

# Developing an automatic generation tool for cryptographic pairing functions

Luis Julian Dominguez Perez

B. Sc. M. A.

A Dissertation submitted in fulfilment of the  
requirements for the award of  
Ph.D.

to the



Dublin City University

Faculty of Engineering and Computing  
School of Computing

Supervisor: Michael Scott

January, 2011

# Declaration

I hereby certify that this material, which I now submit for assessment on the programme of study leading to the award of Ph.D. is entirely my own work, that I have exercised reasonable care to ensure that the work is original, and does not to the best of my knowledge breach any law of copyright, and has not been taken from the work of others save and to the extent that such work has been cited and acknowledged within the text of my work.

---

Signed: Luis Julian Dominguez Perez.

ID: 57104328.

Date: January 2011.

# Abstract

Pairing-Based Cryptography is receiving steadily more attention from industry, mainly because of the increasing interest in Identity-Based protocols. Although there are plenty of applications, efficiently implementing the pairing functions is often difficult as it requires more knowledge than previous cryptographic primitives. The author presents a tool for automatically generating optimized code for the pairing functions which can be used in the construction of such cryptographic protocols.

In the following pages I present my work done on the construction of pairing function code, its optimizations and how their construction can be automated to ease the work of the protocol implementer.

Based on the user requirements and the security level, the created cryptographic compiler chooses and constructs the appropriate elliptic curve. It identifies the supported pairing function: the Tate, ate, R-ate or pairing lattice/optimal pairing, and its optimized parameters. Using artificial intelligence algorithms, it generates optimized code for the final exponentiation and for hashing a point to the required group using the parametrisation of the chosen family of curves.

Support for several multi-precision libraries has been incorporated: Magma, MIRACL and RELIC are already included, but more are possible.

# Acknowledgements

This research was sponsored by the Consejo Nacional de Ciencia y Tecnología, Conacyt. Also thanks to the Claude Shannon Institute for its support and extra training.

The extensive and key support from my supervisor has been fundamental to the realization of this research. Thank you so much Mike. Also, I would like to acknowledge all of the anonymous referees on my submitted and shared publications. Their feedback was also a key factor in the consummation of this project. From CSI, thanks to Gary, Marcus and all of the professors, post-docs and students for giving lectures and talks.

A special thanks to Paulo Barreto for reading an early draft of this work.

I would like to thanks to my friends from the lab. In particular to Naomi, for her outstanding support in proof-reading and joint-work. To Ezekiel for his support and the joint-work we had together. To Prof.. Kim for his extensive patience in the lab and his interesting talks. To Chen for always being there. To Rob for attending several times the same speech, and always giving feedback. To Denis for joining the cryptolab ship, and to Manuel. To Neil for proof-reading my thesis. And my friends from the bigger group: Brian, John, Richard, Danny, Geoff, and Jens, for that work and funny talks. Also, thanks for all the visitors.

An special thanks to Diego F. Aranha and Juan Martinez Castillo, for his help on timing the code.

*A mis papás por apoyarme siempre en mis estudios, a mis hermanas y familiares, tanto tiempo perdido en familia. Y a mi abuelita (+) también*

I have had so many house-mates, you all make me happy, specially the 31ers. Hopefully we meet again.

Atentamente,

Luis Julian.

# TABLE OF CONTENTS

<b>List of Tables</b>	<b>vii</b>
<b>List of Figures</b>	<b>viii</b>
<b>List of Code</b>	<b>1</b>
<b>1 Introduction</b>	<b>1</b>
<b>2 Background</b>	<b>5</b>
2.1 Elliptic Curves . . . . .	5
2.1.1 Arithmetic of elliptic curves . . . . .	7
2.1.2 Tower extensions of finite fields . . . . .	9
2.1.3 Twist of a curve over an extension field . . . . .	11
2.1.4 ECDLP . . . . .	12
2.1.4.1 Security level . . . . .	12
2.1.5 Elliptic Curve Isomorphisms . . . . .	14
2.2 Divisors . . . . .	14
2.3 Cyclotomic Polynomial . . . . .	15
2.4 Pairings . . . . .	16
2.4.1 The Tate Pairing . . . . .	17
2.4.1.1 The Miller Loop for the Tate Pairing . . . . .	18
2.4.2 The ate Pairing . . . . .	19
2.4.2.1 The Miller Loop for the ate Pairing . . . . .	20
2.4.3 The R-ate Pairing . . . . .	22
2.4.3.1 Shorter R-ate pairing . . . . .	25
2.4.3.2 Choosing the R-ate pairing parameters . . . . .	27
2.4.4 Pairing Lattices . . . . .	28

2.4.5	Optimal Pairing . . . . .	31
2.4.6	Comparison of the Miller-loop length . . . . .	31
2.5	Pairing-Friendly Elliptic Curves . . . . .	32
2.5.1	MNT Curves . . . . .	33
2.5.2	Freeman Curves . . . . .	34
2.5.3	Barreto-Naehrig (BN) Curves . . . . .	34
2.5.4	Kachisa-Schaefer-Scott Curves . . . . .	34
2.5.4.1	KSS curves with $k = 8$ . . . . .	35
2.5.4.2	KSS curves with $k = 18$ . . . . .	35
2.5.4.3	KSS curves with $k = 36$ . . . . .	36
2.6	Matrices . . . . .	36
2.6.1	Greatest Common Divisor . . . . .	36
2.6.2	Echelon Form . . . . .	36
2.7	Programming Languages . . . . .	37
2.7.1	Magma . . . . .	37
2.7.2	MIRACL . . . . .	38
2.7.3	RELIC . . . . .	38
2.8	Metaprogramming . . . . .	39
2.8.1	Automatic Code Generation . . . . .	40
2.8.1.1	L- and R-value . . . . .	40
2.8.1.2	Suffix Notation . . . . .	41
2.8.1.3	Operator Overloading . . . . .	42
2.8.2	Attribute-Oriented Programming . . . . .	43
2.8.3	Reflective Programming . . . . .	44
2.8.4	Template-based Metaprogramming . . . . .	45
2.9	Exponentiation in $G_{2,T}$ . . . . .	46
2.9.1	Weak Popov Representation . . . . .	49
2.9.1.1	Popov Form . . . . .	50

2.9.1.2	Quasi-Echelon Form . . . . .	51
2.9.1.3	Weak Popov Form . . . . .	52
2.9.2	The Galbraith-Scott Method using the Weak Popov Form . . . . .	55
<b>3</b>	<b>Addition Chains</b>	<b>57</b>
3.1	Introduction . . . . .	58
3.2	Bos and Coster Method . . . . .	59
3.3	Binary Method . . . . .	60
3.3.1	Construction of the Binary Method . . . . .	61
3.3.2	Example using the Binary Method . . . . .	62
3.4	Artificial Intelligence method . . . . .	63
3.4.1	Artificial Immune System . . . . .	64
3.4.1.1	Types of Cells on the Immune System . . . . .	64
3.4.1.2	Immune Engineering . . . . .	65
3.4.1.3	Immune System Metaphors . . . . .	65
3.5	Addition Chain Construction . . . . .	66
3.5.1	Initial Sequence Generation . . . . .	67
3.5.2	Auto-Immune Disease . . . . .	68
3.5.3	Hypermutation . . . . .	68
3.5.4	Core Function . . . . .	69
3.5.5	Results . . . . .	72
3.6	Comparison of the Methods . . . . .	72
3.7	Final Thoughts on Addition Chains . . . . .	74
<b>4</b>	<b>The Final Exponentiation</b>	<b>76</b>
4.1	The Du, Hong and Pei Method . . . . .	77
4.2	Devegili, Scott and Dahab Method . . . . .	78
4.3	A new method . . . . .	79
4.3.1	More Examples . . . . .	86

## Table of contents

---

4.3.1.1	The MNT curves . . . . .	86
4.3.1.2	The BN Curves . . . . .	87
4.3.1.3	Freeman Curves . . . . .	89
4.3.1.4	The KSS: $k = 8$ Family of Curves . . . . .	90
4.4	A Note on the Final Exponentiation . . . . .	90
<b>5</b>	<b>Fast Hashing to <math>G_2</math></b>	<b>92</b>
5.1	Introduction . . . . .	93
5.2	Point Counting . . . . .	93
5.3	A new method . . . . .	96
5.4	Final Thoughts . . . . .	101
<b>6</b>	<b>Code Generator</b>	<b>103</b>
6.1	Instruction Construction . . . . .	104
6.2	Attribute-based construction . . . . .	108
6.3	Reflective program construction . . . . .	109
6.4	Template-based construction . . . . .	111
6.5	Adding support to a library . . . . .	112
6.5.1	Hashing to $G_2[r]$ . . . . .	113
6.5.2	Final exponentiation . . . . .	114
6.5.3	The pairing function . . . . .	115
6.5.4	Others . . . . .	116
6.5.5	Linking the library to the project . . . . .	116
6.6	Parameters of the program . . . . .	116
6.6.1	Other user-specified options . . . . .	117
6.7	Feedback symbols in the program . . . . .	118
6.8	Code sequence . . . . .	119
6.8.1	Default variables and functions load . . . . .	119
6.8.2	Sample generated code . . . . .	126



## Table of contents

---

6.8.2.1	KSS $k = 8$ . . . . .	127
6.8.2.2	KSS $k = 18$ . . . . .	130
6.8.3	Timings . . . . .	141
6.9	Final Thoughts . . . . .	145
<b>7</b>	<b>Conclusions</b>	<b>146</b>
<b>A</b>	<b>Example parameters of the curve</b>	<b>150</b>
A.1	Hamming Weight . . . . .	150
<b>B</b>	<b>Certicom Challenge</b>	<b>151</b>
B.1	Methodology . . . . .	153
B.2	Technical Requirements . . . . .	154
B.3	Running the Attack from Ireland . . . . .	154
<b>C</b>	<b>Online Tutorial</b>	<b>157</b>
C.1	Introduction . . . . .	158
C.2	Magma Online Calculator . . . . .	159
C.3	AJAX . . . . .	160
C.4	YUI . . . . .	162
C.4.1	YUI Structure . . . . .	162
C.5	Our Construction . . . . .	164
C.5.1	The XMLHttpRequest . . . . .	165
C.5.2	Reading the Output . . . . .	167
C.5.3	Displaying the Results . . . . .	167
C.6	Internal Comparisons . . . . .	169
C.7	Final Thoughts about the Online Tutorial . . . . .	171
<b>D</b>	<b>Code Generator function list</b>	<b>172</b>
	<b>Bibliography</b>	<b>180</b>

<b>Subject Index</b>	<b>196</b>
----------------------	------------

# LIST OF TABLES

2.1	Tower extension construction . . . . .	10
2.2	Irreducible polynomials for the tower extension construction . . . . .	11
2.3	NIST and ECRYPT security level recommendations . . . . .	13
2.4	KSS: $k = 18$ Curves A,B Parameters . . . . .	27
2.5	Comparison of Miller-loop length for a KSS: $k = 18$ curve, with $\text{Log}_2 x \approx 63$	32
3.1	Input Matrix for the Binary Method. . . . .	62
4.1	Constructing the vector chain . . . . .	83
4.2	Partial final exponentiation code from the Olivos and Scott et al. method. KSS Curve: $k = 18$ . . . . .	83
4.3	Partial final exponentiation code for the same curve, with a reduced number of $t_i$ elements. . . . .	83
6.1	Filename Notation . . . . .	109
6.2	Definitions for $G_2$ . . . . .	113
6.3	Definitions for FE . . . . .	114
6.4	Definitions for pairing function . . . . .	115
6.5	Timing the generated code . . . . .	141
6.6	Comparison of the generated code, BN curves. CPU cycles in millions. . .	144
A.1	Useful Low Hamming-Weight Values . . . . .	150
D.1	List of functions with Input/Output description . . . . .	173

# LIST OF FIGURES

2.1	Addition Operation . . . . .	8
2.2	Doubling Operation . . . . .	8
2.3	Point-at-Infinity . . . . .	9
2.4	$\psi$ map for KSS: $k = 18$ curves . . . . .	21
2.5	$\psi$ map for BN curves . . . . .	21
6.1	Feedback symbols . . . . .	118
C.1	Screenshot of the welcome screen. . . . .	159
C.2	Screenshot of the paginator . . . . .	164
C.3	Screenshot of the Panel and Dialog . . . . .	164
C.4	Screenshot of the displayed stored values . . . . .	168
C.5	Screenshot of the Miller loop length computation for a random KSS: $k = 8$ curve. . . . .	169
C.6	Screenshot of the R-ate pairing comparison . . . . .	170

# LIST OF CODE

$\psi$  via lifting, 44

Binary Method, 61

Display Computation, 167

Generic Assignment, 105

GenericOperation, 106

Parsing the Magma Calculator, 167

Preparing the XMLHttpRequest, 165

Testing the Fast Cofactor Function, 110

The XMLHttpRequest and Response,  
166

# 1 INTRODUCTION

*"If it were ammunition, you would not be here." – Gral. Pedro Maria Anaya after being asked to surrender the weapons and ammunition at the defense of the 'Convento de Churubusco', August 20th, 1847.*

INTEREST in pairing-based cryptography has been growing since the arrival of the new millennium, thanks to the development of many constructive protocols; for example, those given in [Jou00] and [BLS01]. The usefulness of these protocols has caught the attention of industry.

Traditional cryptographic protocols, such as RSA, are well established and seen as “secure enough” for the immediate future, but have limited functionality. Pairing-based cryptography is slowly being seen as a viable option.

The main disadvantage of implementing pairing-based protocols instead of these well-established solutions is the deeper mathematical background required to produce an efficient implementation. Every year, new improvements on pairing computation methods appear. A pairing-based protocol designer may prefer to focus on the proof and formalization of the protocol itself rather than on the physical construction of the primitives upon which it relies. Given the many improvements, it is easy to lose track of the most “up-to-date” optimizations and use a less efficient implementation.

This research in pairing-based cryptography started in a joint work of the author with Kachisa and Scott. This was an exploratory research, aiming to “get to know pairings” and to find possible areas for development.

During that exploration, we were working with the recently discovered  $KSS:k = 18$  family of curves looking for new properties of the curves.

On the completion of a bilinear and non-degenerate pairing implementation, we noticed how slow it was; however, we were able to detect the cause, it was the infamous final ex-

ponentiation. In an attempt to speed-up this step, we tried the [DSD07] implementation method for the BN curves. However, the implementation was not suitable for our purposes, as we failed to reproduce their results using our target curve. We commenced a joint work with Scott, Benger, Charlemagne and Kachisa [SBC<sup>+</sup>09b] to speed-up the Final Exponentiation operation for general pairing-friendly elliptic curves. It was our idea to use a base- $p$  representation of the exponent as much as possible to find a pattern in the final exponentiation of the pairing. We tried simple exponentiation by factorization, as was suggested by the other authors, but we preferred a method making use of addition-chain theory. We developed a fast final exponentiation implementation that can be constructed for any family of pairing-friendly elliptic curves. This work resulted in a publication which was presented at the conference *Pairing 2009*.

Switching focus to some operations necessary in pairing-based protocols, we noted yet another expensive operation: the scalar-point multiplication by a large cofactor.

We applied the Galbraith and Scott [GS08] ideas for exponentiation in  $G_2$  groups to our problem for the family of curves we were originally working on. We developed a formal and faster method to hash a random point into a group  $G_2$  of order  $r$ . We were able to extend this method to other elliptic curves. This research was published in [SBC<sup>+</sup>09a]. Again, making use of the base- $p$  representation of the curve parameters, and the Frobenius exponentiation, we added the map from Galbraith and Scott [Ibidem.] and obtained some promising results. We presented this work at *Pairing 2009*. This research is detailed in Chapter 5.

Once these two issues had been solved, we decided to go back to explore implementation issues. This paper contained nothing new as we had already published the results separately. We simply compiled the results in one article and released it in the IACR eprint archive [DKS], where it has received much attention and has been referenced. The examples in the paper guide the reader through a tutorial which uses Magma software. We realized that not everybody has access to this software so, in order to reach a larger audience, I developed a webpage, where a user without a Magma license can follow the examples. The

details of this online tutorial are explained in Appendix C.

What seemed an unnecessarily complex algorithm, developed into a very useful method for finding addition sequences; we used the computer paradigm of Artificial Immune Systems to find the necessary sequences. In the work already described, we needed addition chains for the exponentiation in finite fields and for scalar-point multiplication in a group of points. As the embedding degree  $k$ , and the degree and complexity of the polynomials  $p$  and  $r$  defining the parameters of the curve grows, finding an addition sequence becomes unmanageable by pen-and-paper. I adapted the Cruz-Cortez et al. method [CCRHC08] to our addition sequences. A peer-reviewed paper which contains this modified method and some work from Scott was presented at the SPEED-CC 2009 Workshop [DS09].

As part of the publication [DS09], I started the design of a code generator for pairing functions. The afore mentioned publications and the new popularity of pairings functions requiring a construction method, needed be automatised. The aim of the code generator for cryptographic pairing functions is the following: to decide (or suggest) which family of pairing-friendly elliptic curves to choose (for example curves we refer to Freeman et al. [FST10, Table 1.1 and 8.3]); to find a low Hamming-weight  $x$ -parameter for the definition of the system parameters; to generate the elliptic curve with a subgroup size corresponding to the desired security level; to choose the pairing function that best suits the family of curves to which the chosen curve belongs, and its representation; to include supportive functions; and, optionally, to generate a sample “playground” for testing the code.

In order for a code generator to be flexible, support for several multi-precision libraries should be included. Some characteristics of the code rely on the programming language itself; others rely on the library. Some operations use an in-fixed operator, depending on the library. For some, this may only be possible with the use of a map and with explicit intermediate storage, where for others, the compiler can handle it.

Some recent speed records on the pairing computation are not based on a particular multi-precision library, but on a hand-crafted set of finite field arithmetic functions, such as: [NNS10, BGM<sup>+</sup>10, LMN10]. When referring to a multi-precision library in this research,



we would like to include in the definition any set of specialised finite field arithmetic. We have a description of the code generator in Chapter 6.

Our implementation paper is still alive and is highlighting interesting research areas. After the improvements given by the mentioned optimizations, in practical implementation of a pairing-based protocol [Sco02], the slowest part of the pairing function is now the exponentiation in the  $G_2$  and  $G_T$  groups.

As a side project, I have a contribution in an attempt to break a security system detailed in Appendix B, with the aim of developing an understanding of the real security of the detailed cryptosystems.

The rest of this thesis is as follows: Chapter 2 gives a background on pairings, and metaprogramming for the techniques used in the code generator, Chapter 3 details the automatic construction of addition chains and the required computer science background. The conclusions and final remarks are presented in Chapter 7. Chapter 6 presents selected generated code; we stress that more sample code will be hosted on my personal webpage. Appendix A gives selected parameters for the pairing implementation.

## 2 BACKGROUND

*"I am proud to be humble." – Moises Castillo criticizing a spiritual leader.*

THE public release of the public key cryptosystem by Diffie and Hellman in 1976 [DH76] not only created modern cryptography, but also concentrated the Computational Number Theory efforts in this direction.

The first usable public key system was the RSA scheme introduced by Rivest, Shamir and Adleman in [RSA78]; it is based on the problem of factoring large integers.

Later, in 1985 Miller [Mil86], and independently Koblitz in [Kob87] pointed out a discrete logarithm problem (DLP) in the group of points of an elliptic curve defined over a finite field.

A Public-Key Cryptosystem relies on the infeasibility of finding a decryption process given an encryption key.

Elliptic Curve Cryptography (ECC), is a public-key cryptography system that uses the structure of the elliptic curves defined over finite fields [Sti95].

The main advantage of ECC is that it requires significantly smaller keys to encrypt/decrypt data, compared to other public-key cryptosystems, for similar security levels. It also scales better at higher security levels. The higher the security level, the better ECC compares with other public-key systems. This is particularly true when compared to cryptosystems based on RSA, which is the most widely used.

### 2.1 Elliptic Curves

Let  $p$  be a prime number, and  $\mathbb{F}_p$  a field of integers modulo  $p$ .

An elliptic curve  $E$  over the finite field  $\mathbb{F}_p$ , denoted as  $E(\mathbb{F}_p)$ , has its arithmetic in terms

of the underlying finite field and is defined by the following equation:

$$y^2 = x^3 + ax + b \quad (2.1)$$

where  $a, b$  are defined in  $\mathbb{F}_p$  and satisfy  $4a^3 + 27b^2 \not\equiv 0 \pmod{p}$ . The set of solutions  $(x, y)$  with  $x, y \in \mathbb{F}_p$  satisfies equation 2.1. Additionally, the point-at-infinity, denoted as  $\mathcal{O}$  or  $\infty$  is also on the curve.

Let  $P$  be a point in  $E(\mathbb{F}_p)$  with prime order  $r$ ; then a cyclic subgroup of  $E(\mathbb{F}_p)$  generated by  $P$  is denoted as

$$\langle P \rangle = \{\infty, P, 2P, 3P, \dots, (r-1)P\}$$

where  $r$  is the number of elements in the subgroup.

