2])+'}}'+'{'+tl[j][1]+'^{'+str(tl[j][2])+'}}'
    #return '$'+t_tex+'$'
    t_tex = tl[0][0]+'^{'+str(tl[0][2])+'}'
    t_tex += '\\frac{'+tl[1][0]+'^{'+str(tl[1][2])+'}}'+'{'+tl[1][1]+'^{'+str(tl[1][2])+'}}'
    s = len(tl)
    for j in range(2,s):
        if tl[j][1] == '1':
            if tl[j][2]>1:
                t_tex += tl[j][0]+'^{'+str(tl[j][2])+'}'
            else:
                t_tex += tl[j][0]
        else:
            if tl[j][2]>1:
                t_tex += '\\frac{'+tl[j][0]+'^{'+str(tl[j][2])+'}}'+'{'+tl[j][1]+'^{'+str(tl[j][2])+'}}'
            else:
                t_tex += '\\frac{'+tl[j][0]+'}'+'{'+tl[j][1]+'}'
```

```
    return '$'+t_tex+'$'
def f2_print(nn):
    """
    returns string of LaTeX code
    Miller function calculated with algorithm 2
    """
    tl = [['f_{1}','1',0],['g_{P\,P}','g_{2P}',0]]
    V = 1
    n=nn.digits(base=3)
    b=nn.ndigits(base=3)
    if n[b-1] == 1:
        tl[0][2]=1
        V = 1
    if n[b-1] == 2:
        tl[0][2]=2
        tl[1][2]=1
        V = 2
    i=b-2
    while i > -1:
        tl.append(['g_{'+str(V)+'P\,'+str(V)+'P}','g_{'+str(3*V)+'P}',1])
        tl.append(['g_{'+str(2*V)+'P\,'+str(V)+'P}','1',1])
        V = 3*V
        s = len(tl)
        for k in range(0,s-2):
            tl[k][2] = 3*(tl[k][2])
        if n[i] == 1:
            tl.append(['g_{'+str(V)+'P\,P}','g_{'+str(V+1)+'P}',1])
            tl[0][2]=tl[0][2]+1
            V = V+1
        if n[i] == 2:
            tl.append(['g_{'+str(V)+'P\,2P}','g_{'+str(V+2)+'P}',1])
            tl[0][2] = tl[0][2]+2
            tl[1][2] = tl[1][2]+1
            V = V+2
        i=i-1
    t_tex = tl[0][0]+'^{'+str(tl[0][2])+'}'
    t_tex += '\\frac{'+tl[1][0]+'^{'+str(tl[1][2])+'}}'+'{'+tl[1][1]+'^{'+str(tl[1][2])+'}}'
    s = len(tl)
    for j in range(2,s):
        if tl[j][1] == '1':
            if tl[j][2]>1:
                t_tex += tl[j][0]+'^{'+str(tl[j][2])+'}'
            else:
                t_tex += tl[j][0]
        else:
            if tl[j][2]>1:
                t_tex += '\\frac{'+tl[j][0]+'^{'+str(tl[j][2])+'}}'+'{'+tl[j][1]+'^{'+str(tl[j][2])+'}}'
            else:
```

```
                t_tex += '\\frac{'+tl[j][0]+'}'+'{'+tl[j][1]+'}'
    return '$'+t_tex+'$'
def f3_print(nn):
    """
    returns string of LaTeX code
    Miller function calculated with algorithm 3
    """