**Definition 1.** The discriminant  $\Delta(E)$  of an elliptic curve  $E$  is defined as [CF06]:

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

where  $a, b$  are the parameters of the short Weierstrass formula of  $E$ .

**Definition 2.** The absolute invariant, also known as the  $j$ -invariant of  $E$ , is defined as [CF06]:

$$j(E) = 12^3 \frac{-4a^3}{\Delta(E)}.$$

**Definition 3.** Let  $E$  be an elliptic curve defined over  $\mathbb{F}_p$ , then

$$p + 1 - 2\sqrt{p} \leq \#E(\mathbb{F}_p) \leq p + 1 + 2\sqrt{p}$$

is called the Hasse interval. The number of points on  $E(\mathbb{F}_p)$  is in the Hasse interval. The number  $t = p + 1 - \#E(\mathbb{F}_p)$  is called the trace of Frobenius of  $E$  over  $\mathbb{F}_p$  and satisfies  $|t| \leq 2\sqrt{p}$  [HMOV04].

**Definition 4.** The embedding degree of a curve is the smallest integer  $k$  where  $r \mid (p^k - 1)$  and sometimes it is also referred as the security multiplier of the curve.

**Definition 5.** An elliptic curve  $E$  defined over  $\mathbb{F}_p$  is supersingular if  $p$  divides  $t$ , the trace of the Frobenius, otherwise the curve is ordinary [HMOV04].

A supersingular curve with  $p > 3$  has  $\#E(\mathbb{F}_p) = p + 1$ .

**Definition 6.** Let  $\mathbb{F}_{p^k}$  be an extension field of  $\mathbb{F}_p$  of degree  $k$ .  $\mathbb{F}_{p^k}^*$  is a field composed by the non-zero elements of  $\mathbb{F}_{p^k}$  under the multiplicative law.

**Definition 7.** The Complex Multiplication method or simply CM method constructs an elliptic curve with a group of points with selected order  $r$ .

The CM method is also referred to as the *Atkin-Morain method* for curves over prime fields; and the *Lay-Zimmer* method for curves over binary fields. This method is efficient if the finite field order  $p$  and the elliptic curve order  $r = p + 1 - t$  are selected so that the complex multiplication field  $\mathbb{Q}(\sqrt{t^2 - 4p})$  has a small enough class number ([HMOV04], pp. 179).

### 2.1.1 Arithmetic of elliptic curves

An elliptic curve  $E$  can be represented in several coordinate systems for instance, in [CF06, §13.2], a few of these systems are presented. Some of these systems reuse intermediate values in the arithmetic to gain a speed-up. Lauter, Montgomery and Naehrig in [LMN10] made an analysis of the affine coordinate system against some other projective coordinate systems which were thought in the literature to be faster. Their analysis covered the same type of computations as this thesis. Their findings suggest that, due to the simplicity of the affine coordinate formulae, it may be preferred.

The following are the formulae for the group law for arithmetic on elliptic curves defined over  $\mathbb{F}_p$ , using affine coordinates [CF06, 13.2.1.a].

**Negation.** Let  $P = (x_1, y_1)$ , then  $-P = (x_1, -y_1)$ .

Figure 2.1: Addition Operation

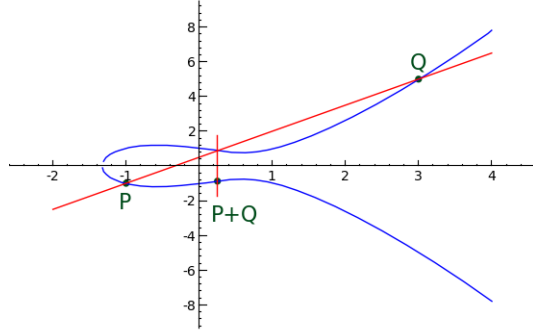
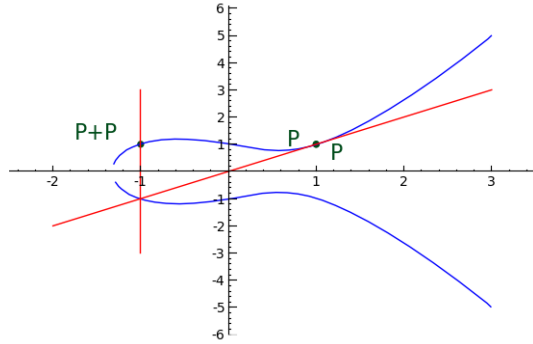


Figure 2.2: Doubling Operation



**Addition.** Let  $P = (x_1, y_1)$ , and  $Q = (x_2, y_2)$  such that  $P \neq \pm Q$ , then  $P + Q = (x_3, y_3)$ .

For an addition operation, set  $\lambda = \frac{y_2 - y_1}{x_2 - x_1}$ .

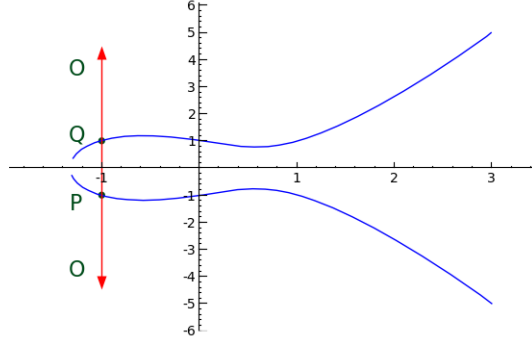
**Doubling.** Let  $[2]P = (x_3, y_3)$ . For a doubling operation, set  $\lambda = \frac{3x_1^2 + a}{2y_1}$

Then, we have  $x_3 = \lambda^2 - x_1 - x_2$ , and  $y_3 = \lambda(x_1 - x_3) - y_1$ .

In the case of operations with the point-at-infinity:  $P + \mathcal{O} = \mathcal{O} + P = P$ .

Suppose we defined the elliptic curve over  $\mathbb{C}$  instead of a finite field. Then the curve operations would be given geometrically as in Figures 2.1, 2.2 and 2.3, where  $\lambda$  is the slope of the line joining  $P$  and  $Q$ .

Figure 2.3: Point-at-Infinity



### 2.1.2 Tower extensions of finite fields

An element  $\alpha \in \mathbb{F}_{p^k}$  can be represented as a polynomial up to degree  $k - 1$  with coefficients in  $\mathbb{F}_p$  modulo an irreducible polynomial  $\in \mathbb{F}_p[X]$ . For efficiency purposes, this irreducible polynomial should be simple.

One method to construct the  $\mathbb{F}_{p^k}$  is using towers of extensions. Baktir and Sunar in [BS04] introduced the concept of tower field representation, which facilitates the finite field operations, in particular the inversion. They defined a *tower field* as a field obtained by extending its ground field with several irreducible polynomials.

Benger and Scott in [BS09] presented a method to construct a tower, using towering-friendly fields.

**Definition 8.** A towering-friendly field is a field of the form  $\mathbb{F}_{q^m}$ , where  $q$  is a prime power, where  $q - 1$  is divisible by all of the prime divisors of  $m$ . [BS09]

These fields are constructed using a tower of sub-extensions using binomials as irreducible polynomials. Each sub-extension is a layer constructed by adjoining the roots of the previous level, this is, every other sub-extension is represented with elements of the previous sub-extension.

The tower can be constructed using quadratic and cubic extensions of the previous base field until we reach the desired extension field ( $\mathbb{F}_{p^k}$ ). The table 2.1 presents the recommended choice of sub-extensions for selected embedding degrees [BS09]. The first column

k	Construction	Tower
8	KSS	1-2-4-8
12	6.8	1-2-4-12
18	6.12	1-3-6-18
24	6.6	1-2-4-8-24
36	6.14	1-2-6-12-36

Table 2.1: Tower extension construction

is the  $k$  embedding degree, the second column is the name or the construction number of the family of elliptic curves from the Taxonomy paper [FST10]. The last column shows the degree of the extension fields to be constructed to get the final extension degree.

For example, the first row in table 2.1 correspond to the KSS curves with  $k = 8$ . For this family, we construct a finite field  $\mathbb{F}_p$ , on top of that, we construct a tower of extension fields in the following order:  $\mathbb{F}_{p^2}$ , then  $\mathbb{F}_{p^4}$ , and finally  $\mathbb{F}_{p^8}$ .

In [BN06] Barreto and Naehrig proposed the use of a polynomial  $X^6 - \xi$ , where  $1/\xi = \lambda^2\mu^3$  with  $\lambda \in \mathbb{F}_p$  a non-cube, and  $\mu \in \mathbb{F}_{p^2}$  a non-square. Corollary 5 from the Benger and Scott method [BS09] says that a polynomial  $x^m - (a \pm b\sqrt{-1})$  is irreducible over  $\mathbb{F}_{p^2}$  if  $a^2 + b^2$  is neither a square nor a cube in  $\mathbb{F}_p$ . We can trivially find either a small  $(a, b)$  or  $(\lambda, \mu)$  pair with a linear search.

Some of the values for  $(a, b)$  in the Benger and Scott paper are already calculated and are based on some values of  $p$ , the parameter defining the base field. See table 2.2 for a selection of the pairs.

Column 4 from table 2.2 shows the  $(a, b)$  pair that can be used in the tower construction. Column 3 show the values for which column 4 is valid. Column 2 sets the degree of the irreducible polynomial. Column 1 lists the degree of the first tower, or the construction number on which the  $(a, b)$  pair applies.<sup>1</sup>

<sup>1</sup>In the case of construction 6.12, since  $x \equiv 14 \pmod{42}$ ,  $x = 42x_0 + 14$ .

$\mathbb{F}_{p^n}$	$X^m$	Test	$(a, b)$
2	$2^i 3^j$	$p \equiv 3 \pmod{8}$	(1,1)
2	$2^i 3^j$	$p \equiv 2, 3 \pmod{5}$	(2,1)
Construction	$X^m$	Test	$(a, b)$
6.8	6	$x \equiv 7, 11 \pmod{12}$	(1,1)
6.8	6	$x \equiv 1, 3, 7, 11, 12, 13 \pmod{15}$	(1,2)
6.8	12	$x \equiv 1, 3, 7, 11, 12, 13 \pmod{15}$	(5,0)
6.12	18	$x_0 \equiv 1, 4, 5, 8 \pmod{12}$	(2,0)
6.12	18	$x_0 \equiv 7, 9, 12, 14 \pmod{15}$	(5,0)
6.12	18	$x_0 \not\equiv 2, 3, 4 \pmod{9}$	(3,0)

Table 2.2: Irreducible polynomials for the tower extension construction

### 2.1.3 Twist of a curve over an extension field

A twist curve  $E'$  defined over  $\mathbb{F}_{p^e}$ , with  $e = \frac{k}{d}$ , is another elliptic curve isomorphic to  $E$  defined over  $\mathbb{F}_{p^k}$ .

We define  $d$  as the degree of the twist of the curve  $E$ . The values of  $d$ , with a CM field defined as  $\mathbb{Q}(\sqrt{D})$  are as follows [HSV06]:

- $d = 6$  if the curve  $E$  has CM discriminant  $D = -3$ , and  $j(E) = 0$ .
- $d = 4$  if the discriminant is  $D = -1$ , and  $j(E) = 1728$ .
- $d = 2$  for any other value of  $D$ , and  $j(E) \neq 0, 1728$ .

When  $d = 2$ , the curves is said to have or support a quadratic twist, when  $d = 3$ : cubic twist, when  $d = 4$ : quartic twist, and when  $d = 6$ : sextic twist.

The formulae for an elliptic curve  $E$  is  $y^2 = x^3 + a.x + b$ , defined over  $\mathbb{F}_p$ , whereas the formulae for the twisted curve  $E'$  depends on the degree of the twist, and are as follows:

$$\begin{aligned}
 E' : y^2 &= x^3 + 1/D^2 ax + 1/D^3 b && \text{for } d = 2 : \\
 E' : y^2 &= x^3 + 1/D ax && \text{for } d = 4 \\
 E' : y^2 &= x^3 + 1/D b && \text{for } d = 6
 \end{aligned}$$

where  $D \in \mathbb{F}_{p^e}$  such that  $W^d - D$  is irreducible over  $\mathbb{F}_{p^e}[W]$ .



Furthermore, if  $\delta \in \mathbb{F}_{p^k}$  is a root of  $W^d - D$ , then there exists a homomorphism which maps points on the twist  $E'$  to the points of the curve  $E$  as follows:

$$\psi : E'(F_{p^{k/d}}) \rightarrow E(\mathbb{F}_{p^k}) \text{ defined by: } (x', y') \rightarrow (x' \cdot \delta^{1/3}, y' \cdot \delta^{1/2}),$$

with an isomorphism given by:

$$\phi : \mu_d \rightarrow \text{Aut}(E) : \delta \mapsto [\delta] \text{ with } [\delta](x, y) = (x \cdot \delta^2, y \cdot \delta^3) \text{ [HSV06]}.$$

One way to compute this map is by pre-computing  $\delta^2, \delta^3$ .

### 2.1.4 ECDLP

The *Elliptic Curve Discrete Logarithm Problem* (ECDLP) may be described as follows: Given an elliptic curve  $E$  defined over a finite field  $\mathbb{F}_p$ , a point  $P \in E(\mathbb{F}_p)[r]$ , and a point  $Q$  in the group of points generated by  $P$ , find the integer  $n \in [0 \dots r - 1]$  such that  $Q = nP$ . The integer  $n$  is called the discrete logarithm of  $Q$  with respect to  $P$ .

The hardness of the ECDLP is essential for the security of all elliptic curve cryptographic schemes [HMOV04]. Computing  $nP$  should be easier than finding  $n$  (solving the discrete log), otherwise the system can be easily comprised. On the another hand, the cost of finding  $n$  should be significantly greater than the value of  $n$ .

One way to find  $n$  is by a linear search:  $R = P$ , for  $n = 0$  to  $r - 1$ :  $R + P = P$  until  $R = Q$ . For sufficiently large values of  $r$  there is no feasible algorithm to find  $n$ . Appendix B gives details of an attack to finding  $n$  for a “sufficiently large” value of  $r$ .

#### 2.1.4.1 Security level

The main concern about the security of a system, is how long does an attacker need to break it, and how many resources are needed. The cost of breaking a system, measured both in time and in money invested in computational resources, should be greater than the value of the protected information. For example, if an attacker would need the equivalent of one million Euro to get access to an asset worth a hundred Euro, we presume that the attacker would cease on his attempt.

Equivalent symmetric key size		80	112	128	192	256
NIST	$\text{Log}_2(\mathbb{F}_{p^k})$	1024	2048	3072	7680	15360
	$\text{Log}_2(r)$	160	224	256	384	512
ECRYPT	$\text{Log}_2(\mathbb{F}_{p^k})$	1248	2432	3248	7936	15424
	$\text{Log}_2(r)$	160	224	256	384	512

Table 2.3: NIST and ECRYPT security level recommendations

When we say that an attacker would need 100 years to break the security of a system, it is usually calculated with the computational resources available at the start of the attack. As the time passes, newer, faster and cheaper computational resources appear; an attacker can be upgrading its computational resources during the attack, reducing the initial 100 years attack into a smaller number of years.

On symmetric cryptographic schemes, the lifetime of the protection of the data can be calculated straight-forward by the time it would take to break the security by brute force, guessing or other attack, and including the improvements on computational power during the time of the attack.

The traditional RSA cryptosystems has its hardness based on the integer factoring problem. In the case of elliptic curve cryptography, the hardness is based on the discrete log of elliptic curve groups defined over finite fields.

It is traditionally suggested that an elliptic curve in a subgroup of size  $m$ -bit is equivalent to a  $m/2$ -bit symmetric key.

In [NIS] NIST states that an 80-bit symmetric key is equivalent to a 160-bit one using discrete logs subgroups and elliptic curve groups. This is defined as a 80-bit security level, and it is not recommended for use after 2012. An 128-bit security level is recommended therefore after that year. The ECRYPT network also publishes its own recommendations [ECR].

Table 2.3 shows the NIST and the ECRYPT recommendations of the pairing-friendly subgroup and the size of the finite field extension for several security levels.

### 2.1.5 Elliptic Curve Isomorphisms

**Definition 9.** The ring of endomorphisms of  $E$ , defined over  $\mathbb{F}_p$  is denoted by  $\text{End}_{\mathbb{F}_p} E$ . For any integer  $n$ , the multiplication-by- $n$  map  $P \mapsto [n]P$  is an endomorphism of  $E$ .

**Definition 10.** The Frobenius endomorphism, denoted  $\pi_p$ , is the mapping given by [Men93]:

$$\pi_p : E \longmapsto E : (x, y) \longmapsto (x^p, y^p)$$

satisfying  $\pi_p^k(Q) = Q$  for any  $Q \in E(\mathbb{F}_{p^k})$ .

**Definition 11.** An efficiently computable homomorphism:  $\psi^i = \phi^{-1} \pi_p^i \phi$ , where

$\phi : E'(\mathbb{F}_{p^{k/d}}) \rightarrow E(\mathbb{F}_{p^k})$  is the isomorphism which takes a point from the twisted curve  $E'(\mathbb{F}_{p^{k/d}})$  to the isomorphic group on  $E(\mathbb{F}_{p^k})$ , and  $\pi$  is called the  $p$ -power Frobenius map on  $E$ . Then  $\psi : \phi^{-1} \pi \phi$  is an endomorphism  $\psi : G'_2(E') \rightarrow G_2(E)$ , where  $G'_2$  and  $G_2$  are the group of points on the twisted curve and on the elliptic curve over the extension field respectively.

## 2.2 Divisors

We recall the following definitions from [Sil86]:

**Definition 12.** The divisor group of a curve  $C$ , denoted  $\text{Div}(C)$ , is the free Abelian group generated by the points of  $C$ . Thus a divisor  $D \in \text{Div}(C)$  is the formal sum:

$$D = \sum_{P_i \in C} n_{P_i}(P).$$

with  $n_{P_i} \in \mathbb{Z}$  and  $n_{P_i} = 0$  for all but finitely many  $P \in C$ .

**Definition 13.** The degree of  $D$  is defined by

$$\deg D = \sum_{P_i \in C} n_{P_i}.$$

**Definition 14.** The divisors of degree 0 form a subgroup of  $\text{Div}(C)$ , denoted as:

$$\text{Div}^0(C) = \{D \in \text{Div}(C) : \deg D = 0\}.$$

**Definition 15.** If  $C/K$ , then  $G_{\bar{K}/K}$  acts on  $\text{Div}(C)$  and  $\text{Div}^0(C)$  as

$$D^\sigma = \sum_{P_i \in C} n_{P_i}(P^\sigma).$$

It follows that  $D$  is defined over  $K$  if  $D^\sigma = D$  for all  $\sigma \in G_{\bar{K}/K}$ .

**Definition 16.** Let  $C$  be a smooth curve and  $f \in \bar{K}(C)^*$ . The divisor  $\text{div}(f)$  of  $f$  is given by

$$\text{div}(f) = \sum_{P_i \in C} \text{ord}_{P_i}(f)(P).$$

**Definition 17.** A divisor associated to a function is called the principal divisor. The set of principal divisors forms a group  $\text{Princ}_C$ .

The  $\text{div}(f)$  can be represented as the difference of the following divisors:

$$\text{div}(f) = \text{div}(f)_0 - \text{div}(f)_\infty.$$

the points in  $\text{div}(f)_0$  with non-zero coefficients are called *zeroes*. Similarly, the points in  $\text{div}(f)_\infty$  with non-zero coefficients are called the *poles*.

## 2.3 Cyclotomic Polynomial

The  $n^{\text{th}}$  cyclotomic polynomial  $\Phi_n(x)$  is defined to be  $\prod(x - \zeta)$ , where  $\zeta$  ranges over the primitive  $n^{\text{th}}$  roots of unity in  $\mathbb{C}$ .

**Definition 18.** The Euler totient function, represented as  $\varphi$  is:

$$\varphi(N) = |\{x | 1 \leq x \leq N, \gcd(x, N) = 1\}|.$$

The  $\Phi_n(x)$  is an irreducible polynomial in  $\mathbb{Z}[x]$ . The  $\deg \Phi_n(x) = \varphi(n)$  [LL96].

The irreducible factorization of  $x^t - 1$  in  $\mathbb{Z}[x]$  is given by  $x^t - 1 = \sum_{d|t} \phi_d(x)$ , where  $\phi_d(x)$  is the  $d^{\text{th}}$  cyclotomic polynomial. The factor  $\phi_t(x)$  is the only irreducible factor  $x^t - 1$  that does not appear in the factorization of  $x^s - 1$  for divisors  $s$  of  $t$  with  $s < t$  [Len97].