    tl = [['f_{1}',1,'1',0,'1',0,'1',0,'1',0,'1',0]] #first three strings are the nominator last three the denominator
    V = 1
    n = nn.digits(base=4)
    b = nn.ndigits(base=4)
    if n[b-1] == 2:
        tl[0][1] = 2*tl[0][1]
        tl.append(['g_{P\,P}',1,'1',0,'1',0,'g_{2P}',1,'1',0,'1',0])
        V = 2
    if n[b-1] == 3:
        tl[0][1] = 3*tl[0][1]
        tl.append(['g_{P\,P}',2,'1',0,'1',0,'g_{P}',1,'g_{2P\,P}',1,'1',0])
        V = 3
    i = b-2
    while i > -1:
        if n[i] == 0:
            s = len(tl)
            for k in range(0,s):
                for j in range(0,12):
                    if mod(j,2)==1:
                        tl[k][j] = 4*(tl[k][j])
            tl.append(['g_{'+str(V)+'P\,'+str(V)+'P}',2,'1',0,'1',0,'g_{'+str(2*V)+'P}',1,'g_{2P\,P}',1,'1',0])
            V = 4*V
        elif n[i] == 1:
            s = len(tl)
            for k in range(0,s):
                for j in range(0,12):
                    if mod(j,2)==1:
                        tl[k][j] = 4*(tl[k][j])
            tl.append(['g_{'+str(V)+'P\,'+str(V)+'P}',2,'g_{'+str(4*V)+'P\,P}',1,'1',0,'g_{'+str(2*V)+'P\,'+str(2*V)+'P}',1,'g_{'+str(4*V+1)+'P}',1,'1',0])
            tl[0][1] += 1
            V = 4*V+1
        elif n[i] == 2:
            s = len(tl)
            for k in range(0,s):
                for j in range(0,12):
                    if mod(j,2)==1:
                        tl[k][j] = 4*(tl[k][j])
```

```
            tl.append(['g_{'+str(V)+'P\,'+str(V)+'P}',2,'g_{'+str(2*V)+'P\,P}',2,'1',0,'g_{'+str(2*V+1)+'P\,'+str(2*V+1)+'P}',1,'g_{'+str(2*V)+'P}',1,'1',0])
            tl[0][1] += 2
            V = 4*V+2
        elif n[i] == 3:
            s = len(tl)
            for k in range(0,s): # raise the power of all previous factors
                for j in range(0,12):
                    if mod(j,2)==1:
                        tl[k][j] = 4*(tl[k][j])
            tl.append(['g_{'+str(V)+'P\,'+str(V)+'P}',2,'g_{'+str(2*V)+'P\,P}',2,'g_{'+str(4*V+2)+'P\,P}',1,'g_{'+str(2*V)+'P}',2,'g_{'+str(2*V+1)+'P\,'+str(2*V+1)+'P}',1,'g_{'+str(4*V+3)+'P}',1])
            tl[0][1] += 3
            V = 4*V+3
        i=i-1
    t_tex = ''
    s = len(tl)
    for j in range(0,s):
        for i in [0,2,4]:
            # Here it should print several factors in nominator or denominator
            if tl[j][i+1]>0:
                if tl[j][i+7]>0:
                    if tl[j][i+1]>1 and tl[j][i+7]>1:
                        t_tex += '\\frac{'+tl[j][i]+'^{'+str(tl[j][i+1])+'}}'+'{'+tl[j][i+6]+'^{'+str(tl[j][i+7])+'}}'
                    elif tl[j][i+1]>1:
                        t_tex += '\\frac{'+tl[j][i]+'^{'+str(tl[j][i+1])+'}}'+'{'+tl[j][i+6]+'}'
                    elif tl[j][i+7]>1:
                        t_tex += '\\frac{'+tl[j][i]+'}'+'{'+tl[j][i+6]+'^{'+str(tl[j][i+7])+'}}'
                    else:
                        t_tex += '\\frac{'+tl[j][i]+'}'+'{'+tl[j][i+6]+'}'
                else:
                    if tl[j][i+1]>1:
                        t_tex += '{'+tl[j][i]+'^{'+str(tl[j][i+1])+'}}'
                    else:
                        t_tex += '{'+tl[j][i]+'}'
    return '$'+t_tex+'$'

def f4_print(nn):
    """
    returns string of LaTeX code
    Miller function calculated with algorithm 4
    """
```