## 2.4 Pairings

Originally there were two pairings used in cryptography, the Weil and Tate-Lichtenbaum pairings, both evaluated using the *Miller Algorithm*. The Weil pairing requires two *Miller loops* to generate the  $r^{\text{th}}$  roots of unity [Sil86, III.§8], while the Tate pairing requires an exponentiation of a single application of the *Miller loop* [Hes08], making it more efficient than the Weil pairing [GPS06].

The Miller Algorithm uses  $\lg(r - 1)$  iterations. It uses a double-and-add, and a line-and-tangent approach. The basic Miller loop is presented in Algorithm 1, and explained in the following.

---

### Algorithm 1 Basic Miller loop

---

**Input:**  $P, Q \in E(\mathbb{F}_p)[r]$ ,  $P \neq Q$  and  $l \in \mathbb{N}$

**Output:**  $f \in E(\mathbb{F}_p)$

$T \leftarrow P$

$f \leftarrow 1$

$i \leftarrow \lfloor \lg_2(l) \rfloor - 1$

**for**  $i = 0$  to  $l - 1$  **do**

$f \leftarrow f^2 \cdot l_{T,T}(Q)/v_{[2]P}(Q)$

$T \leftarrow [2]P$

**if**  $l_i = 1$  **then**

$f \leftarrow f \cdot l_{T,P}(Q)/v_{T+P}(Q)$

$T \leftarrow T + P$

**end if**

**end for**

**return**  $f$

---

### 2.4.1 The Tate Pairing

The Tate pairing was introduced by Tate as a rather general pairing on Abelian varieties over local fields. Lichtenbaum gave an application of this pairing to the Jacobians of curves over local fields. The Tate-Lichtenbaum pairing is hereafter referred to as the Tate pairing. [CF06]

Let  $P \in E(\mathbb{F}_p)[r]$  and  $Q \in E(\mathbb{F}_{p^k})$ , and consider the divisor  $D = (Q + S) - (S)$  with  $S$  a random point in  $E(\mathbb{F}_{p^k})$ . Let  $f_{a,P}$  be a function with a divisor

$$(f_{a,P}) = a(P) - (aP) - (a-1)(0)$$

for  $a \in \mathbb{Z}$ . A non degenerate, bilinear Tate pairing is the map:

$$\begin{aligned} e_r : E(\mathbb{F}_p)[r] \times E(\mathbb{F}_{p^k})/rE(\mathbb{F}_{p^k}) &\rightarrow \mathbb{F}_{p^k}^*/(\mathbb{F}_{p^k}^*)^r \\ (P, Q) &\mapsto f_{r,P}(Q) \end{aligned}$$

The value of the pairing is in an equivalence class,  $\mathbb{F}_{p^k}^*/(\mathbb{F}_{p^k}^*)^r$ . For practical purposes it is preferred to raise the value of the pairing to the power of  $(p^k - 1)/r \in \mathbb{F}_{p^k}^*$  to obtain a unique representative of the class, and to make it bilinear [Hes08]. This exponentiation is known as the *final exponentiation*, and the pairing is referred to as the *Reduced Tate Pairing*.

The Tate pairing becomes:

$$e_r(P, Q) : (P, Q) \mapsto f_{r,P}(Q)^{(p^k-1)/r}. \quad (2.2)$$

The focus of Chapter 4 will be on how to speed-up this operation.

We define the  $G_1 \in E(\mathbb{F}_p)[r]$  as the group of points of order  $r$  on  $E$  over the base field  $\mathbb{F}_p$ , and  $G_2$  as the group on  $E(\mathbb{F}_{p^k})/rE(\mathbb{F}_{p^k})$ . Let  $P$  be a point in  $G_1$  and  $Q$  a point in  $G_2$ .

### 2.4.1.1 The Miller Loop for the Tate Pairing

Implementing the Tate Pairing is almost as easy as implementing the *Miller loop*. A straightforward method for implementing the Miller's algorithm is explained in [BLKS02, Theorem 2] and [Sco07].

For each pair  $A, B \in E(\mathbb{F}_p)$ , we define  $g_{A,B} : E(\mathbb{F}_{p^k}) \rightarrow \mathbb{F}_{p^k}$  as the equation of the line through the points  $A, B$  to the curve  $E(\mathbb{F}_{p^k})$  [BLKS02].

The code in Listing 2.1 computes the line function  $l_{A,B}(Q)$ , where  $A, B$  are two points on the curve  $E(\mathbb{F}_p)$ , and a point  $Q$  on the curve  $E$  defined over the extension. This function is required to evaluate the contribution to the pairing value of the elliptic curve point addition,  $A + B$ . In essence these are distances calculated between the fixed point  $Q$  and the lines that arise when adding points  $A$  and  $B$ . In this code there are three cases to be considered.

Listing 2.1: Line function code in Magma

---

```
//Input <- A,B in G_1, Q in E(\F_{p^k})
//Output -> return in G_T
L:= function(A,B,Q)
    if A eq -B then
        return Q[1]-A[1];
    end if;
    if A eq B then
        lambda:=(3*A[1]^2+a) / (2*A[2]);
    else // A ne B
        lambda:=(B[2]-A[2]) / (B[1]-A[1]);
    end if;
    return (lambda*(Q[1]-A[1]) + A[2]-Q[2]);
end function;
```

---

The first case is where  $A = B$ , the line passing through this point is the tangent to the curve at point  $A$  the second case is when  $A \neq B$ . These two cases use the following operation:  $(\text{lambda} * (Q[1] - A[1]) + A[2] - Q[2])$ , where  $\text{lambda}$  is the slope of the line. The last case is when the point  $A$  is a negative of point  $B$ . In this case we have a vertical line and we compute  $Q[1] - A[1]$ . Refer to [Sil86] or [BLKS02] for more details.

In practice, the equation of the curve, defined with a tuple  $(a, b) \in \{(-3, b), (0, b \neq$

$0), (a \neq 0, 0)\}$  is preferred as it simplifies the line function operations.

The full *Tate Pairing* algorithm is presented in Listing 2.2. In this code the double-and-add stages can be identified by a  $2 * T$  (double) and a  $T + P$  (add), where the addition is required when a 1 (a pole) is present in the binary expansion of  $r$ . Additionally, there is the “vanilla” implementation of the final exponentiation step, using *direct exponentiation*.

Listing 2.2: Basic Tate pairing with single Miller Loop

---

```
// Input <- P \in G_1, Q \in E(\mathbb{F}_{p^k})
// Output -> f \in G_T
pairing:= function (P,Q)
f:=1;
T:=P;
i:=Floor(Log(2,r))-1;
si:=Intseq(r,2);
while i ge 0 do
    f:=f^2*L(T,T,Q);
    T:=2*T;
    if si[i+1] eq 1 then
        f:=f*L(T,P,Q);
        T:=T+P;
    end if;
    i:=i-1;
end while;
f:=f^((p^k)-1 div r);
return f;
end function;
```

---

### 2.4.2 The ate Pairing

The ate pairing [HSV06] is a variant of the Tate pairing and it is a generalization of the Eta pairing, a pairing that can be used with supersingular elliptic curves [BGHS07]. The ate pairing is particularly suitable for pairing-friendly curves with small values of  $t$ .

We take  $G_1 = E[r] \cap \text{Ker}(\pi_p - [1])$  and  $G_2 = E[r] \cap \text{Ker}(\pi_p - [p])$ , and let  $T = t - 1$ . Let  $N = \gcd(T^k - 1, p^k - 1)$  and  $T^k - 1 = LN$ . For  $Q \in G_2$  and  $P \in G_1$ , the ate pairing is defined as [HSV06]:

$$e_T : (Q, P) \longmapsto f_{T,Q}(P)^{c_T(p^k-1)/N},$$

where  $c_T = \sum_{i=0}^{k-1-i} p^i \equiv kp^{k-1} \pmod{r}$ . The ate pairing is a bilinear, non-degenerate



pairing if  $r \nmid L$ . There is a switch in the arguments with respect of the Tate pairing; the first parameter  $Q$  is defined over the extension field and  $P$  is defined over the base field.

We can always use the traditional final exponentiation  $(p^k - 1)/r$  if  $T^k \not\equiv 1 \pmod{r^2}$ , since  $r \nmid N$  and  $r \nmid c$  the function is always bilinear, and non-degenerate as a result of Theorem 2 of [Hes08].

#### 2.4.2.1 The Miller Loop for the ate Pairing

The number of iterations in the ate pairing in the Miller loop depends on the size of the trace of the Frobenius  $t$  rather than on the size of the subgroup  $r$ . As a result, if  $(\omega = \log r / \log |t|) > 1$  for a particular family then it is possible to compute the ate pairing faster than the Tate pairing for that family of curves [Sco07]. The larger the value of the  $\omega$  the faster the ate pairing computation is compared to the Tate pairing computation.

For example, the KSS curves with embedding degree  $k = 18$  have  $\omega = 3/2$ , and so, this curve is suitable for an implementation of the ate pairing. The code in Magma for computing the ate pairing is given in Listing 2.3, where  $G_1 \in E(\mathbb{F}_p)[r]$  and  $G_2 \in E'(\mathbb{F}_{p^3})$ . Note that here we define  $G_2$  on the sextic twist.

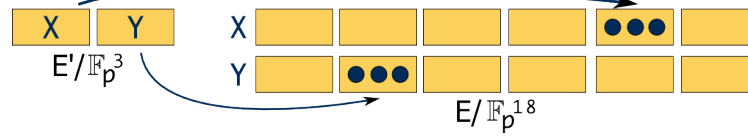
Listing 2.3: ate pairing with single Miller Loop

---

```
//Input <- P \in $G_2$, Q \in $G_1$
//Output -> f \in $G_T$
pairing:= function(P,Q)
f:=1;
T:=P;
s:= t-1;
i:=Floor(Log(2,s))-1;
si:=Intseq(s,2);
while i ge 0 do
    f:=f^2*L(T,T,Q);
    T:=2*T;
    if si[i+1] eq 1 then
        f:=f*L(T,P,Q);
        T:=T+P;
    end if;
    i:=i-1;
end while;
f:=f^((p^k)-1 div r);
return f;
end function;
```

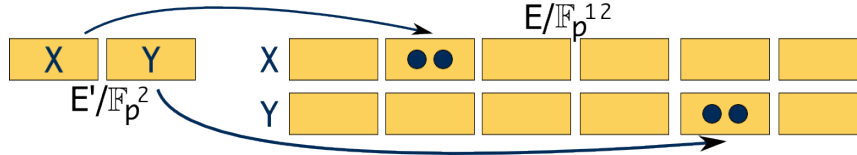
---

In the Miller loop, however, some operations on the curve over the extension field are required; in the case of the ate pairing it is the line function. Since one is using a point  $Q \in E'(\mathbb{F}_{p^{k/d}})$  instead of  $E(\mathbb{F}_{p^k})$ , it is therefore required to apply the map  $\psi$ , discussed in §2.1.5, in the line function.

Figure 2.4:  $\psi$  map for KSS: $k = 18$  curves

Each of the  $x$  and  $y$  coordinates of a point  $Q \in E'(\mathbb{F}_{p^3})$  has three components defined over the base field  $\mathbb{F}_p$ . A point in a curve  $E(\mathbb{F}_{p^{18}})$  has six components of  $\mathbb{F}_{p^3}$ , or eighteen of  $\mathbb{F}_p$ . Instead of applying the map function  $\psi$ , one can move the  $x$  and  $y$ -coordinates of  $Q$  to their corresponding positions in a point in a curve  $E(\mathbb{F}_{p^{18}})$ , and apply the Frobenius exponentiation part there.

A similar approach can be taken for the BN curves:

Figure 2.5:  $\psi$  map for BN curves

Unfortunately, this cannot be done in Magma, and we don't have this transformation for other families of curves. Instead we apply the “full”  $\psi$  map operation  $(x, y) \mapsto (x.\delta^2, y.\delta^3)$ . Since  $\delta$  is a constant for the generated curve, we can pre-compute these values and the cost of the mapping will be just one multiplication by a coordinate component (each time the line function in Listing 2.1 is called.)

Listing 2.4 shows the corresponding code for the Line function.

Listing 2.4: Line Function with  $\psi$  map

---

```
//Input <- A,B \in G_2, Q \in G_1
//Output -> return in G_T
```

---

---

```

L:= function (A,B,Q)
  Ax:=A[1]*deltas[1];
  Ay:=A[2]*deltas[2];
  Bx:=B[1]*deltas[1];
  By:=B[2]*deltas[2];
  if (Ax eq Bx) then
    if (Ay eq -By) then
      return Q[1]-Ax;
    else
      if (Ax eq Bx) and (Ay eq By) then
        lambda:=(3*Ax^2+a) / (2*Ay);
      end if;
    end if;
  else // A ne B
    lambda:=(By-Ay) / (Bx-Ax);
  end if;
  return (lambda*(Q[1]-Ax) + Ay-Q[2]);
end function;

```

---

Alternatively, if  $\delta \in E(\mathbb{F}_{p^{k/d}})$  already, then we do not need to lift the element from the twisted curve to the curve over the extension field to apply the map.

### 2.4.3 The R-ate Pairing

The R-ate pairing introduced by Lee, Lee and Park [LLP08], is a generalization of the ate [HSV06] and  $\text{ate}_i$  [ZZH07] pairing, improving the computation efficiency. It takes three short *Miller loops* to calculate the pairing, that together require a shorter loop than a single typical application of the Tate or the ate pairing. The *R-ate* pairing can be regarded as a ratio of two pairings, hence the name.

Corollary 3.3 from [LLP08] defines four combinations of the R-ate pairing. Combination 1 matches the Miller loop length of the  $\text{ate}_i$  pairing. Combination 2 requires a Miller loop length of the field size, which is not optimal. Combination 4 matches the Miller loop length of the Tate pairing.

In this research, we prefer the *R-ate* pairing option 3 since we can achieve the smallest number of Miller loops, and from now on, we refer to it simply as the R-ate pairing, unless specified. The R-ate pairing, like the  $\text{ate}_i$  [ZZH07], requires the calculation of  $T_i \equiv p^i \bmod r$ , with  $0 \leq i < k$ , where  $k$  is the *embedding degree*. This pairing is constructed from the parameters  $(p, r)$ , which are also used to define the ate and Tate pairing, and a combination

of the  $T_i$ s.

The definition of the *R-ate* pairing with  $A = aB + b$  where  $A = T_i$ ,  $B = T_j$ , and  $a, b, \in \mathbb{Z}$  is as follows:

$$e_{A,B}(P, Q) : (P, Q) \mapsto f_{a,BP}(Q) \times f_{b,P}(Q) \times G_{aBP,bP}(Q)$$

Generally this equation does not automatically give a bilinear and non-degenerate pairing. However, with a careful choice of pairs  $A$  and  $B$ , this is possible. For efficiency, we look for a working and non-trivial combination of  $A$  and  $B$  that gives the shortest *Miller loop*.

In [LLP08, Algorithm 2] there are three *Miller loops* to compute and the *final exponentiation* is calculated at the end of the computation. The code for the *Miller loop* denoted as  $M$  in Listing 2.5 can be used to compute the *R-ate pairing*. It makes function calls to Listing 2.1, the *line function*.

Listing 2.5: Miller loop as to be used by the *R-ate* pairing

---

```
// Input <- P \in G_2, Q \in G_1, l \in \mathbb{Z}
// Output -> f \in G_T, T \in \mathbb{Z}
M:= function (P,Q,l)
T:=P;
f:=1;
i:=Floor(Log(2,l))-1;
li:=Intseq(l,2);
while i ge 0 do
  f:=f^2*L(T,T,Q);
  T:=2*T;
  if li[i+1] eq 1 then
    f:=f*L(T,P,Q);
    T:=T+P;
  end if;
  i:=i-1;
end while;
return f,T;
end function;
```

---

Listing 2.7 shows an implementation of the *R-ate* pairing function. As described in [LLP08, Algorithm 2], we denote  $f_a, f_b, aQ$  and  $bQ$  as  $\{a, b\} = \{m1, m2\}$ , where  $m1$  and  $m2$  are as follows:  $m1 = \max\{a, b\}$  and  $m2 = \min\{a, b\}$ .

The *R-ate* pairing is also denoted as  $R_{A,B}(P, Q)$ . For the KSS  $k = 18$  curve the linear

combination:  $T_{13} = \frac{3x}{7} \cdot T_6 + \frac{2x}{7}$ , where  $a = \frac{3x}{7}, j = 6$  and  $b = \frac{2x}{7}$  gives the shortest bilinear and non-trivial *R-ate* pairing.

We know from [LLP08, Algorithm 2] that  $c \leftarrow \lfloor \frac{m_1}{m_2} \rfloor$  and  $d \leftarrow m_1 - c \times m_2$ . For the  $a$  and  $b$  values proposed, we set  $d = \frac{a}{2}$ . With these values, the first and third Miller function call can be integrated into one as shown in Listing 2.6. This is a good choice since  $A$  and  $B$  have almost the same number of bits with both values less than the  $x$ -parameter, the parameter of the curve, and as such, it provides a short *Miller length* of which one can use an intermediate value of  $f$  and  $T$  from Listing 2.5 to reuse computation. Listing 2.6 presents the optimised Miller's loop for the *R-ate* pairing reusing partial results.

Listing 2.6: Modifications to the Miller loop for the R-ate pairing

---

```
// Input <- P \in G_2, Q \in G_1, l \in \mathbb{Z}
// Output -> f \in G_T, T \in \mathbb{Z}
M:= function (P,Q,l)
T:=P;
f:=1;
i:=Floor(Log(2,l))-1;
li:=Intseq(l,2);
while i ge 0 do
  f:=f^2*L(T,T,Q);
  T:=2*T;
  if li[i+1] eq 1 then
    f:=f*L(T,P,Q);
    T:=T+P;
  end if;
  if i eq 1 then
    f2:=f; T2:=T;
  end if;
  i:=i-1;
end while;
return f, T, f2, T2;
end function;
```

---

Since the parameters of the first and the third Miller loop are the same exempt the loop length, we only need to verify that  $d$  is equal to the most significant bits of  $m_2$ . Unless this condition is met, we use either twice the function at Listing 2.5 or a single execution of the function at Listing 2.6.

Another interesting characteristic of the selected combination is that if  $c = 1$ , we also avoid some computations. Furthermore, one can omit unnecessary use of memory since by

definition;  $a, b, j, m_1, m_2, c, d, f_{cm2}$  and  $cm2Q$  are known in advance.

Listing 2.7: R-ate pairing function

---

```
// Input <- P \in G_1, Q \in G_2, l \in \mathbb{Z}
// Output -> f \in G_T, T \in \mathbb{Z}
pairing:= function(P,Q)
dd:=(xx) div 7;
m1:=(3*dd);
m2:=(2*dd);
jj:=6;

//Compute
fm2,m2Q,fd,dQ:=M(Q,P,m2);

f1:=fm2*fd;
fm1:=f1*L(m2Q,dQ,P);
m1Q:=m2Q+dQ;

//Exponentiation
f2:=Frobenius(fm1,Fp,jj)*fm2;
Q1:=[Frobenius(m1Q[1]*V2,Fp,jj),Frobenius(m1Q[2]*V3,Fp,jj)];
Q1:=ExtT! [Q1[1]*V1_2,Q1[2]*V1_3];
f3:=f2*L(Q1,m2Q,P);

//Final Exponentiation
f3:=f3^(((p^k)-1) div r);

return f3;
end function;
```

---

The code, “ $fm2, m2Q, fd, dQ := M(Q, P, m2)$ ”, in the *Compute* section of Listing 2.7 returns  $f_{m_2}$  as  $f$  and  $m2Q$  as  $T$  in Listings 2.5 or 2.6. Clearly  $Q$  and  $m_2$  are the parameters of the  $M$  function, where  $m_2$  is the *Miller loop length*. Hence  $m2Q$  is the actual state of  $T$ . Additionally, since there is only one  $M$  function call, one would be able to insert the *Miller loop* inside a modified version of the Listing 2.7 to avoid overhead at runtime. Finally,  $Q1 \in E(\mathbb{F}_{p^{18}})$  and the line function requires it to be in the twisted curve  $E'(\mathbb{F}_{p^3})$ ; hence, we apply the inverse  $\psi$  map from §2.1.5 before the line function.

#### 2.4.3.1 Shorter R-ate pairing

Recalling notation from [LLP08], to evaluate the R-ate pairing, the generic algorithm can be used. Then, the overall shortest Miller loop can be constructed as follows:

The three Miller loop calls are:

$$M(Q, P, m_2) \tag{2.3}$$

$$M(m_2 Q, P, c) \tag{2.4}$$

$$M(Q, P, d) \tag{2.5}$$

where the parameters for these Miller function calls are as follows:

$$m_1 \leftarrow \max\{a, b\}; \tag{2.6}$$

$$m_2 \leftarrow \min\{a, b\}; \tag{2.7}$$

$$c \leftarrow \left\lfloor \frac{m_1}{m_2} \right\rfloor; \tag{2.8}$$

$$d \leftarrow m_1 - c \cdot m_2. \tag{2.9}$$

and  $M$  is the Miller function.