```
    tl = [['f_{1}',1,'1',0,'1',0,'1',0,'1',0,'1',0]] #first three strings are the nominator last three the denominator
    V = 1
    n = nn.bits()
    b = nn.nbits()
    if n[b-2] == 0:
        tl[0][1] = 2*tl[0][1]
        tl.append(['g_{P\,P}',1,'1',0,'1',0,'1',0,'1',0,'1',0])
        V = 2
    else:
        tl[0][1] = 3*tl[0][1]
        tl.append(['g_{P\,P}',1,'g_{2P\,P}',1,'1',0,'g_{2P}',1,'1',0,'1',0])
        V = 3
    i = b-3
    while i > -1:
        if n[i] == 0:
            s = len(tl)
            for k in range(0,s):
                for j in range(0,12):
                    if mod(j,2)==1:
                        tl[k][j] = 2*(tl[k][j])
            tl.append(['g_{'+str(2*V)+'P}',1,'1',0,'1',0,'g_{'+str(V)+'P\,'+str(V)+'P}',1,'1',0,'1',0])
            V = 2*V
        else:
            s = len(tl)
            for k in range(0,s):
                for j in range(0,12):
                    if mod(j,2)==1:
                        tl[k][j] = 2*(tl[k][j])
            tl.append(['g_{'+str(2*V)+'P\,P}',1,'1',0,'1',0,'g_{'+str(V)+'P\,'+str(V)+'P}',1,'1',0,'1',0])
            tl[0][1] += 1
            V = 2*V+1
        i=i-1
    t_tex = ''
    s = len(tl)
    for j in range(0,s):
        for i in [0,2,4]:
            # Here it should print several factors in nominator or denominator
            if tl[j][i+1]>0:
                if tl[j][i+7]>0:
                    if tl[j][i+1]>1 and tl[j][i+7]>1:
                        t_tex += '\\frac{'+tl[j][i]+'^{'+str(tl[j][i+1])+'}}'+'{'+tl[j][i+6]+'^{'+str(tl[j][i+7])+'}}'
                    elif tl[j][i+1]>1:
                        t_tex += '\\frac{'+tl[j][i]+'^{'+str(tl[j][i+1])+'}}'+'{'+tl[j][i+6]+'}'
                    elif tl[j][i+7]>1:
```

```
                        t_tex += '\\frac{'+tl[j][i]+'}'+'{'+tl[j][i+6]+'^{'+str(tl[j][i+7])+'}}'
                    else:
                        t_tex += '\\frac{'+tl[j][i]+'}'+'{'+tl[j][i+6]+'}'
                else:
                    if tl[j][i+1]>1:
                        t_tex += '{'+tl[j][i]+'^{'+str(tl[j][i+1])+'}}'
                    else:
                        t_tex += '{'+tl[j][i]+'}'
    return '$'+t_tex+'$'

def f5_print(nn):
    """
    returns string of LaTeX code
    Miller function calculated with algorithm 5
    """

    tl = [['f_{1}',1,'1',0,'1',0,'1',0,'1',0,'1',0]] #first three strings are the nominator last three the denominator
    V = 1
    tl.append(['g_{P\,P}',0,'1',0,'1',0,'g_{2P}',0,'1',0,'1',0])
    n = nn.digits(base=3)
    b = nn.ndigits(base=3)
    if n[b-1] == 1:
        tl[0][1] = 1
        V = 1
    if n[b-1] == 2:
        tl[0][1] = 2
        tl[1][1] += 1
        tl[1][7] += 1
        V = 2
    i = b-2
    while i > -1:
        s = len(tl)
        for k in range(0,s):
            for j in range(0,12):
                if mod(j,2)==1:
                    tl[k][j] = 3*(tl[k][j])
        tl.append(['g_{'+str(V)+'P\,'+str(V)+'P}',1,'g_{'+str(V)+'P}',1,'1',0,'g_{'+str(2*V)+'P\,'+str(V)+'P}',1,'1',0,'1',0])
        V = 3*V
        if n[i]==1:
            tl.append(['g_{'+str(V)+'P\,P}',1,'1',0,'1',0,'g_{'+str(V+1)+'P}',1,'1',0,'1',0])
            tl[0][1] += 1
            V = V+1
        if n[i]==2:
            tl.append(['g_{'+str(V)+'P\,2P}',1,'1',0,'1',0,'g_{'+str(V+2)+'P}',1,'1',0,'1',0])
```