If an  $(A, B)$  pair is chosen such that the parameters are close to each other, then the ratio can be 1 (or close to 1), leaving one Miller loop that does not require execution as the result can be known in advance<sup>2</sup>. The other two function calls can be embedded inside each other to use a partial result from the computation [DKS, Listing 15].

On the another hand, if the chosen parameters have a high ratio, some speed improvement is still possible. For example, some of the intermediate values of the R-ate pairing can be computed in parallel. This is not always possible for all of the families of pairing-friendly curves and still a smaller ratio on the parameters should be faster than distributed loops. Additionally, in our best effort, there are no family of curves for which the  $T_i, T_j$  combinations suggest that it is possible to compute two parallel Miller loops instead of one short loop, and gain speed<sup>3</sup>.

For example, we examine the case of the KSS curves with embedding degree  $k = 18$

---

<sup>2</sup>we can at least simplify the code

<sup>3</sup>If the third Miller loop needs to be embedded into the first one, it means that we need the second Miller loop, otherwise the first loop is not needed

Miller-length in iterations								
#	a	b	T <sub>j</sub>	loops	Ham	m <sub>2</sub>	c	d
1	3	$x$	1	64	30	1	62	1
2	$5x/7$	$x^2/7$	1	187	85	63	61	63
3	$3x/7$	$2x^2/7$	1	186	90	62	63	61
4	$5x^2/49$	$3x^2/49$	3	246	119	123	0	123
5	$5x^2/49$	$8x^2/49$	6	247	125	124	0	123
6	$5x/14$	$3x^2/14$	16	186	88	62	63	61
7	$8x^2/49$	$3x^2/49$	6	247	119	123	1	123
8	$3x/7$	$2x/7$	6	62	22	62	0	61
9	$2x/7$	$3x/7$	12	62	22	62	0	61
10	$3x/14$	$x^2/14$	7	184	89	61	62	61
11	$8x^2/49$	$5x^2/49$	12	247	125	124	0	123
12	$3x^2/49$	$8x^2/49$	12	247	119	123	1	123
13	$3x^2/49$	$5x^2/49$	15	246	119	123	0	123

Table 2.4: KSS:  $k = 18$  Curves A,B Parameters

[KSS08] in Table 2.4 with a subgroups size of approximately 192-bits. The first and second columns give the  $a$  and  $b$  parameters from  $T_i = a.T_j + b$ . The ‘loops’ column presents the total number of loops necessary for the R-ate pairing computation. The columns ‘ $m_2$ ’ and ‘ $c$ ’ represent the parameters of the Miller loop in bits (as given in formulæ 2.3 and 2.4). The column ‘Ham’ shows the Hamming-weight of the length of the Miller loops.

We see in Table 2.4 that for the example illustrated above, there are several  $(A, B)$  pairs which already have the shortest Miller loop length: row 8 and 9. These are our recommended parameters for the R-ate pairing.

As mentioned before, other families of pairing-friendly elliptic curves may present favourable ratios and this feature (parallel execution) can be exploited. Generating code for parallel computation is left as a future option.

#### 2.4.3.2 Choosing the R-ate pairing parameters

Choosing the length parameters of the R-ate pairings is an easy task that requires the  $p(x)$  and  $r(x)$  parameters of the desired curve. See Algorithm 2.

Once the  $T_i$  polynomials are generated, it is necessary to verify the bilinearity and non-



**Algorithm 2** Generate the  $T_i - T_j$  combinations

---

```

for  $i = 0$  to  $k - 1$  do
   $\text{Poly}_i \leftarrow p^i \bmod r$ 
   $\diamond$  Discard  $\text{Poly}_i \in [1, -1]$ 
  for  $j = i + 1$  to  $k - 1$  do
     $\diamond$  Discard  $\text{Poly}_j \in [1, -1]$ 
     $\diamond$  Discard negative A,B components
     $\diamond$  Discard trivial combinations
     $A_i \leftarrow \left[ \frac{\text{Poly}_i}{\text{Poly}_j} \right]$ 
     $B_i \leftarrow \text{Poly}_i \bmod \text{Poly}_j$ 
     $T_i \leftarrow i$ 
     $T_j \leftarrow j$ 
  end for
end for

```

---

degeneracy properties of the R-ate pairing with the given parameters.

We stress that the R-ate pairing requires an exponentiation to the parameter  $c$  from Equation 2.8; hence, when constructing the pairing, we may prefer to have a smaller footprint in favour of a slightly longer Miller loop. For example, the first row of table 2.4 is almost as short as row 8 and 9 but this selection would require an exponentiation of a number of approximately 62 bits.

#### 2.4.4 Pairing Lattices

The family of ate pairings [HSV06, GHO<sup>+</sup>07, ZZH07, MKHO07] are optimized versions of the Tate pairing restricted to the eigenspaces of the Frobenius.

Let  $s \in \mathbb{Z}$ ,  $h = \sum_{i=0}^d h_i x_i \in \mathbb{Z}[x]$  with  $h(s) \equiv 0 \bmod r$  and  $d = \varphi(k)$ , with  $k$  the embedding degree, and  $Q \in E(\mathbb{F}_{p^k})[r]$ , then:

$$(f_{s,h}, Q) = \sum_{i=0}^d h_i ((s^i Q) - (\mathcal{O})).$$

Defining  $\|h\|_1 = \sum_{i=0}^d |h_i|$  we have that if  $s$  is a primitive  $k^{\text{th}}$  root of unity modulo  $r^2$ ,

and if  $h(s) \equiv 0 \pmod r$  but  $h(s) \not\equiv 0 \pmod{r^2}$ , then,

$$e_{s,h} : (Q, P) \mapsto f_{s,h,Q}(P)^{(p^k-1)/r}$$

defines a bilinear and non-degenerate pairing [Hes08].

### Choosing $s$

For the choice of  $s$ , following the ate pairing definition, we can take  $s = r$ , the subgroup size, but we prefer to take  $s = T = t - 1$ , which is already an improvement with respect to the Tate pairing.

### Constructing $h$

For the case of  $h$ , we construct a matrix  $m \times m$ , with  $m = \varphi(k)$ :

$$M = \begin{pmatrix} r & 0 & \cdots & 0 \\ -T & 1 & 0 & \cdots & 0 \\ -T^2 & 0 & 1 & \cdots & 0 \\ \vdots & & & \ddots & \vdots \\ -T^{m-1} & 0 & \cdots & & 1 \end{pmatrix}$$

Let  $w = (w_0, w_1, \dots, w_{m-1})$  be the shortest  $\mathbb{Z}$ -linear combination of the rows of  $M$ , then we can construct  $h = \sum_{i=0}^{m-1} w_i x^i$ .

In [Hes08], the author suggest to LLL-reduce the matrix  $M$  to get the shortest vector. To find the expansion with short coefficients, in [Ver10] Vercauteren uses the “ShortestVectors()” Magma function. A method to get vectors of lower degree is by transforming the matrix to have the Weak Popov Form using Algorithm 3.

The explicit pairing lattice is defined as:

$$(Q, P) \mapsto \left( \prod_{i=0}^l f_{c_i, Q}^{p^i}(P) \cdot \prod_{i=0}^l G_{[s_{i+1}]Q, [c_i q^i]Q}(P) \right)^{(p^k-1)/r}$$

where  $s_i = \sum_{j=i}^l c_j p^j$ ,  $G$  is the line function, and  $k$  is even.

For the construction, note that:  $f_{1,Q} = 1$ ,  $f_{-1,Q} = 1/f_{1,Q}$  (for  $k$  odd), and  $[1/2]Q = [1/2 \bmod r]Q = [(r+1)/2]Q$ . [Ver10]

### BN Curves

For the BN curves [BN06], described in 2.13, we construct the matrix  $M$  from 2.4.4 as:

$$M_{BN} = \begin{pmatrix} -2x & -x+1 & x & -x \\ 5x-1 & 3x-2 & -x & 0 \\ -6x+1 & -6x+3 & 1 & 0 \\ \hline -6x+2 & & 1 & -1 & 1 \end{pmatrix}$$

The pairing lattice can be constructed with  $s = t - 1$ , and  $h(s) = (-6x+2, 1, -1, 1)$  as follows:

$$e_{s,h}(\cdot, \cdot) : (Q, P) \mapsto \frac{1}{f_{6x-2,Q}(P)} \cdot G_{-[6x-2]Q, Q_1-Q_2+Q_3}(P) \cdot G_{Q_1, -Q_2+Q_3}(P)$$

where  $Q^i$  is computed as  $\psi^i(Q)$ . At the end of the computation, we also need the final exponentiation.

### KSS Curves

For the KSS curves with  $k = 18$  [KSS08], described in 2.5.4.2

$$M_{k=18} = \begin{pmatrix} 10/7 & 0 & 6x/7 & 6/7 & 0 & -2x/7 \\ 0 & 1/7 & 0 & x/7 & 2/7 & 0 \\ 0 & 2x/7 & 1 & 0 & x/7 & 0 \\ 1/2 & 0 & x/2 & 1 & 0 & 0 \\ \hline -x & -3 & 0 & 0 & 1 & 0 \\ \hline 0 & -x & -3 & 0 & 0 & 1 \end{pmatrix} \quad (2.10)$$

The pairing lattice can be constructed with  $s = t - 1$ , and  $h(s) = (-x, -3, 0, 0, 1, 0)$  as follows:

$$e_{s,h}(\cdot, \cdot) : (Q, P) \mapsto \frac{1}{f_{x,Q}(P)} \cdot \pi \left( \frac{1}{f_{3,Q}(P)} \right) \cdot G_{\psi^4(Q), -3\psi(Q)}(P) \cdot G_{\psi^4(Q) - 3\psi(Q), -x.Q}(P)$$

At the end of the computation, we also need the final exponentiation.

The pairing lattice construction reduces the number of iterations of the Miller loop by exploiting powers of the Frobenius endomorphism  $\pi_p^i$  for  $0 \leq i < k$  by decomposing a multiple of  $r^4$  as a sum of this endomorphism.

### 2.4.5 Optimal Pairing

Let  $\lambda^i \equiv p^i \pmod{r}$  and  $r \mid \Phi_{k/d}(\lambda_i)$  with  $d = \gcd(i, k)$ , an ate pairing can be defined as

$$e_{\lambda_i} : (Q, P) \mapsto f_{\lambda_i, Q}(P)^{(p^k - 1)/r}$$

which implies that the minimum value of  $\lambda_i$  is  $r^{1/\varphi(k/d)}$ .

**Definition 19.** A pairing function  $e(\cdot, \cdot)$  is called an Optimal Pairing if it can be computed in  $\log_2 r / \varphi(k) + \varepsilon(k)$  basic Miller iterations, with  $\varepsilon(k) \leq \log_2 k$ . [Ver10]

### 2.4.6 Comparison of the Miller-loop length

In this section we compare the *Miller length* of the pairings implemented in sections §2.4.1, §2.4.2 and §2.4.3. This is the common comparison system for pairing functions using the same elliptic curve.

For the Tate pairing, presented in section §2.4.1, one can see how the number of Miller loops iterations is related to  $r$ . In Listing 2.2, we take the length to be  $\lfloor \log_2(r) \rfloor - 1$ .

In section §2.4.2, the ate pairing is described. Listing 2.3 shows the number of Miller loops iterations is related to  $t$ ; its length is  $\lfloor \log_2(t - 1) \rfloor - 1$ .

---

<sup>4</sup> $t - 1$  in this case

Miller-length in iterations		
Tate	$e_r(\cdot, \cdot)$	376
ate	$e_T(\cdot, \cdot)$	253
R-ate	$e_{A,B}(\cdot, \cdot)$	61
Lattice	$e_{s,h}(\cdot, \cdot)$	63

Table 2.5: Comparison of Miller-loop length for a KSS: $k = 18$  curve, with  $\text{Log}_2 x \approx 63$ 

The R-ate pairing, described in §2.4.3, requires three Miller loops with lengths depending on the  $A$  and  $B$  parameters. Its length is  $\lfloor \log_2(m_2) \rfloor + \lfloor \log_2(c) \rfloor + \lfloor \log_2(d) \rfloor - 3$ , where  $m_1, m_2, c$  and  $d$  are as given in §2.4.3.1. Simplifying for the chosen parameters, the length is  $\lfloor \log_2(m_2) \rfloor - 1 + \lfloor \log_2(c) \rfloor - 1$ .

The optimal pairing, described in the previous section, requires up to  $\varphi(k)$  Miller loops and up to  $\varphi(k) - 1$  line functions, which requires up to  $\varphi(k)$  scalar-point multiplications and up to  $\varphi(k) - 2$  point additions, depending on the vector used for constructing the pairing.

For a security level approximately of the strength of AES-192 using the KSS: $k = 18$  curves, the Miller loop length for these pairings is shown in Table 2.5. Clearly, the Miller loop length of the R-ate pairing is one sixth of the Tate pairing Miller loop length. The R-ate pairing requires fewer iterations of the Miller loop than the pairing lattice, but may require more point additions. However, the pairing lattice for this particular case requires a scalar-point multiplication of  $x$  and a point in  $G_2$ .

The cost of the Rate pairing for this family of curves is quite close to that of the pairing lattice, which is the optimal pairing. Certain values of the parameter of the curve would make one pairing function or the other the fastest, but only by a marginal number of operations. This varies a bit on the chosen  $x$ -parameter of the curve.

## 2.5 Pairing-Friendly Elliptic Curves

Let  $t(x)$  be the *trace of Frobenius*,  $p(x)$  the *field size* and  $r(x)$  the *pairing-friendly subgroup size*. As is well known, the number of points on the elliptic curve  $E(\mathbb{F}_p)$  is  $\#E = p + 1 - t$ .

**Definition 20.** The  $\rho$ -value is the ratio of the size of the field to the size of the subgroup

$$\rho = \frac{\deg(p(x))}{\deg(r(x))}.$$

For example, an elliptic curve with a 256-bit subgroup<sup>5</sup> and  $\rho = 1$  is defined over a 256-bit field, while a curve with a  $\rho = 2$  is defined over a 512-bit field. In practice, since the  $\rho$ -value is an approximate number, we may prefer to round the field size to a number of bits which is easier to implement.

**Definition 21.** *A family of pairing-friendly elliptic curves is a set of elliptic curves, each of which is given by a function or parametrisation of the field and subgroup size, the trace of Frobenius, and sometimes other values characterizing the elliptic curve. The term pairing-friendly refers to its small embedding degree and large subgroup order [FST10].*

There are many constructions for pairing-friendly elliptic curves. We refer to Freeman, Scott and Teske [FST10] for more information about these type of curves. In the following sections, we present a few particular pairing-friendly elliptic curves used in this thesis.

Freeman, Scott and Teske in [Ibidem] suggest that for choosing the adequate security level for a family of pairing friendly elliptic curves, we have to match  $\log_2(p^k)/\log_2(r) = \rho.k$ . Look up the corresponding values in Table 2.3 on page 13.

### 2.5.1 MNT Curves

The MNT curves were reported by Miyaji, Nakabayashi and Takano in [MNT01]. For the  $k = 6$  case, the curve has  $\rho = 1$ , and the parameters of the curve are defined as follows:

$$\begin{aligned} p(x) &= x^2 + 1; \\ r(x) &= x^2 - x + 1; \\ t(x) &= x + 1. \end{aligned} \tag{2.11}$$

This curve needs the CM method to construct the elliptic curve equation.

---

<sup>5</sup>which is adequate for a security level of 128-bits, see §2.1.4.1

### 2.5.2 Freeman Curves

Freeman in [Fre06] discovered a family of curves with embedding degree  $k = 10$  and  $\rho = 1$ . The construction uses the factorization of  $\Phi_{10}(10x^2 + 5x + 2)$ , discovered by Galbraith, McKee and Valena [GMV07]. This is a sparse family of curves and it again requires the CM construction method. The parameters of the curve are defined as follows:

$$\begin{aligned} p(x) &= 25x^4 + 25x^3 + 25x^2 + 10x + 3; \\ r(x) &= 25x^4 + 25x^3 + 15x^2 + 5x + 1; \\ t(x) &= 10x^2 + 5x + 3. \end{aligned} \tag{2.12}$$

### 2.5.3 Barreto-Naehrig (BN) Curves

Barreto and Naehrig in [BN06] presented a family of curves with embedding degree  $k = 12$  and  $\rho = 1$ , with small discriminant  $D = -3$  and plenty of curves. The family present the following parametrisation:

$$\begin{aligned} p(x) &= 36x^4 + 36x^3 + 24x^2 + 6x + 1; \\ r(x) &= 36x^4 + 36x^3 + 18x^2 + 6x + 1; \\ t(x) &= 6x^2 + 1. \end{aligned} \tag{2.13}$$

These curves support a sextic twist.

### 2.5.4 Kachisa-Schaefer-Scott Curves

Kachisa, Schaefer and Scott in [KSS08] proposed families of pairing-friendly elliptic curves of embedding degrees  $k = 16, 18, 32, 36$ , and  $40$ . The main idea in the construction is to use minimal polynomials of the elements of the cyclotomic field other than the cyclotomic polynomial  $\Phi_l(x)$  to define the cyclotomic field  $\mathbb{Q}(\zeta_l)$ . Interestingly, all these families of curves admit high order twists.

#### 2.5.4.1 KSS curves with $k = 8$

The family of curves for  $k = 8$ , with CM discriminant  $-1$ , is parametrised by the authors as follows:

$$\begin{aligned} p(x) &= (x^6 + 2x^5 - 3x^4 + 8x^3 - 15x^2 - 82x + 125)/180; \\ r(x) &= (x^4 - 8x^2 + 25)/450; \\ t(x) &= (2x^3 - 11x + 15)/15; \\ x &\equiv 5 \pmod{30} \end{aligned} \tag{2.14}$$

These curves support a quartic twist.

#### 2.5.4.2 KSS curves with $k = 18$

The family of curves for  $k = 18$ , with CM discriminant  $-3$ , is parametrised by the authors as follows:

$$\begin{aligned} p(x) &= (x^8 + 5x^7 + 7x^6 + 37x^5 + 188x^4 + 259x^3 + 343x^2 + 1763x + 2401)/21; \\ r(x) &= (x^6 + 37x^3 + 343)/343; \\ t(x) &= (x^4 + 16x + 7)/7; \\ x &\equiv 14 \pmod{42}. \end{aligned} \tag{2.15}$$

These curves support a sextic twist.



### 2.5.4.3 KSS curves with $k = 36$

The family of curves for  $k = 36$ , with CM discriminant  $-3$ , is parametrised by the authors as follows:

$$\begin{aligned}
 p(x) &= (x^{14} - 4x^{13} + 7x^{12} + 683x^8 - 2510x^7 + 4781x^6 + 117649x^2 \\
 &\quad - 386569x + 823543)/28749 \\
 r(x) &= (x^{12} + 683x^6 + 117649)/161061481 \\
 t(x) &= (2x^7 + 757x + 259)/259 \\
 x &\equiv 287 \pmod{777}
 \end{aligned} \tag{2.16}$$

These curves support a sextic twist.

## 2.6 Matrices

### 2.6.1 Greatest Common Divisor

A greatest common *right* divisor (gcd) of two matrices  $\{N(s), D(s)\}$  with the same number of columns is any matrix  $R(s)$  with the following properties [Kai80, pp. 376]:

1.  $R(s)$  is a right divisor of  $N(s)$  and  $D(s)$ ; i.e., there exist polynomial matrices  $\bar{N}(s)$ ,  $\bar{D}(s)$  such that:

$$N(s) = \bar{N}(s)R(s), \quad D(s) = \bar{D}(s).$$

2. If  $R_1(s)$  is any other right divisor of  $N(s)$  and  $D(s)$ , then  $R_1(s)$  is a right divisor of  $R(s)$ ; i.e., there exists a polynomial matrix  $W(s)$  such that  $R(s) = W(s)R_1(s)$ .

### 2.6.2 Echelon Form

An Echelon in this context, is the military formation that resembles the migration paths of birds, which is used as an analogy in the notation of matrices [Wil02].

An  $m \times n$ - matrix  $A = (a_{ij})$  is said to be in echelon form if there exist integers  $r \leq 0$ ,  $q \leq j_1 < j_2 < \dots < j_r \leq n$  such that  $a_{ij} = 0$  for  $i > r$  or  $(i \leq r \text{ and } j < j_i)$  and  $a_{ij} \neq 0$  for  $i \leq r$ , where the rank of  $A$  is  $r$  [BCS97, pp. 431].

In other words, a matrix is in *row echelon form* if:

1. All non-zero rows are above any all-zero rows;
2. The leading coefficient of a non-zero row is 1 and is strictly one column to the right of the previous row. This coefficient is called the *pivot*.

A matrix is in *reduced row echelon form* if additionally, the pivot is 1 and is the only non-zero element in the column.

Similarly, a matrix  $A$  is in *column echelon form* if its transpose  $A^T$  is in row echelon form [KMM04]. See example 3.21 from [KMM04, p.108].

## 2.7 Programming Languages

For the purposes of this research, the programming language used will be Magma, unless stated. All of the mathematical tests were run using Magma. When generating code, the output code is the MIRACL library. In some cases, code for RELIC is also generated. Some external libraries were used, but are referenced. In any case, when the code is not in Magma, is written in C/C++. The only exemption is in the Appendix C.

The compiler described in Chapter 6 is written entirely in Magma. As it will be shown later in the cited chapter, it will use some templates written in other programming languages than Magma, the programming languages used depend on the multi-precision library used to generate the code.

### 2.7.1 Magma

Magma is a computer algebra system designed to provide a software environment for computing with the structures which arise in areas such as algebra, number theory, algebraic

geometry and (algebraic) combinatorics [BCP97]. Magma is also the programming language to use such structures.

At the start of this research, there were other similar systems. Magma was suitable for the purposes of this research as it was able to handle finite field arithmetic of cryptographic size. In other words, Magma was able to handle operations with very large numbers, large enough for a real application.

Magma has state-of-the-art implementation of arithmetic operations, making it faster than other similar systems. Magma is quite big for a real applications, also it is slower than an implementation using a professional C/C++ implementation of the required arithmetic, however, Magma has the flexibility of fast development and tests. Additionally, since the learning curve was smooth, it was a suitable option for this research.

### 2.7.2 MIRACL

MIRACL is a multi-precision arithmetic library written in C/C++, with support for large integers, and several routines. It is portable, and also includes support for assembler arithmetic. It is a well established library used for research [Sco].

This library currently supports finite field and elliptic curve arithmetic for several families of elliptic curves, which can be used for some of the security levels.

The implementations of pairing functions are highly optimised and are used not only as a model to follow in this research, but also for benchmarking purposes.

Additionally, the implemented functions in this library are also implemented on current research in the area, making a suitable option for comparison.

### 2.7.3 RELIC

RELIC is another multi-precision arithmetic library, it is written in C, also with support for large integers and several routines. It does not have assembler routines, and it is at early stages of development, but it has a growing user base and it is a promising library, it is also suitable for small devices [AP].

This library currently supports the BN curves [BN06]; i.e., the finite field and elliptic curve arithmetic for this family is present. This is the only curve supported at the start of this research, but work for others is on the way.

## 2.8 Metaprogramming

Metaprogramming is a programming paradigm which manipulates computer programs. These programs can be either external or the program itself. The program data can be treated as code and *vice versa*. This concept is old and can be traced back as to the classical *modus operandi* of a system compromise using a buffer overflow.

A buffer overflow attack is an example of program data being used as code. This technique can be used to change the behavior of a computer system either locally or remotely. In an attack, the program is forced to overflow its data stack (where the program's data is stored) typically overwriting some carefully chosen sections of the stack. Modifying the data stack with a buffer overflow will give mixed results depending on the operating environment; a careful construction will lead to interesting behaviour, such as data code execution. In the end, the program can either crash or end up as a breach of the system security. To avoid this, many techniques have been developed, including the following: Monitoring the program, setting up limits on the program code size, and attribute-based protection. This can be achieved via hardware or software and it is always a relevant topic in computer science.

In contrast to a simple buffer overflow, Metaprogramming is a method to produce code for constructive purposes by modifying program code (regardless of the intention of the code). In this research we are focusing on a constructive approach.

The expectations on a code generator are usually high [ASU86], it must produce valid code, it should make use of very few resources, and the output should require little or no modifications by the end user. From a mathematical point of view, it is undecidable if the generated code is optimal [AF96]. Hence, we rely on heuristic techniques. It is this choice of techniques which should determine the quality of the code, rather than the construction

techniques for building it up.

Traditionally, a code generator outputs machine code or assembly code from intermediate code [Muc97]. Sometimes, the code generators output to several programming languages and a single computer architecture. Sometimes they target a single programming language and several architectures.

### 2.8.1 Automatic Code Generation

Automatic code generation is a very simple mode of code generation. There are two methods of code generation: One is to generate the code straight away; the other is to create an intermediate code sequence where one can choose segments depending on which is best suited for our purposes. In practice, code generators use a mix of these approaches [PP82].

Traditionally, the main focus in code generation is the Turing Machine capability. We need to know if the output code is suitable for a Turing Machine<sup>6</sup> (Von Neumann architecture). This conceptual machine, of course, is only used as a reference. In a more contemporary context, the main feature for a code generator is satisfying the customer's requirements, for speed, code size, memory footprint and others.

The optimal sequence of code for a particular algorithm may not be the obvious one. There are instances where the definition of optimal is not speed, but code size. This is difficult to achieve by hand for long code sequences, and an automatic code generator would be a better option. For example, generated code may have a big footprint and may not fit within the space requirements. If so, another branch in the code sequence, perhaps slower, can be generated instead.

#### 2.8.1.1 L- and R-value

To handle the instruction construction, we recall notation from Barron et al. [BBH<sup>+</sup>63]:

---

<sup>6</sup>A Turing Machine is an abstract model of a computational device. This model was devised by Alan Turing in 1937. The Turing Machine basically consists of a read and write device over a tape that can be randomly positioned. Reading one single character by position is actually used as a conceptual machine to understand the physical limits of mechanical computers.

**Definition 22.** *The Left Hand (LH) operator, also referred as an L-value, is an expression that when evaluated as an LH, produces a memory address.*

**Definition 23.** *The Right Hand (RH) operator, also referred as an R-value, is an expression that when evaluated as an RH produces a value.*

An LH operator is used to set the location (memory address) where the result of the defined operation is going to be stored. An RH operator is used to obtain the location of the data to be transformed (the value).

**Definition 24.** *The number of operands an operator can take is called its arity.*

We classify an operator in terms of its arity:

- Nullary: An operator that does not take parameters; e.g.: functions with global parameters, such as initialization code.
- Unary: An operator that takes one single operand; e.g.:  $-Q$ ,  $Q.\text{normalize}()$ .
- Binary: An operator that takes two operands; e.g.:  $a + b$ ,  $x^3$ .
- Ternary: An operator that takes three operands; e.g.:  $(a > b)?c : d$ . (operand 1:  $(a > b)$ , operand 2:  $c$ , operand 3:  $d$ )
- N-ary: This type is usually referred to as a function with one composed parameter, e.g. an N-tuple:  $M(a, \dots, f)$ .

### 2.8.1.2 Suffix Notation

The next issue to address in the instruction construction is the position of the operands. There are several types of notation and the multi-precision libraries may be influenced by the programming language of the library.

Types of notation:

- Polish notation [Chu96], also known as Prefix notation. This is a parentheses-free notation. Examples:

- MOV ah,09, which translates into:  $ah = 09$ ,
- C C p A q r N p, which translates into:  $((p \supset (q \vee r)) \supset \sim p)$ . There is no need for parenthesis.
- Reverse Polish notation, also known as Suffix notation. This is a parentheses-free notation.<sup>7</sup> Example:
  - a b ADD, which translates into:  $a + b$ .
- Infix notation. Example:
  - $3 + 4$ .
- Circum-fix notation. Example:
  - $\text{pow}(x,3)$ , which translates into:  $x^3$ .

Extremely rare cases will require special notation or ordering of parameters. An example of a special case would be defining a scalar-point multiplication on an elliptic curve with the reversed parameters: point then scalar. This, however, is on a case-by-case situation.

In [DD01] Deitel and Deitel mentioned that humans generally use *infix* notation whereas computers prefer *postfix* notation. This is not necessary the case for a high-level computer programming language, however.

### 2.8.1.3 Operator Overloading

In C++ and other languages, there is a type of polymorphism: Operator overloading. The arithmetic operands are defined over simple data-types, such as the integers, float, or bytes; however, there are no definitions for user-defined data objects (datatypes). It is possible to define the operation associated with an operator with non-standard parameters. This is particularly useful in ECC, as one can associate the “+” operator with the point addition operation, where the operands are points in an elliptic curve. In the particular case of C++ one can only work with the standard operators.

---

<sup>7</sup>Used mainly with LIFO (Last In, First Out) structures.

Operator overloading is a computer language feature that is not present in multi-precision libraries based on C, as the computer language does not support it. Hence, a more elaborate approach is needed. In some circumstances, the use of explicit temporary storage is required for every operator, but this is a specific circumstance.

This is important as affects the code construction. When there is no support for overloading operators, every non-standard arithmetic operation must be computed as a function call<sup>8</sup>. That is, instead of using the  $*$  symbol for a multiplication, we have to use a function such as `multiply()`. Also, unless the so-called `multiply` function supports several operands, each multiplication must be explicitly accumulated.

For example, if we need to compute  $f_1 \leftarrow f_{m2}^c \cdot f_d$  using operation overload, we can generate code as: `f1=fm2^c*fd;`, whereas without operator overload, we need to split this into two operations: `exponentiate(f1, fm2, c); multiply(f1, f1, fd);`.

## 2.8.2 Attribute-Oriented Programming

Direct output of text strings for code generation works adequately for simple syntax directed code, as needed for some cryptographic functions, such as arithmetic and “mapping operators”. Another method for code generation is Attribute-Oriented Programming, also known as Attribute Code Generation. This method use “tags”, labels around the code, which are instructions in a meta language for the compiler. It is then necessary to define a meta language to instruct the compiler.

For this method to be implemented, we need a partially finished program or a program with pseudo-code chunks. This has the potential to make program debugging difficult until all the parts are finished and it is successfully rebuilt with the compiler.

The parts of the code that need to be constructed will be tagged within the code and they will be replaced. This way, one can define the variable and place a tag for a future assignment. For example, we can define a program to test the bilinearity of a pairing function, but at coding time we do not know the value of the  $x$  parameter<sup>9</sup>. The compiler may generate

<sup>8</sup>See circum-fix in section 2.8.1.2

<sup>9</sup>Actually, we may not even know the elliptic curve.



its value and replace it with an assignment.

This method is commonly used in languages with capabilities for preprocessors directives, such as C/C++. This technique is also known as “macros” in other environments. This code generation method is an offline technique, but can be used online as will be shown in the next section.

Consider the following example:

$$X.set((BIG)ALPHA, (BIG)(BETA)); \quad (2.17)$$

This is a piece of code generated for the MIRACL library [Sco]. Here,  $X$  is an element of a field. ALPHA represents the real part of an element, and BETA represents the imaginary part. BIG represents the data type of a multi-precision number. Here, however, we are using the assign function “set” and because we know this is for the MIRACL library, we do not actually need to place the “BIG” tag, as we already know the code is for MIRACL.

For example purposes, we present the following piece of code:

Listing 2.8: Code in Magma for the  $\psi$  function via lifting

---

```
// Lift ECnD -> ECnK
// Apply the Frobenius
X=Frobenius(X,F,i);
Y=Frobenius(Y,F,i);
// Unlift ECnK -> ECnD
```

---

In Listing 2.8 we left a couple of tags, *Lift*, and *Unlift*, where the compiler should overwrite and place the corresponding generated code. In this example, the compiler will generate code for moving a point from the twist of an elliptic curve, to the curve over an extension field, or vice versa.

### 2.8.3 Reflective Programming

Producing machine code has the obvious advantage that we can place it directly into the computer memory for immediate use. In the case of interpreted languages, it is possible to load a file into the memory for interpretation.

Integrating recently generated code into a running program is one of the main attractions of reflective programming. A program that can embed code resulting from its own production into itself is a *reflective program*. Not all programming languages provide tools to integrate code at execution time. Some languages provide tools to load external libraries at run time.

Reflection in computer science means that a program can observe and modify its own structure and behaviour. Introduced by Smith in [Smi82] as a method for a program to “reason” its internal functions, Reflection, has been extended to the object-oriented paradigm as *reflective-oriented programming* to implement polymorphism. *Polymorphism* is a capability in the object-oriented programming paradigm that enables the writing of programs in a more general approach, setting up interfaces not yet defined. This is a key technique for handling complex software programs [DD01].

Another example of reflective programming is the work of Van den Bussche et al. [VVV93], where the authors data-typed some algebraic structures into a database and optimized them at runtime.

#### 2.8.4 Template-based Metaprogramming

Another metaprogramming technique we would like to cover is *template-based* metaprogramming. With this method, one creates generic interfaces for the functional part of the code.

An object will have a set of methods or actions it can take. At some stage, the name of the method that an object is capable of executing is unknown to the programmer. To illustrate, consider the following example: A penguin cannot fly, a parrot can talk, and a roadrunner prefers to run<sup>10</sup>; however, all are birds. A programmer, can create a common interface for a general description of a bird. Using class/object inheritance, the programmer, can add specific behaviour to the birds: speak, run, don’t fly, etc. A proxy class is used where generic names of the methods are set up to access the main features of the final

---

<sup>10</sup>The roadrunner can actually fly to escape from predators, but it prefers to sprint.

object, as in the previous example. This would be an *online* template-based metaprogram.

An *offline* template-based metaprogram would be something similar to the classical C++ templates, the one used to define the operations, without defining the data type. At compile time, the compiler fills in the blanks and one runs the code with the desired data-types.

We are not using online template-based metaprogramming methods in the current research; however, we use labels around the code to setup an undefined data type to be used in the code as a way to implement offline templates. For example, some parts of the pairing function behave the same way regardless of the finite field used. Depending on the elliptic curve, the finite fields are chosen, but some parts of the code are basically the same.

## 2.9 Exponentiation in $G_{2,T}$

In [GLV01], Gallant, Lambert and Vanstone introduced a method to speed up general point multiplication  $nP \in E(\mathbb{F}_p)[r]$  when there is an efficient computable endomorphism  $\psi$  on  $E$  defined over  $\mathbb{F}_p$  such that  $\psi(P) = \lambda P$

The idea is to compute  $nP$  efficiently by writing  $n \equiv n_0 + n_1\lambda \pmod{r}$  with  $|n_i| < \sqrt{r}$  and performing a double exponentiation  $n_0P + n_1\psi(P)$ . In this case, the number of bits in  $n_0$  and  $n_1$  is half the bit-length of  $n$ . One can save a significant number of point doublings at the expense of a few point additions and the application of a map. If the map  $\psi$  is also cheaper than a point addition then, it is possible to get computational savings.

Gallant et al. [GLV01] pointed out that their method can be generalized to use higher powers of the endomorphism. We applied their method to the KSS curves with embedding degree  $k = 18$  with positive results.

Galbraith and Scott [GS08] recently showed a technique for generalizing the GLV method for higher powers of the endomorphism for the groups  $G_2$  and  $G_T$ . To get an  $m$ -dimensional expansion  $n \equiv n_0 + n_1\lambda + \dots + n_{m-1}\lambda^{m-1} \pmod{r}$  of  $nP$ , one must compose  $n$  with powers of  $\lambda$  sufficiently different and modulo  $r$ . This can be done by solving a closest vector problem in a lattice as done in Babai's rounding method [Bab86]. An LLL-reduced lattice basis, however, must be precomputed [GS08].

**KSS  $k = 18$  Curves**

For KSS  $k = 18$  curves [KSS08], it is possible to get a “natural” 6-dimensional expansion since the  $\varphi(k) = 6$ .

The modular lattice basis is defined as, by:

$$L = \left\{ x \in \mathbb{Z}^m : \sum_{i=0}^{m-1} x_i \lambda^i \equiv 0 \pmod{r} \right\}$$

where  $\lambda = T = t - 1$  as in [GS08, Example 5]. This 6-dimensional modular lattice  $L$  will be used to construct a  $6 \times 6$  matrix. Then, one can fill the matrix with any combination with  $\lambda$  that gives  $L_{i,j} \equiv 0 \pmod{r}$ . One can use a predefined sequence of values for the  $L$ -matrix to be LLL-reduced, however it is possible to use random values. The Magma program at Listing 2.9 gives both a random and an ordered construction of the matrix  $L$ . The function “`LLL()`” at the end of the program computes the LLL-reduced lattice bases  $B$ , as in [GS08, Example 5].

Listing 2.9: Initial Lattice and LLL-reduction

---

```
x:=0x164458D77281230F6;
m:=6;
r:=(x^6 + 37*x^3 + 343) div 343;
t:=(x^4+16*x)/7+1;
lambda:=t-1;

// Option 1, randomly
L:= Matrix(Rationals(), m, [
    lambda,-1,0,0,lambda,-1,
    0,0,lambda,-1,0,0,
    0,0,0,lambda,-1,0,
    0,lambda,-1,0,lambda,-1,
    lambda,-1,0,lambda,-1,0,
    r,lambda,-1,lambda,-1,0
]);

// Option 2, ordered
L:= Matrix(Rationals(), m, [
    r,0,0,0,0,0,
    lambda,-1,0,0,0,0,
    lambda^2,0,-1,0,0,0,
    lambda^3,0,0,-1,0,0,
    lambda^4,0,0,0,-1,0,
    lambda^5,0,0,0,0,-1
```

```

]);

B:=LLL(L);

```

Note that one can automate the creation of  $L$  in the second matrix using the code in Listing 2.9.

One can verify the lattice consistency mod  $r$  using the code in Listing 2.10

Listing 2.10: Validate matrix B

```

checklattice:=procedure(L)
  sum:=0;
  for i:=1 to m do
    for j:= 1 to m do
      sum:=sum+L[i][j]*lambda^(j-1);
    end for;
    sum:= Integers()!sum mod r;
    if sum eq 0 then print "OK"; else print "BAD",i; end if;
    sum:=0;
  end for;
end procedure;

```

At this point, there is a valid LLL-reduced  $B$  matrix.

$$B = \begin{pmatrix} -\frac{2x}{7} & -1 & 0 & -\frac{x}{7} & 0 & 0 \\ 0 & -\frac{2x}{7} & -1 & 0 & -\frac{x}{7} & 0 \\ 0 & 0 & \frac{2x}{7} & 1 & 0 & \frac{x}{7} \\ \frac{x}{7} & 0 & 0 & -\frac{3x}{7} & -1 & 0 \\ 0 & \frac{x}{7} & 0 & 0 & -\frac{3x}{7} & -1 \\ -1 & 0 & -\frac{x}{7} & 1 & 0 & \frac{3x}{7} \end{pmatrix}$$

Choose a random exponent  $n$  and compose a vector  $(n,0,0,0,0,0)$  with respect to the basis formed by  $B$ . With the  $\lambda$ -value generating the matrix  $B$  from Listing 2.9, one obtains a vector  $v \approx wB^{-1}$  as follows:

$$v \approx \left( -\frac{3x^5+56x^2}{343}, \frac{8x^4+147x}{343}, \frac{19x^3+343}{343}, \frac{x^5+3x^2}{343}, -\frac{5x^4+98x}{343}, -\frac{18x^3+343}{343} \right) \cdot \frac{n}{r}$$

Since all the elements of  $v$  are divided by  $r$  and  $\notin \mathbb{Z}$ , one just needs to *round* each element [Bab86],[GS08] of  $v$  before getting the vector  $u = w - vB$ , as is done by the code

in Listing 2.11.

Listing 2.11: Rounding and generating vector  $u$

---

```

n:= Random(r);
w:= Matrix(RationalField(), 1, m, [n,0,0,0,0,0]);

// Rounding
v0:= w*B^-1;
v:= Matrix(RationalField(), 1, m, [0,0,0,0,0,0]);
for i:= 1 to m do
    v[1][i]:= Round(v0[1][i]);
end for;

// Main vector
u:= w-v*B;

```

---

In the code of Listing 2.11 the vector  $u$  contains the coefficients  $n_i$  with the decomposition of  $n$  mentioned at the beginning of this section. Finally, one can verify

$$n \equiv n_0 + n_1\lambda + \cdots + n_{m-1}\lambda^{m-1} \pmod{r}$$

for this vector in Listing 2.12

Listing 2.12: Verification of the  $n_i$  components

---

```

sum:=0;
for i:= 1 to m do
    sum:= sum + u[1][i]*lambda^(i-1);
end for;
if (Integers()!sum mod r) eq n then print "OK" else print "BAD";

```

---

### 2.9.1 Weak Popov Representation

Galbraith and Scott in [GS08] presented a reduced  $v$  matrix for the BN curves case. Dominguez Perez, Kachisa and Scott in [DKS], presented the same matrix but for the KSS  $k = 18$  curves. They did not present a method to generate this matrix in reduced form, which can be easily stored and generated by our code generator.

The polynomial representation of the elements of the vector  $v$  can be constructed as follows: Let  $u$  be a vector of degree  $m$ , where  $m = \deg(r(x))$ . For every  $0 \leq j < m$  in

$v_j$  we remove the  $n$  random integer and the  $r$  components:  $v' = v \cdot r/n$ . We can easily construct a vector  $[k_{m,i} \cdot u^i + \dots + k_{m,0}, \dots, k_{0,i} \cdot u^i + \dots + k_{0,0}]$  (by decreasing  $i$  and increasing its coefficients until we have a polynomial representation of  $v_j$ .)