```
            tl[0][1] += 2
            tl[1][1] += 1
            tl[1][7] += 1
            V = V+2
        i=i-1
    t_tex = ''
    s = len(tl)
    for j in range(0,s):
        for i in [0,2,4]:
            # Here it should print several factors in nominator or denominator
            if tl[j][i+1]>0:
                if tl[j][i+7]>0:
                    if tl[j][i+1]>1 and tl[j][i+7]>1:
                        t_tex += '\\frac{'+tl[j][i]+'^{'+str(tl[j][i+1])+'}}'+'{'+tl[j][i+6]+'^{'+str(tl[j][i+7])+'}}'
                    elif tl[j][i+1]>1:
                        t_tex += '\\frac{'+tl[j][i]+'^{'+str(tl[j][i+1])+'}}'+'{'+tl[j][i+6]+'}'
                    elif tl[j][i+7]>1:
                        t_tex += '\\frac{'+tl[j][i]+'}'+'{'+tl[j][i+6]+'^{'+str(tl[j][i+7])+'}}'
                    else:
                        t_tex += '\\frac{'+tl[j][i]+'}'+'{'+tl[j][i+6]+'}'
                else:
                    if tl[j][i+1]>1:
                        t_tex += '{'+tl[j][i]+'^{'+str(tl[j][i+1])+'}}'
                    else:
                        t_tex += '{'+tl[j][i]+'}'
    return '$'+t_tex+'$'


@interact
def select_n(n=257):
    n = Integer(n)
    l2 = baseconvert(n,2)
    l3 = baseconvert(n,3)
    l4 = baseconvert(n,4)
    if n.mod(3) != 0 and n.mod(2) != 0:
        t1 = f1_print(n)
        t2 = f2_print(n)
        t3 = f3_print(n)
        t4 = f4_print(n)
        t5 = f5_print(n)
        #base 2 list to tex
        l2_tex = "$n=["
        for i in range(0,len(l2)-1):
            l2_tex = l2_tex+"%s\,"%l2[i]
        l2_tex = l2_tex + "%s]_2$"%l2[len(l2)-1]
        #base 3 list to tex
        l3_tex = "$=["
```

```
        for i in range(0,len(l3)-1):
            l3_tex = l3_tex+"%s\,"%l3[i]
        l3_tex = l3_tex + "%s]_3$"%l3[len(l3)-1]
        #base 4 list to tex
        l4_tex = "$=["
        for i in range(0,len(l4)-1):
            l4_tex = l4_tex+"%s\,"%l4[i]
        l4_tex = l4_tex + "%s]_4$"%l4[len(l4)-1]
        html('Refinements of the Miller algorithm w.r.t. representation of n:<br>')
        html('Base representations: %s %s %s<br>'%(l2_tex,l3_tex, l4_tex))
        #html('base 3 representation: %s<br>'%l3_tex)
        #html('base 4 representation: %s<br>'%l4_tex)
        html('<table border=1>')
        html('<tr bgcolor="#edcc9c"><td align=center> Algorithm</td><td align=center>f function expression</td>')
        html('<tr><td align=right> 1: simple base 2 </td><td align=left> '+t1+' </td>')
        html('<tr><td align=right> 2: simple base 3 </td><td align=left> '+t2+' </td>')
        html('<tr><td align=right> 3: sparse base 2 </td><td align=left> '+t3+' </td>')
        html('<tr><td align=right> 4: refined base 2 </td><td align=left> '+t4+' </td>')
        html('<tr><td align=right> 5: refined base 3 </td><td align=left> '+t5+' </td>')
        html('</table>')
    else:
        html('Please give n not divisible by 2 or 3')
```

www.mat.dtu.dk

Department of Mathematics
Technical University of Denmark
Matematiktorvet
Building 303S
DK-2800 Kgs. Lyngby
Denmark
Tel: (+45) 45 25 30 31
Fax: (+45) 45 88 13 99
Email: instadm@mat.dtu.dk