In addition to the vector  $v$ , the Galbraith and Scott method needs a matrix  $B$  containing a reduced basis of  $m$  random vectors congruent to zero mod  $r$ . For the reduction of this matrix into a polynomial form one can use simple factorization.

In [GS08], the  $B$  matrix for the BN curves is as follows:

$$B = \begin{pmatrix} x+1 & x & x & -2x \\ 2x+1 & -x & -(x+1) & -x \\ 2x & 2x+1 & 2x+1 & 2x+1 \\ x-1 & 4x+2 & -(2x-1) & x-1 \end{pmatrix} \quad (2.18)$$

Another construction method, as pointed out by Barreto [Bar] involves the use of the Weak Popov form of the matrix.

### 2.9.1.1 Popov Form

A polynomial matrix  $A(x) = (a_{ij})$  is in *Popov form* or in *polynomial-echelon form* if it has the following characteristics: [Kai80, p. 481]

1. It is column reduced, with its column degrees arranged in ascending order, say

$$k_0 \leq k_1 \leq \dots \leq k_{m-1} \quad (2.3.1)$$

2. For column  $j$ ,  $0 \leq j \leq m-1$ , there is a so-called *pivot index*  $p_j$  such that: (2.3.2)

- (a)  $a_{p_j,j}(x)$  has degree  $k_j$
- (b)  $a_{p_j,j}(x)$  is monic
- (c)  $a_{p_j,j}(x)$  is the last (or lowest) entry of degree  $k_j$  in the  $j$ -th column; that is,  $\deg a_{p_j,j}(x) < k_j$  if  $i > p_j$
- (d) if  $k_i = k_j$  and  $i < j$ , then  $p_i < p_j$ ; i.e., the pivot indices are arranged to be increasing

(e) the polynomial  $a_{p_j,j}(x)$  has degree less than  $k_j$  if  $i \neq j$ .

The following is a matrix in polynomial-echelon form [Kai80, Example 6.7-3, pp.483]:

$$A_e(x) = \begin{bmatrix} 5x+1 & x^2+3x+2 & 4x+5 \\ 3x+4 & 2x+1 & x^3+x^2+2 \\ x+7 & 3 & 2 \end{bmatrix}$$

We write  $A_e(x) = A_3 + A_2x + A_1x^2 + A_0x^3$ , and we form the *coefficient matrix*:

$$\begin{aligned} \mathcal{A}'_e &\triangleq [ A'_3 \quad A'_2 \quad A'_1 \quad A'_0 ] \\ &= \left[ \begin{array}{ccc|ccc|ccc|ccc} 1 & 4 & 7 & 5 & 3 & \textcircled{1} & 0 & 0 & 0 & 0 & 0 & 0 \\ 2 & 1 & 3 & 3 & 2 & 0 & \textcircled{1} & 0 & 0 & 0 & 0 & 0 \\ 5 & 2 & 2 & 4 & 0 & 0 & 0 & 1 & 0 & 0 & \textcircled{1} & 0 \end{array} \right] \end{aligned} \quad (2.19)$$

There is an additional constraint on these pivots (the circled elements in Equation 2.19): the number of elements in each row before the pivots is different; in other words, no pivot shares a column with another pivot. Also, every pivot is the only non-zero element in its respective column.

### 2.9.1.2 Quasi-Echelon Form

*Quasi-Echelon Form*, also known as *Quasi-Canonical Polynomial-Echelon Form*, is a relaxed construction of the polynomial echelon form.

For this form, we introduce the requirement that all the pivot indexes are distinct. Condition 2.3.2.e did not require this.

Consider the following matrix [Kai80, Example 6.7-4, p.484]:

$$\tilde{A}_e(x) = \begin{bmatrix} 5x+1 & x^2+3x+2 & 2x^3+3x^2+4x+5 \\ 3x+4 & 2x+1 & x^3+x^2+2 \\ x+7 & 3+4x & 3x^2+5x+2 \end{bmatrix}$$



This matrix has the following coefficient matrix construction:

$$\tilde{\mathcal{A}}'_e = \left[ \begin{array}{ccc|ccc|ccc} 1 & 4 & 7 & 5 & 3 & \textcircled{1} & 0 & 0 & 0 & 0 & 0 & 0 \\ 2 & 1 & 3 & 3 & 2 & 4 & \textcircled{1} & 0 & 0 & 0 & 0 & 0 \\ 5 & 2 & 2 & 4 & 0 & 5 & 3 & 1 & 3 & 2 & \textcircled{1} & 0 \end{array} \right] \quad (2.20)$$

We have lost the characteristic that every pivot is the only non-zero element of its respective column. However, if we restrict them to the unique pivot of the column, the rest of the conditions remain.

### 2.9.1.3 Weak Popov Form

In [MS03], Mulders and Stojohann proposed a new matrix construction: the *Weak Popov Form*. This construction presents more relaxed conditions than the Quasi-echelon form: The pivot is any non-zero element at the end of the vector, and a pivot is unique in the column.

The following is the coefficient matrix of matrix 2.18:

$$\tilde{\mathcal{B}}'_e = \left[ \begin{array}{cccc|cccc|c} 1 & 1 & 0 & -1 & 1 & 2 & 2 & \textcircled{1} & 0 \dots \\ 1 & -1 & 1 & 2 & 1 & -1 & 2 & \textcircled{4} & 0 \dots \\ 0 & -1 & 1 & 1 & 1 & -1 & 2 & \textcircled{-2} & 0 \dots \\ 0 & 0 & 1 & -1 & -2 & -1 & 2 & \textcircled{1} & 0 \dots \end{array} \right] \quad (2.21)$$

We can see that the coefficient matrix 2.21 is not in echelon form, as all its pivots are in the same column.

Mulders and Stojohann presented a few algorithms to transform a matrix into Weak Popov form. The implementation requires the Lemmas 2.1-2.5 from [MS03].

The algorithm to transform a matrix into Weak Popov form is given in Algorithm 3:

With this algorithm, we get an awkward denominator of 2. We can follow the same approach as in Chapters 4 and 5, or we can modify this algorithm to force the reduction

**Algorithm 3** Transformation into Weak Popov form**Input:**  $A \in \mathbb{Z}[x]^{n \times m}$ **Output:**  $A$  in Weak Popov form

---

```

while  $A$  not in Weak Popov form do
   $p \leftarrow$  the pivot indices.
  for all  $0 \leq i \leq n - 1$  do
     $D \leftarrow \deg A_{i,p_i}$ 
     $d \leftarrow$  leading coefficient of  $A_{i,p_i}$ 
    for all  $0 \leq j \leq n - 1$ , excluding  $i$  do
       $\diamond$  Ensure the pivot is the greatest element in the column
      if  $\deg A_{j,p_i} \geq D$  then
         $\diamond$  Reduce the row
        { $c$  the leading coefficient and  $e$  the degree difference}
         $A_j \leftarrow A_j - c.x^e.M_i$ 
        break k
      end if
    end for
  end for
end while

```

---

step to be in  $\mathbb{Z}$ , and achieve the following reduced  $B$  matrix for the BN curves case:

$$BN = \begin{pmatrix} 2x & x+1 & -x & x \\ x+1 & -(3x+1) & x-1 & 1 \\ -1 & 6x+2 & 2 & -1 \\ 6x+2 & 1 & -1 & 1 \end{pmatrix} \quad (2.22)$$

which has the following coefficient matrix:

$$\tilde{\mathcal{BN}}'_e = \left[ \begin{array}{cccc} 0 & 1 & -1 & 2 \\ 1 & -1 & 2 & 1 \\ 0 & -1 & 2 & -1 \\ 0 & 1 & -1 & 1 \end{array} \middle\| \begin{array}{cccc} 2 & 1 & 0 & \textcircled{6} \\ 1 & -3 & \textcircled{6} & 0 \\ -1 & \textcircled{1} & 0 & 0 \\ \textcircled{1} & 0 & 0 & 0 \end{array} \middle\| \begin{array}{c} 0 \cdots \\ 0 \cdots \\ 0 \cdots \\ 0 \cdots \end{array} \right]$$

which evidently is in Weak Popov form.

For the KSS: $k = 18$  curves case, we found previously in Matrix 2.10 on page 30 the

following:

$$M_{k=18} = \begin{pmatrix} 10/7 & 0 & 6x/7 & 6/7 & 0 & -2x/7 \\ 0 & 1/7 & 0 & x/7 & 2/7 & 0 \\ 0 & 2x/7 & 1 & 0 & x/7 & 0 \\ 1/2 & 0 & x/2 & 1 & 0 & 0 \\ -x & -3 & 0 & 0 & 1 & 0 \\ 0 & -x & -3 & 0 & 0 & 1 \end{pmatrix}$$

which was used to achieve the optimal pairing. This matrix, however, contains coefficients of degree zero in the rationals, which are not useful for this operation.

---

**Algorithm 4** Transformation into a safe coefficient form

---

**Input:** Matrix  $BB$  with some coefficients of degree zero in the rationals

**Output:** Corrected matrix with no coefficients of degree zero in the rationals

Let  $P$  be the lattice matrix  $BB$  of dimension  $\varphi(k) \times \varphi(k)$

Let  $N$  be a zero matrix of the same dimension as  $P$ .

Let  $N_{i,j} \leftarrow 1$  if  $P_{i,j}$  is a polynomial of degree zero with coefficients in the rationals for all  $1 \leq i, j < \varphi(k)$ .

◇ A) Get 2 elements of  $P$  in the same column to be fixed (degree zero, w/coeff. in rationals, use matrix  $N$ )

- Multiply and divide the whole row with the smallest denominator.

◇ B)  $cw \leftarrow P_{a,i} - P_{b,i}$ , where  $a$  is the row with the greater denominator,  $b$  the other, and  $i$  is the column where the “bad” coefficient is present.

- Increment the row with the smallest coefficient by  $cw * P_a$ .

Set  $N_{a,i}, N_{b,i} \leftarrow 0$

For an odd number of “bad” coefficients, pickup a polynomial of degree zero of the same column and proceed as in (B).

Repeat until  $N$  contains only zeroes.

---

The following is the reduced matrix in Weak Popov form:

$$K_{18} = \begin{pmatrix} 1 & 0 & 5x/7 & 1 & 0 & -x/7 \\ -5x/7 & -2 & 0 & x/7 & 1 & 0 \\ 0 & 2x/7 & 1 & 0 & x/7 & 0 \\ 1 & 0 & x & 2 & 0 & 0 \\ -x & -3 & 0 & 0 & 1 & 0 \\ 0 & -x & -3 & 0 & 0 & 1 \end{pmatrix} \quad (2.23)$$

In this family of curves, since  $x \equiv 14 \pmod{42}$ , we know that  $7|x$ , so we do not need to force the coefficients to be in  $\mathbb{Z}$ .

### 2.9.2 The Galbraith-Scott Method using the Weak Popov Form

We can use the Weak Popov Form with the Galbraith-Scott method.

For the BN curves, the vector  $\mathbf{v}$  generated from Matrix 2.22, which can be used for pre-computing is as follows:

$$\mathbf{v} \approx (-(6x^2 + 6x + 2), -1, x, 6x^3 + 6x^2 + 3x + 1) \cdot \frac{n}{r}.$$

The KSS curves with  $k = 18$  has the following vector  $\mathbf{v}$  from matrix 2.23:

$$\mathbf{v} \approx (2 + \frac{5x^3}{49}, -\frac{x^2}{49}, \frac{3x}{7} + \frac{x^4}{49}, -(1 + \frac{17x^3}{343}), -(\frac{2x^2}{49} + \frac{x^5}{343}), \frac{2x}{7} + \frac{5x^4}{343} \cdot \frac{n}{r}$$

This vectors  $\mathbf{v}$  can be trivially converted into polynomial form by selecting any  $v_i$ , and comparing it with a decreasing  $a_j \mathbf{v}^{b_j}$  until we have a polynomial representation. Algorithm 5 is presented for this conversion. This algorithm requires a “magical” number  $\text{ex}$  to setup a limit in the search of divisors of the polynomials. This number can be set from either the common denominator of  $r(x)$  of the curve, when present; and the last element of any of the addition chains<sup>11</sup>.

---

<sup>11</sup>In the case of some curves, this number can be very high, such as in the case of the KSS curves with  $k = 36$ .

**Algorithm 5** Transformation of vector  $v$  into polynomial form**Input:** Vector  $v$ , the parameter  $x$  of the curve,  $ex$  maximum coefficient to test**Output:** Vector  $v$  represented in base- $x$ For every element in  $v$ : $t$  vector of dimension  $\varphi(k)$ ,  $w = |v_i|$ **for**  $j \leftarrow \deg(r(x)) + 1$  **downto** 1 **do** $t_j \leftarrow 1, T \leftarrow \sum_{l=1}^{\#t} t_l \cdot 2^{l-1}$ **if**  $w > T$  **then****for**  $l \leftarrow 2$  **to**  $ex$  **do** $t_j \leftarrow l, T \leftarrow \sum_{l=1}^{\#t} t_l \cdot 2^{l-1}$ **if**  $w < T$  **then** $t_j \leftarrow -$ We found an inflection, next degree ( $j$ )**end if****if**  $w = T$  **then**

◊ Adjust sign accordingly

Construct polynomial  $v_i$  from vector  $t$ .Done, next  $v$ **end if****if**  $l = ex$  **then** $t_j \leftarrow 0, s \leftarrow t$ **for**  $a \leftarrow ex$  **downto** 1 **do****for**  $b \leftarrow ex$  **downto** 1 **do** $T \leftarrow \sum_{l=1}^{\#t} s_l \cdot 2^{l-1}$  $T+ = b \cdot (x^j) / a$ **if**  $T \notin \mathbb{Z}$  **then**

Non dividable, try next value

**end if****if**  $w \geq T$  **then** $Tdif \leftarrow w - T$ **if**  $Tdif < x^{j-1}$  **then** $s_j+ = b/a; t_{j+1}+ = s;$ We found an inflection, next  $l$ **end if****end if****end for****end for****end if****end for****end if****end for**

## 3 ADDITION CHAINS

*"When [s]he woke up, the dinosaur was still there." – The dinosaur, by Augusto Monterroso. In "Obras completas (y otros cuentos)", 1959.*

ONE of the hard parts of the pairing calculation that requires specific optimisation for the Tate and similar pairing functions is the so-called "final exponentiation", performed after the Miller loop operation. This exponentiation is to the power of  $(p^k - 1)/r$  in  $\mathbb{F}_{p^k}^*$ . It is clear that a short addition chain for an integer  $e$  gives a faster method for computing  $f^e \in \mathbb{F}_{p^k}^*$  than direct exponentiation. The operation can be simplified with the use of multiplication and squaring operations using addition chains. This multi-scalar multiplication makes the calculation faster by reusing intermediate values of the computation.

**Definition 25.** An Addition Chain for a given integer  $e$  is a sequence  $U = (u_0, u_1, u_2, \dots, u_l)$  such that  $u_0 = 1$ ,  $u_l = e$  and  $u_k = u_i + u_j$  for  $k \leq l$  and some  $i, j$  with  $0 \leq i \leq j$ .

Consider the following *Fibonacci* sequence:  $\{1, 2, 3, 5, 8, 13, 21\}$ . This is a addition chain for  $e = 21$  which contains 7 elements. Each element is obtained from the addition of the previous two elements.

An alternative chain is:  $\{1, 2, 4, 8, 16, 20, 21\}$ . In this case, the element  $e = 21$  can be constructed using 4 doubling operations and 2 addition operations, but has the same length.

Now, consider  $e = 34$ , the Fibonacci sequence grows by one element:  $\{1, 2, 3, 5, 8, 13, 21, 34\}$ . We can also construct the following addition chain to reach 34:  $\{1, 2, 4, 8, 16, 32, 34\}$ . This is a shorter chain and makes use of addition and doubling operations instead of only addition operations.

From the previous example, we can see that for small values of  $e$  it is trivial to find a short addition chain using an exhaustive search. As  $e$  grows, however, the difficulty of determining the length of the shortest possible addition chain grows considerably.

In this chapter, we introduce some of the relevant algorithms for finding a short addition chain. The user should be able to construct a short addition chain for a random exponent  $e$  either by implementing the methods given here in a computer program or manually.

This chapter is organized as follows: §3.1 contains definitions on addition chains. In §3.2 is presented the basic method for finding an addition chain. In §3.3 is presented the Bernstein method. In §3.4 the concept of Artificial Immune System is introduced. In §3.5 is presented our construction, which is based on Artificial Immune Systems. In §3.6 is presented a comparison between the Binary and the AIS methods for solving a long sequence. Finally, in §3.7 we present conclusions and final thoughts on this part of the research.

### 3.1 Introduction

Determining if we have found the shortest addition chain for a given positive integer is an NP-complete problem [DLS81]. It is clear the shorter the addition chain, the lesser the number of operations required. There are special cases where the shortest addition chain does not necessarily give the most speed up. For example, this might be the case where one can exchange slower operations for a few extra faster ones (i.e. exchange an addition for a few doubling operations, if they are faster). This idea will be explored in Chapter 4.

The previous definition of an addition chain is useful mainly for cryptography based on the RSA method. To extend the use of addition chains to the present work, we introduce the following definitions.

**Definition 26.** *A Vector Addition Chain is the shortest possible list of vectors where each vector is the addition of two previous vectors. The last vector contains the final exponent  $e$ .*

Let  $V$  be a vector chain, and  $m$  be the dimension of every vector. The vector addition chain starts with  $V_{i,i} = 1$  for  $i = 0 \dots m-1$ :  $[1, 0, 0, \dots, 0], [0, 1, 0, \dots, 0], \dots, [0, \dots, 0, 1]$ ; we then proceed adding any two previous vectors to form a new vector in the chain, and continue until  $V_{j,1} = e$  with  $j$  typically  $> m$ .

**Definition 27.** *Given a list of integers  $\Gamma = \{v_1, \dots, v_l\}$  where  $v_l \geq v_i$  for all  $i = 1, \dots, l-1$ ,*

*an Addition Sequence for  $\Gamma$  is an addition chain for  $v_l$  containing all elements of  $\Gamma$ . The last element of the sequence is the exponent  $e = v_l$ .*

Addition sequences, otherwise known as *multi-addition-chains*, are used to speed up the final exponentiation and for fast hashing to a point in  $G_2$ . To use these implementation improvements it is necessary to have code to generate the multi-addition chains for a given list of integers.

Different methods exist to construct addition chains. Bos and Coster [BC89] presented a set of algorithms to construct addition chains. Bernstein presented in [Ber09] a method for multi-scalar multiplication which constructs short addition sequences without the use of the Bos and Coster heuristic methods. Cruz-Cortez et al. [CCRHC08] presented a new approach to find short addition chains using Artificial Immune Systems which Dominguez Perez and Scott [DS09] extended to addition sequences. The following sections present the above mentioned methods.

We refer to the set of integers which we wish to incorporate into an addition sequence as a “proto-sequence”. Some of its elements cannot be constructed by the addition of any other member of the set.

For example, the set containing the elements we need to include in the addition sequence, but only them, is a proto-sequence.

## 3.2 Bos and Coster Method

Bos and Coster, in [BC89], suggested that an addition chain computation for an RSA exponent has to be fast to be useful, as one needs a different chain every time. In Pairing-Based Cryptography, this is not always true; as we will see in Chapters 4 and 5, we can generate an addition chain for certain purposes and reuse it several times.

In their paper, Bos and Coster propose a “Makesequence” algorithm. This algorithm starts with a proto-sequence 1, 2 and  $e$ , which we complete with at least one of the following methods:



- Approximation
- Division
- Halving
- Lucas.

**Approximation:** This method computes the difference between the target element and any two smaller elements already in the chain, adding the difference to the greatest of the smaller elements to include it in the chain.

**Division:** inserts into the addition chain the set formed by  $\lfloor e/S_i \rfloor$  for all of the elements in  $S$ , where  $S$  is a set of arbitrary length  $l$  containing the first  $l$  prime numbers.

**Halving:** inserts into the chain the set formed by  $\{\lfloor L_i/a \rfloor\}$  for every element in  $L$ , where  $a = e - s$ ,  $e$  is the target element,  $s$  is a random small element in the chain, and  $L$  is a set of arbitrary length containing small even integers.

**Lucas:** include a Lucas sequence containing the target element  $e$  as the last element in the series.

Once the target  $e$  is reached, we must proceed with every other single element in the chain.

### 3.3 Binary Method

Another similar approach to the Bos and Coster method is to subtract elements from  $e$ . At SPEED-CC 2009, Bernstein [Ber09] presented a method for optimizing linear maps modulo 2, which incidentally, can be used to find a short addition chain. This is an example of the binary method.

Instead of using subtractions it uses XORs on the binary representation of the elements in the chain. The algorithm applies an in-place XOR with the two largest values in the

chain, and repeating the operation until all of the elements are zero. The addition chain is all of the values from the original chain plus the results of the XORs.

If the position of the most significant bit is not the same in both of the two largest elements, then we cannot reduce the element with an XOR. In this case, we have to use integer subtraction.

### 3.3.1 Construction of the Binary Method

In the Magma language there are no implementation of the binary XOR operation. However, one is able to obtain the binary representation of the coefficients in the addition chain with the use of the `Seqint` and `Intseq` functions for converting a sequence of “0”s and “1”s into its integer representation and vice-versa.

Listing 3.1: Addition chain construction with the Binary Method

---

```
// Input  <- Ochain: proto-sequence to accomplish.
// Output -> Fchain: completed chain.
Reduce:=procedure(~Bchain,~a,~b,~Fchain)
  a,b:=GetHighest(Bchain); //a,b;
  if IsXORable(Bchain,a,b) then
    Bchain[a]:=BitXOR(Bchain,a,b);
  else
    tInteger:=Seqint(Bchain[a],2);
    tInteger-:=Seqint(Bchain[b],2);
    Bchain[a]:=Intseq(tInteger,2);
  end if;
  Include(~Fchain,Seqint(Bchain[a],2));
end procedure;

Bchain:=ToBitSeq(Ochain);
Fchain:=Ochain;
a:=0; b:=0;

while IsAllZero(Bchain) eq false do
  Reduce(~Bchain,~a,~b,~Fchain);
end while;
Exclude(~Fchain,0);
Sort(~Fchain);
Fchain;
```

---

Listing 3.1 takes a proto-sequence and reduces it until all of the elements are 0. The functions `IsXORable` and `GetHighest`, can be trivially coded by the interested reader.

0	0	0	0	0	0	0	1
0	0	0	0	0	0	1	0
0	0	0	0	0	1	0	0
0	0	0	0	0	1	0	1
0	0	0	0	0	1	1	1
0	0	0	0	1	0	1	0
0	0	0	0	1	1	1	1
0	0	0	1	1	0	1	0
0	0	0	1	1	1	0	0
0	0	0	1	1	1	1	0
0	0	1	1	0	1	1	1
0	1	0	0	1	0	1	1
0	1	0	1	0	0	0	0
0	1	1	0	0	1	0	0
1	0	0	1	0	0	0	0

Table 3.1: Input Matrix for the Binary Method.

The IsXORable function determines whether we can proceed with an XOR or with an integer subtraction in the current state of the algorithm.

The GetHighest function returns the index position in the chain of the two greatest elements. The highest is the first parameter returned.

### 3.3.2 Example using the Binary Method

Consider the following proto-sequence, which will also be used in Chapter 4:

$$\{1, 2, 4, 5, 7, 10, 15, 26, 28, 30, 55, 75, 80, 100, 108, 144\}.$$

A toy example using an implementation in Magma of the algorithm described in [Ber09] was compared against the source code provided by the author to test its validity. The algorithm requires the matrix in Table 3.1<sup>1</sup>. Each row represents an element of the addition chain in binary.

Since the last row in 3.1 is equal to 144 and the second last is 108, the most significant bits are not both 1. We need to apply an integer subtraction, which in this case, replaces the

<sup>1</sup>The most significant bit is on the left.

last line with 36 (0, 0, 1, 0, 0, 1). Now, we focus our attention in 108 and 100 (second and third last rows respectively), which are the new highest values. An XOR results in 8, which replaces the second to last line; then we continue with the rest. We can sort the table after every XOR as suggested by the author.

The addition chain generated using this algorithm is:

$$\{1, 2, \underline{3}, 4, 5, \underline{6}, 7, \underline{8}, 10, \underline{11}, \underline{12}, 15, 26, \underline{27}, 28, 30, \underline{36}, 55, \underline{63}, 75, 80, 100, 108, 144\},$$

which is of length 24 and requires 5 doubling and 19 addition operations.

The addition chain presented at Scott et al. [SBC<sup>+</sup>09b] is:

$$\{1, 2, 4, 5, 7, 10, 15, \underline{25}, 26, 28, 30, \underline{36}, \underline{50}, 55, 75, 80, 100, 108, 144\}.$$

This chain is of length 19 and requires 6 doubling and 13 addition operations.

The downside is that the construction used to construct the Scott et al. chain required hours of manual labour, whereas the implementation of the Bernstein algorithm was very quick.

### 3.4 Artificial Intelligence method

To automate the multi-addition chain code generation, we suggested in Dominguez Perez and Scott [DS09] to generalise the Cruz-Cortez et al. [CCRHC08] and Bos and Coster [BC89] methods for generating addition-chains. We construct the code using vectorial addition chains as in [Oli81]. Selecting which integers must remain in the sequence, and which can be discarded, will continuously improve the sequence.

This method, is optimal for families of pairing-friendly curves with small  $\rho$ -value or embedding degree  $k$  as the time needed to find good chains increases with  $\rho$  and  $k$ . Also, the largest coefficients may, at some point, require more storage than a word-size, with the obvious performance hit and search space implications.

A type of Artificial Intelligence method is the Artificial Immune System (AIS). The aim of this method is to mimic the elements and behaviour of the immune system to find solutions to a problem.

### 3.4.1 Artificial Immune System

The immune system is a complex system of cells, molecules and organs which has proven to be capable of performing several tasks, such as recognizing patterns, learning, acquiring memory, and optimization, amongst others. The Immune System is a complex system, whose mechanics are still not fully understood [NV99].

An artificial Immune system is modelled on some of its features. To have a better understanding of the Immune system, we introduce the following definitions.

#### 3.4.1.1 Types of Cells on the Immune System

**Definition 28.** *Leukocytes, also known as “white blood cells” are cells of the immune system which defend the organism against infections or foreign materials.*

The *Lymphocytes*, are a type of leukocytes. They interact with the system during the maturing process or at the initial immune response. There are three types of lymphocytes in the system: B-cells, T-cells, and the Natural Killer cells.

**Definition 29.** *The B-cells are specifically programmed to produce and segregate antibodies. They coordinate themselves to launch an attack on a recognized threat, but they need to be activated.*

**Definition 30.** *The T-cells not only regulate the actions of the other cells, but are also able to launch a direct attack to the infected cells.*

The CD4, a sub-type of the T-Cells, determine the antibody to be generated. The Regulatory T-cells inhibit the action of other cells from the immune system, they are an analogy of the “Military Police” in the context of the immune system.

**Definition 31.** *The Natural Killer-cells are a type of lymphocytes which need no stimulation to start destroying foreign cells, they just read chemical signatures on the cells to kill accordingly.*

#### 3.4.1.2 Immune Engineering

In [NV99], Nunes de Castro and Von Zuben devised an Artificial Immune System (AIS) as a way to get a better understanding of the biological Immune Systems by modelling specific mechanisms, such as generation of diversity, distributed detection and pattern recognition among others.

Immune Engineering (IE) is any engineering or computational tool whose design was inspired by the same immune system tasks. Garret [Gar05b] extends this to any method based on immune abstractions.

There are many biologically inspired computer paradigms, for instance: genetic and evolutionary computation, artificial neural networks and artificial life methods. The choice of AIS metaphors, instead of other computer paradigms in [CCRHC08], may seem to be useful only in a sense of “distinct” in Garret’s notation. However, Cruz-Cortez et al. [Ibidem.] noted that combining these heuristics with other well-known approaches, such as sliding-windows may also be an “effective” method. This is because a plain AIS would only be effective for small values of the exponent  $e$ , in other words, it becomes inefficient as the search space grows.

#### 3.4.1.3 Immune System Metaphors

Before explaining the Cruz-Cortez et al. method [CCRHC08], we will introduce the following commonly used AIS metaphors.

**Immune Network Theory.** An immune system maintains a network of interconnected B-cells. These cells search for antigens and eliminate each other to bring balance to the system.

It was suggested [NV99] that the immune system is composed of a self regulated net-

work of elements that can recognize each other and that can detect the absence of antigens.

An Immune Network has three characteristics: its structure (which describes the interaction of its elements, but not the components itself), the dynamics of the immune elements, and the metadynamics of the network (renewal of death elements, by lack of stimulation or self-reactivation, with the inclusion of new cells.)

A **Negative Selection Mechanism** is a protection mechanism to avoid the destruction of immune system's cells. Is a mechanism to avoid the self-destruction. Tolerance to the immune system's cells is warranted by the system, no cell that can potentially destroy non-recognized antigens are allowed to enter the system, they are destroyed before they are born.

The **Clonal Selection Principle** dictates that only the cells that respond to the antigens must proliferate.

Clonal selection works on both B- and T-cells. When an antigen enters the system, certain B-cells will secrete the appropriate antibody (leaving a chemical signature on the corresponding antigens). These types of cells will be stimulated by the signals left by the CD4 T-cells and they will be cloned by cellular division. These cloned cells, however, will mature into terminal antibody secreting cells, proliferating into the system.

As mentioned by Coelho and Von Zuben in [CV06], once the cloning phase has started, the newly generated cells will enter into a hypermutation state. The hypermutation state forces the cells with the highest affinity to the antigens into a low mutation substate. Similarly, the cloned cells with the lowest affinity to the antigen enter into a high mutation substate.

### 3.5 Addition Chain Construction

For the addition sequence construction, we preferred to use an Artificial Intelligence approach, more specifically, Artificial Immune Systems. This choice seemed optimal in the initial tests. Also, these types of techniques (AIS) can feedback their output as input, potentially generating shorter sequences. The algorithm presented in section 3.4 tries to shorten a

sequence, trying to reduce an already shortened sequence may produce a shorter sequence. The time required is, however, not predictable; if the user would like to spend more time, he may get a better solution. Using one of the other methods presented, once they found a solution, they cannot feedback their result to get an improvement.

Finding an addition-chain is easy, deciding if it is optimal or not is hard, that is an NP-complete problem. We therefore propose to limit the search to a number of improvements of the original sequence, after which the user can decide if he wants to continue searching for improved sequences.

As the intention of the compiler is to generate fast code but not necessarily generate it quickly, we can either start from the proto-sequence and take a while to generate the code, or specify a starting addition chain and improve it. In the compiler, two addition sequences are needed: one for the final exponentiation (Chapter 4), and one for hashing a point in  $G_2$  (Chapter 5).

### 3.5.1 Initial Sequence Generation

For the immune system to be successful it requires one of the B-cells to produce an antibody which matches the antigen. Once a useful antibody is generated, we can work on improving it.

The first step in the immune system is to get an antibody, the analogy for computer science is to get an addition sequence. When a human is sick, he can take a pill or his immune system can develop a cure by itself; in computer science, an initial addition sequence can be supplied, or the compiler can construct one.

To construct the initial sequence, Cruz-Cortez et al. [CCRHC08] insert into the sequence the difference ( $\Delta$ ) of the greatest two elements in the set.

The output of this algorithm is a sequence that is not optimal, but it is an antibody in the immune system analogy, and a starting point for the main algorithm.



### 3.5.2 Auto-Immune Disease

For the addition sequence construction, I followed the Cruz-Cortez et al. [CCRHC08] algorithm [CCRHC08]. Their algorithm was designed to solve an addition chain and not an addition sequence<sup>2</sup>.

We can use their algorithm in the context of finding an addition sequence, however, in terms of an Immune System metaphor, the algorithm needs a mechanism to inhibit the destruction of certain cells to be used to construct addition sequences.

For this compiler, we start with a set of elements which always have to be part of the sequence. For example, the elements of a multi-scalar multiplication operation that we would like to speed up, cannot be removed<sup>3</sup>. The elements in the original sequence are analogous to the normal tissue of the body and a destruction of any of these elements is what we will call an “auto-immune disease”.

To prevent the elimination of normal tissue, the immune system has the Regulatory T-cells, which inhibit the action of the killing of these type of cells. By analogy, a computer system can add a mechanism to prevent the deletion of the original members in the addition sequence being constructed.

### 3.5.3 Hypermutation

The B-cells in the Immune system produce the antibodies. They produce different types of antibodies but not all of them are a match to the antigens. The B-cells that do not produce the corresponding antibodies to the current threat eventually die by lack of stimulation<sup>4</sup>, whereas the others get cloned. In the AIS implementation, we will limit the number of clones to a reduced set to avoid heavy memory use.

By definition, any complete sequence produces antibodies matching the antigens, e.g.: the multi-scalar multiplication. We may not have an optimal sequence for a proto-sequence

<sup>2</sup>At the beginning of the algorithm they do construct a sequence, but they look forward for an addition chain.

<sup>3</sup>Perhaps, we could separate them into several sequences, this however is out of the scope of the present work.

<sup>4</sup>unless another threat is presented and their antibodies matches it

and so we change elements to get a better one, but as no antibody is a perfect match for an antigen, we replace the B-cells in the immune system to get better protection.

If the antibody of a B-cell has a high affinity with the antigen, we prefer this cell to have small variations in its structure when improving it. Conversely, a B-cell whose antibodies, despite matching the antigen, present a low affinity with them will need a higher level of variation in its structure to achieve a high level of affinity.

Cruz-Cortez et al. [CCRHC08] suggested that a change in the lower part of the sequence will cause a chain reaction in the sequence reconstruction, whereas a change in the upper part will only affect a few elements in the sequence. The sequences that have a shorter length than the average will be considered to have high affinity to the optimal sequence, whereas a longer sequence will be considered to have a low affinity. We change elements in the sequence accordingly.

In the Immune System, a B-cell enters a high hypermutation sub-state when its antibodies present a low affinity; hence, our sequence in the AIS will suffer modifications in its lower part. The opposite reaction will happen with modifications in its upper part .

Our hypermutation function is basically a modification of the Cruz-Cortez et al. method [CCRHC08]. The elements of the proto-sequence need to be included, and the only modification is to ensure they are not removed them during the mutation process. This process is described in Algorithm 6.

We have set  $m$  as the maximum number of tries in the search of an element to remove in the sequence. This is because in some cases we have a limited number of options, such as when the sequence to solve is small.

### 3.5.4 Core Function

In the previous section, the hypermutation function is atomic and only replaces an element if the one to be added is “easy” to include in the sequence. An element that requires several elements in the addition sequence is not a candidate for inclusion (“difficult” to include), as it will make the antibody have less affinity to the antigen.

---

**Algorithm 6** Hypermutation algorithm

---

**Input:** Current sequence  $S$ , Original sequence  $O$ , region  $r$  to hypermutate, Max tries  $m$ **Output:** New sequence  $S$ Example Input: 1,2,3,5,8,10,12,1,5,12,  $r = 0, m = 100$ Example Output: 1,2,4,5,8,10,12 for  $i = 3, j = 1, e = 4$  $l \leftarrow \#S$ **repeat**  **if**  $r = 1$  **then**    **repeat**       $m \leftarrow -$        $i \leftarrow \text{Random}(l/2, l)$     **until**  $m = 0$  or  $S_i \notin O$   **else**    **repeat**       $m \leftarrow -$        $i \leftarrow \text{Random}(1, l/2)$     **until**  $m = 0$  or  $S_i \notin O$   **end if**   $j \leftarrow \text{Random}(1, i)$    $e \leftarrow S_i + S_j$ **until**  $m = 0$  or  $e < S_{\#S}$ **if**  $e \in S$  **then**   $S \leftarrow S \setminus S_i$ **else**  **if** DirectSolvable **then**     $S \leftarrow (S \setminus S_i) \cup \{e\}$   **else**

Expensive! Skip.

**end if****end if**Return  $S$ 

---

For longer or more complex sequences, that is, sequences with very sparse components, there may not be another choice. This is not necessarily bad, but should be treated with caution as the chain may grow substantially.

Algorithm 7 finds several candidates for the final addition sequence. Each candidate is shorter than the previous one. After a user defined number of generations in the algorithm the smallest chain is presented. All the previous best candidates are stored.

---

**Algorithm 7** Construct the multi-addition chain
 

---

**Input:**  $O$  the original sequence,  $S$  a start sequence,  $M$  the number of generations to mutate and  $G$  clones per generation.

**Output:**

```

 $W \leftarrow G$  clones of  $S$ 
for  $i \leftarrow 1$  to  $M$  do
  for  $j \leftarrow 1$  to  $G$  do
    if  $\#W[j] \geq avg$  then
      hypermutation( $W[j], 1, O$ )
    else
      hypermutation( $W[j], 0, O$ )
    end if
     $\diamond$  Eliminate unused elements
     $\diamond$  Create elements that are not solved
  end for
   $\diamond$  Get the new  $avg$ 
   $\diamond$  Detect the smallest of this generation
  if  $\text{smallest-here} < \text{smallest-ever}$  then
     $\text{smallest-ever} \leftarrow \text{smallest-here}$ 
  end if
  if  $\#S - \#\text{smallest-here} = stop$  then
    return smallest
  end if
end for

```

---

The algorithm executes for  $M$  generations of improvements, and it will try to reduce elements from  $S$ , the first sequence created from the proto-sequence  $C$  containing the coefficients of the polynomial representation of the final exponentiation  $(p^k - 1)/r \in \mathbb{F}_{p^k}^*$ .

### 3.5.5 Results

For this algorithm, we use the same proto-sequence as in section 3.3.2 on page 62:

$$\{1, 2, 4, 5, 7, 10, 15, 26, 28, 30, 55, 75, 80, 100, 108, 144\}.$$

After a few generations the following addition chain results:

$$\{1, 2, 4, 5, 7, 10, 15, \underline{25}, 26, 28, 30, 55, \underline{60}, 75, 80, 100, 108, \underline{134}, 144\}$$

which is 2 elements shorter than the initial sequence and identical in size to the Scott et al. [SBC<sup>+</sup>09b] sequence.

This method can not only be applied to the final exponentiation problem detailed in Chapter 4, but can also be used to improve the fast hashing to  $G_2$  problem detailed in Chapter 5.

## 3.6 Comparison of the Methods

Consider the sequence: [62, 87, 112, 248, 348, 385, 434, 448, 609, 784, 882, 931, 1029, 1540, 2401, 2695, 2744, 3528, 3724, 4116, 4802, 6174, 6517, 7203, 9604, 10976, 16807, 19208, 25539, 25807, 26075, 33614, 67228, 79436, 83914, 88392, 117649, 128499, 151959, 178773, 180649, 182525, 283073, 333347, 428526, 433895, 597849, 621859, 899493, 936488, 949767, 1126706, 1478520, 1507142, 1817557, 1858374, 1981511, 2333429, 2487436, 3037265, 3189900, 4184943, 4353013, 5428661, 6204184, 6648369, 12722899, 17983490, 38000627] which is our proto-sequence to solve. This is a particularly long sequence that comes from the final exponentiation of a KSS  $k = 36$  curve, which will be explained in detail in Chapter 4.

Using the Binary Method we obtained the following result: [ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 18, 21, 22, 23, 26, 29, 30, 31, 33, 34, 35, 41, 42, 43, 46, 49, 52, 53, 54, 55, 57, 59, 60, 62, 63, 84, 87, 99, 109, 112, 113, 115, 117, 121, 122, 123, 124, 126, 127, 132, 134, 142, 143, 156, 158, 171, 191, 196, 207, 214, 222, 230, 232, 239, 248, 251, 252, 254, 287, 291, 297, 315, 319, 343,

348, 353, 355, 375, 377, 379, 383, 385, 399, 414, 434, 448, 451, 459, 478, 489, 493, 495, 501, 503, 507,  
511, 523, 538, 566, 572, 575, 609, 631, 632, 652, 680, 701, 708, 736, 765, 784, 825, 882, 885, 891, 921,  
931, 979, 981, 989, 991, 1001, 1015, 1019, 1021, 1022, 1029, 1179, 1194, 1339, 1343, 1346, 1395, 1407,  
1439, 1479, 1483, 1517, 1525, 1534, 1540, 1640, 1694, 1758, 1782, 1934, 1998, 2014, 2015, 2027, 2030,  
2065, 2329, 2345, 2401, 2429, 2593, 2695, 2744, 2756, 2783, 2784, 2941, 3031, 3519, 3528, 3557, 3588,  
3716, 3724, 3762, 3780, 3782, 4029, 4046, 4065, 4087, 4095, 4116, 4418, 4802, 4941, 5471, 6047, 6174,  
6517, 6779, 6798, 6845, 6911, 6964, 7202, 7203, 7942, 7996, 8071, 8125, 9604, 10239, 10360, 10976,  
11967, 12029, 12795, 13823, 13954, 14071, 14318, 14319, 15400, 15462, 15471, 15532, 15559, 15870,  
15894, 16063, 16092, 16127, 16807, 17267, 17367, 18423, 18760, 19208, 19973, 20388, 20738, 21475,  
22856, 24085, 25539, 25807, 26075, 27974, 28787, 28985, 28987, 29051, 30207, 30523, 31262, 32435,  
33614, 35277, 36550, 36780, 36812, 40646, 45047, 46775, 49143, 50665, 51711, 52605, 55313, 60594,  
60851, 67228, 63677, 64883, 67228, 70639, 72541, 79436, 79779, 80542, 81750, 83914, 86073, 88392,  
94203, 94521, 94523, 95795, 98302, 108527, 111098, 112127, 112731, 113804, 114094, 114280, 117649,  
119740, 126841, 128499, 130348, 130651, 141011, 148471, 151959, 168070, 178773, 180649, 182525,  
192350, 195600, 219895, 241663, 245501, 250811, 256989, 260063, 261598, 283073, 283690, 333347,  
418741, 428526, 430044, 433895, 438238, 454630, 517307, 597849, 618687, 621859, 872633, 899493,  
936488, 949767, 970461, 1012595, 1013975, 1126706, 1161134, 1478520, 1507142, 1814527, 1817557,  
1858374, 1893642, 1945596, 1981511, 1994619, 2085875, 2281475, 2318335, 2333429, 2487436, 3037265,  
3108703, 3189900, 3923439, 4046410, 4062508, 4184943, 4353013, 5428661, 6074530, 6204184, 6648369,  
12722899, 15702015, 17983490, 20017137, 38000627 ]

The previous sequence, obtained from the Binary Method, has 363 elements and was generated in a few milliseconds. This sequence is 526% longer than the *initial sequence*<sup>5</sup>. The underlined elements in the sequence were not part of the proto-sequence.

Using the AIS method we obtained the following result: [ 1, 2, 4, 5, 7, 8, 10, 11, 14,  
19, 20, 21, 29, 37, 48, 49, 60, 62, 67, 87, 98, 112, 124, 125, 127, 146, 147, 157, 196, 245, 248, 268,  
288, 294, 348, 385, 394, 434, 448, 460, 560, 561, 609, 676, 686, 784, 882, 931, 1029, 1078, 1372,  
1519, 1529, 1540, 1617, 1862, 1876, 1989, 2009, 2401, 2547, 2695, 2724, 2744, 3528, 3724, 3871, 4018,

<sup>5</sup>The initial sequence is the first sequence generated by the algorithm, see section 3.5.1.

4116, 4478, 4578, 4802, 5362, 5369, 5488, 5613, 6174, 6517, 7203, 9604, 9761, 9998, 10290, 10976, 13034, 13279, 13377, 14063, 15092, 16709, 16807, 19208, 22932, 24010, 25539, 25807, 26075, 27048, 28622, 28910, 29449, 33614, 40817, 48020, 52136, 58898, 63476, 67228, 71099, 76712, 79436, 83914, 88392, 104272, 105644, 106673, 117649, 123137, 128499, 129654, 142198, 151959, 152635, 154007, 168070, 176784, 178773, 180649, 182525, 186782, 253624, 283073, 285719, 299782, 308014, 310415, 333347, 365050, 428526, 433895, 444185, 549829, 597849, 621859, 645869, 899493, 907578, 936488, 949767, 1099658, 1126706, 1291738, 1478520, 1507142, 1817557, 1858374, 1981511, 2033647, 2333429, 2487436, 3037265, 3189900, 4067294, 4184943, 4353013, 5260591, 5428661, 6074530, 6204184, 6648369, 12722899, 17983490, 35966980, 38000627 ]

Again, the underlined elements in the sequence were not part of the proto-sequence. This sequence, generated using the AIS method, has 174 elements but was improved by 6000 generations with the use of 5 clones. As we increase the number of generations in the algorithm, the shorter the sequence gets. It is expected that shorter addition sequences can be found if the program runs for more generations. The addition chain constructed by the approximation method from Cruz-Cortez et al., has a length of 189, so using this method we were able to find a shorter sequence.

### 3.7 Final Thoughts on Addition Chains

I have implemented a couple of algorithms for finding addition sequences. These algorithms have proven to be an effective solution for long sequences. Currently, the sequence presented in the previous section is the shortest for such a proto-sequence; however, we can always feed it back to get a shorter sequence. These algorithms help on the implementation of multi-scalar multiplications of large magnitude. This will help in the implementation of pairings functions with a high security level.

Our code can generate shorter addition chains than the binary method [Ber09]. The binary method is really fast in finding a multi-addition chain, but for some proto-sequences the sequence can be substantially longer.

On the another hand, the binary method tends to prefer doubling to addition operations.

This can be very useful in some scenarios, such as when a doubling takes less than a half the time to compute an addition, or the equivalent operation. A formula to convert the equivalent cost of a doubling to an addition may be required. Unless we have a certain conversion rate, the shortest possible sequence should be selected.

For small to medium sized addition chains, our code is the best option, as it creates the shortest sequences in a minimal amount of time. It is also easy to decide which sequence to choose for further shrinking, this is, which sequence is the best and which one has more possibilities of success in finding a shorter solution.

The situation is different for longer sequences. The longer, sparser and the larger the elements in the sequence, the longer it will take to find a shorter solution. A shorter solution will be desired for these type of sequences as the number of operations can be very large. At higher security levels, small variations on the chosen addition sequence will be reflected in an order of magnitude of several milliseconds, which is a big amount of time for a pairing computation, even for higher security levels. The time to solve these sequences can be large, but the wait should be worth it. When using the code for finding an addition chain of these type of sequences, one must be prepared for the wait.



## 4 THE FINAL EXPONENTIATION

*“Natural disasters does not exist, there are disasters built and provoked by the human kind” – Juan Carlos Mora Chaparro*

**A**FTER the execution of the Miller loop in the Tate pairing we need an exponentiation to  $(p^k - 1)/r$  in  $\mathbb{F}_{p^k}^*$  to get the  $r$ -th roots of unity. This is because the Tate pairing is defined with one group over a curve in the extension field and other group over a curve in the base field (Miller lite), and we obtain a group  $\mathbb{F}_{p^k}^*/(\mathbb{F}_{p^k}^*)^r$ . With the final exponentiation, we obtain the desired group  $G_T \subset \mathbb{F}_{p^k}^*$  [Gal05].

Unfortunately, this exponentiation is potentially very time consuming. Hu, Dong and Pei [HDP05] made initial efforts on improving the final exponentiation without considering the families of curves. Devegili, Scott and Dahab in [DSD07] exploited the structure of the BN family of curves and found a method to compute a very fast final exponentiation. They only showed the results but no construction method. There is no algorithm to reproduce their results to other families of curves.

In [SBC<sup>+</sup>09b] we generalized the Devegili et al. final exponentiation method to any family of pairing-friendly curves. Our method consisted in exploiting the parametrisation of the curve and the use of addition sequences as a multi-scalar exponentiation method. To find an addition chain, we relied on a manual exhaustive search. Later, we constructed the final exponentiation using the Olivos method [Oli81].

In a later work with Scott §3.5 and [DS09], we found a method to get rid of the manual part of the algorithm.

The rest of this chapter is organized as follows: §4.1 gives details of initial work on improving the final exponentiation computation. We present the Devegili, Scott and Dahab method in §4.2 to compute the final exponentiation for a specific family of curves. Our method is presented in §4.3 which can be applied to several families of curves. Final

thoughts and future work is discussed in §4.4.

## 4.1 The Du, Hong and Pei Method

One approach to improving the cryptographic pairings functions is to find ways to reduce the number of Miller loop iterations. Another approach is to reduce the complexity of the operation and the time used for the final exponentiation after the Miller loop in the pairing function. This section shows how one can speed-up this expensive computation.

As shown previously in Chapter 2, this function requires 2 finite cyclic additively-written groups denoted as  $G_1$  and  $G_2$ . From each group a single element is taken, and they are mapped to an element in a third group:  $G_T$ . This group is a finite cyclic multiplicatively-written group of order  $r$ .

One is required to raise the output of the Miller loop to  $(p^k - 1)/r$  in order to have a single representative of the class. This exponentiation is known to be very time consuming as shown in [HDP05]. Hu, Dong and Pei estimated that this final powering takes more than one half of the total computation if a direct exponentiation method is used. We have found using the Magma language in our experiments, that this operation can take most of the time of the pairing computation as we use families with larger embedding degrees.

For a MNT curve with  $k = 6$ , let

$$(p^2 - p + 1)/r = k_1 \cdot p + k_0$$

where  $0 \leq k_0, k_1 < p$ . Then,

$$(p^6 - 1)/r = (k_1 \cdot p + k_0)(p + 1)(p^3 - 1).$$

We can efficiently compute the final exponentiation step with Algorithm 8. Here,

$$f^p \in \mathbb{F}_{p^6}$$

can easily be computed with the use of the Frobenius operator. In this way they claimed to reduce the computation time by a factor of 4 to 5.

---

**Algorithm 8** Hu, Dong and Pei final exponentiation algorithm
 

---

**Input:**  $f \in \mathbb{F}_{p^6}$ ,  $k_1, k_0 \in \mathbb{F}_p$

**Output:**  $f^{(p^6-1)/r}$

```

 $g \leftarrow f^{k_1}$ 
 $f \leftarrow f^{k_0}$ 
 $g \leftarrow g^p$ 
 $f \leftarrow gf$ 
 $g \leftarrow f^p$ 
 $f \leftarrow gf$ 
 $g \leftarrow ((f^p)^p)^p$ 
 $f \leftarrow f^{-1}$ 
 $f \leftarrow gf$ 
return  $f$ 

```

---

The authors did not show any way to get the  $k_0$  and  $k_1$  values; however, these can be easily computed by dividing by  $p$  and finding a quotient and a remainder.

## 4.2 Devegili, Scott and Dahab Method

The easy exponentiation is that part of the final exponentiation that can be performed with the use of the Frobenius map [GPS06]. The hard part of the exponentiation is the exponentiation to the power of  $\phi_k(p)/r \in \mathbb{F}_{p^k}^*$ , where  $\phi_k$  is the  $k$ -th cyclotomic polynomial,  $p$  is the field size and  $r$  is the subgroup size. This method is also referred as the *truncated final exponentiation* [Sco07] (or hard part).

Devegili, Scott and Dahab in [DSD07] simplified the hard part of the final exponentiation for the BN curves [BN06], which is a family of elliptic curves with the polynomial representation of the parameters of the curve shown in Equation 2.13.

The hard part of the final exponentiation can be expanded to:

$$\begin{aligned}
 &46656x^{12} - 139968x^{11} + 241056x^{10} - 272160x^9 + 225504x^8 - 138672x^7 \\
 &\quad + 65448x^6 - 23112x^5 + 6264x^4 - 1188x^3 + 174x^2 - 6x + 1.
 \end{aligned} \tag{4.1}$$

However, Devegili et al. suggested a better representation in base- $p$  of equation 4.1.

$$p^3 + (6x^2 + 1)p^2 + (36x^3 - 18x^2 + 12x + 1)p + (36x^3 - 30x^2 + 18x - 2). \quad (4.2)$$

Then, they constructed Algorithm 9, which can be used for the hard part of the final exponentiation.

---

**Algorithm 9** Devegili, Scott and Dahab final exponentiation algorithm

---

**Input:**  $f$ ,  $x$  and  $p$

**Output:**  $f^{(p^4 - p^2 + 1)/r}$

$g \leftarrow f^{6x-5}$

$h \leftarrow f^p$  {Frobenius map}

$h \leftarrow gh$

Precompute  $f^p$ ,  $f^{p^2}$  and  $f^{p^3}$  {Frobenius map}

$f \leftarrow f^3 \cdot [h \cdot (f^p)^2 \cdot f^{p^2}]^{6x^2+1} \cdot h \cdot (f^p \cdot f)^9 \cdot g \cdot f^4$

**return**  $f$

---

This algorithm is fast and does not use as much memory as a multi-exponentiation method.

### 4.3 A new method

Families of pairing-friendly elliptic curves use fixed polynomials to describe their parameters. One way to exploit this fact is with  $\pi$ , the Frobenius exponentiation.

To speed up the final exponentiation, firstly, the exponent must be appropriately factored. In the KSS:  $k = 18$  curves [KSS08], we factor  $(p^{18} - 1)$  into  $(p^9 - 1)$  and  $(p^9 + 1)$ . We can factor  $(p^9 + 1)$  into  $(p^3 + 1)$  and  $\Phi_{18}(p)$ , since  $r \mid \Phi_{18}(p)$ . One can apply the Frobenius map on  $(p^9 - 1)$  and  $(p^3 + 1)$ , and then remains the hard exponent  $\Phi_{18}(p)/r$ , where  $\Phi_{18}(p) = (p^6 - p^3 + 1)$ .

Let  $\lambda = (p^6 - p^3 + 1)/r$  be the “hard part” of the final exponent. To simplify the exponent  $\lambda$ , for example, it can be represented in base- $p$  as  $\lambda_0 + \lambda_1 \cdot p + \lambda_2 \cdot p^2 + \dots + \lambda_5 \cdot p^5$ ,

such that  $f^\lambda$ , where  $f$  is the element to exponentiate, can be written as:

$$f^\lambda = f^{\lambda_0 + \lambda_1 \cdot p + \lambda_2 \cdot p^2 + \dots + \lambda_5 \cdot p^5} = (f)^{\lambda_0} \cdot (f^p)^{\lambda_1} \cdot (f^{p^2})^{\lambda_2} \dots (f^{p^5})^{\lambda_5}$$

Hence, we can apply the Frobenius map over  $\lambda$  to obtain a speed-up. The values of  $\lambda_i$  for the KSS:  $k = 18$  are as follows:

$$\begin{aligned} \lambda_5 &= 49/3x^2 + 245/3x + 343/3. \\ \lambda_4 &= 7/3x^6 + 35/3x^5 + 49/3x^4 + 112/3x^3 + 581/3x^2 + 784/3x. \\ \lambda_3 &= -5/3x^7 - 25/3x^6 - 35/3x^5 - 29x^4 - 150x^3 - 203x^2 + 18. \\ \lambda_2 &= -49/3x^5 - 245/3x^4 - 343/3x^3 - 931/3x^2 - 4802/3x - 6517/3. \\ \lambda_1 &= 14/3x^6 + 70/3x^5 + 98/3x^4 + 91x^3 + 469x^2 + 637x. \\ \lambda_0 &= -x^7 - 5x^6 - 7x^5 - 62/3x^4 - 319/3x^3 - 434/3x^2 + 1. \end{aligned} \tag{4.3}$$

A minor difficulty arises due to the common denominator of 3 which occurs here. We suggest a simple solution. Since 3 is co-prime to  $r$ , evaluate instead the third power of the pairing. This does not affect the important properties of the pairing when  $r$  is of cryptographic size, and now we can simply ignore this denominator, which we call *the awkward denominator*.

After the multiplication by 3, the  $\lambda_0$  becomes:  $-3x^7 - 15x^6 - 21x^5 - 62x^4 - 319x^3 -$

$434x^2 + 3$ . The rest of the  $\lambda_i$  are shown in equation 4.4.

$$\begin{aligned}
3.\lambda_5 &= 49x^2 + 245x + 343. \\
3.\lambda_4 &= 7x^6 + 35x^5 + 49x^4 + 112x^3 + 581x^2 + 784x. \\
3.\lambda_3 &= -5x^7 - 25x^6 - 35x^5 - 87x^4 - 450x^3 - 609x^2 + 54. \\
3.\lambda_2 &= -49x^5 - 245x^4 - 343x^3 - 931x^2 - 4802x - 6517. \\
3.\lambda_1 &= 14x^6 + 70x^5 + 98x^4 + 273x^3 + 1407x^2 + 1911x. \\
3.\lambda_0 &= -3x^7 - 15x^6 - 21x^5 - 62x^4 - 319x^3 - 434x^2 + 3. \tag{4.4}
\end{aligned}$$

The proto-sequence for equation 4.4, which consist of its coefficients, is the following:  
 $\{3, 5, 7, 14, 15, 21, 25, 35, 49, 54, 62, 70, 87, 98, 112, 245, 273, 319, 343, 434, 450, 581, 609, 784, 931, 1407, 1911, 4802, 6517\}$ .

Now, to compute  $f^{\lambda_0}$ , we use the following equation:

$$f^{(-x^7)^3} \cdot f^{(-x^6)^{15}} \cdot f^{(-x^4)^{62}} \cdot f^{(-x^3)^{319}} \cdot f^{(-x^2)^{434}} \cdot f^3$$

from equation 4.4. We associate each element in the proto-sequence with the corresponding elements in the  $\lambda_i$ . For example, the first element in the proto-sequence is 3. Hence,  $x_0 = f \cdot \frac{1}{f^{(x^7)}}$  as the coefficient 3 is present in  $\lambda_0$  as exponent in  $f^{(-x^7)^3}$  and  $f^3$ .

In the case of 5,  $x_1 = \pi^3(\frac{1}{f^{x^7}})$  since the elements have implicit a  $p^i$  exponent, but we can use the Frobenius exponentiation instead.

Following this construction, the  $x_i$  elements are formed as in the following listing:

$x_0 \leftarrow f \cdot \frac{1}{f^{x^7}}$	$x_7 \leftarrow \pi^3(\frac{1}{f^{x^5}}) \cdot \pi^4(f^{x^5})$	$x_{14} \leftarrow \pi^4(f^{x^3})$
$x_1 \leftarrow \pi^3(\frac{1}{f^{x^7}})$	$x_8 \leftarrow \pi^2(\frac{1}{f^{x^5}}) \cdot \pi^4(f^{x^4}) \cdot \pi^5(f^{x^2})$	$x_{15} \leftarrow \pi^2(\frac{1}{f^{x^4}}) \cdot \pi^5(f^x)$
$x_2 \leftarrow \pi^4(f^{x^6})$	$x_9 \leftarrow \pi^3(f)$	$x_{16} \leftarrow \pi(f^{x^3})$
$x_3 \leftarrow \pi(f^{x^6})$	$x_{10} \leftarrow \pi^4(f^{x^4})$	$x_{17} \leftarrow \frac{1}{f^{x^3}}$
$x_4 \leftarrow \pi(f^{x^6})$	$x_{11} \leftarrow \pi(f^{x^5})$	$x_{18} \leftarrow \pi^2(\frac{1}{f^{x^3}}) \cdot \pi^5(f)$
$x_5 \leftarrow \pi(f^{x^5})$	$x_{12} \leftarrow \pi^3(\frac{1}{f^{x^4}})$	$x_{19} \leftarrow \frac{1}{f^{x^2}}$
$x_6 \leftarrow \pi^3(\frac{1}{f^{x^6}})$	$x_{13} \leftarrow \pi(f^{x^4})$	$x_{20} \leftarrow \pi^3(\frac{1}{f^{x^3}})$

$$\begin{array}{lll}
x_{21} \leftarrow \pi^4(f^{x^2}) & x_{24} \leftarrow \pi^2(\frac{1}{fx^2}) & x_{27} \leftarrow \pi^2(\frac{1}{fx}) \\
x_{22} \leftarrow \pi^3(\frac{1}{fx^2}) & x_{25} \leftarrow \pi(f^x 2) & x_{28} \leftarrow \pi^2(\frac{1}{f}) \\
x_{23} \leftarrow \pi^4(f^x) & x_{26} \leftarrow \pi(f^x) &
\end{array}$$

We need to perform a lot of exponentiations, and we will use an addition chain to optimize these operations. For this example, the computation can be done with the use of the following addition sequence:  $\{1, 2, 3, 4, 5, 7, 8, 10, 14, 15, 16, 21, 25, 28, 35, 42, 49, 54, 62, 70, 87, 98, 112, 147, 245, 273, 294, 319, 343, 392, 434, 450, 581, 609, 784, 931, 1162, 1407, 1862, 1911, 3724, 4655, 4802, 6517\}$ <sup>1</sup>.

To construct the operations required for the final exponentiation, we use the Olivos method [Oli81] (also see [CF06, §9.2]), which uses vectorial chains as follows:

Given an addition chain  $\Gamma = \{1, 2, 6, 12, 18, 30\}$  and its corresponding addition sequence  $s = \{1, 2, 3, 6, 12, 18, 30, 36\}$ , we construct a vector chain:  $v = \{[1, 0], [0, 1], [1, 1], [2, 2], [3, 2], [6, 4], [12, 8], [18, 12], [30, 20], [36, 24]\}$ .

To construct the vector chain matrix shown in Table 4.1, we start with  $[1, 0]$  and  $[2, 2]$ . To compute row  $t_2 = [3, 2]$ , we set  $(t_0, c_0) = 1$  and  $(t_1, c_1) = 1$  by induction (prioritizing a doubling over an addition). This means that  $t_2 = t_0 + t_1$ , which, translated into vector operations is equivalent to  $[3, 2] = [1, 0] + [2, 2]$ . The type of operation is expressed in  $(t_2, c_2)$ ,  $(t_2, c_3)$  1 denotes an addition, 2 denotes a doubling.

The remaining cells of the column, if any, are the summation the corresponding cells in the column. For example, in  $(t_5, c_6 \text{ to } c_7) = [3, 2]$  since  $t_5 = t_4 + t_3$ , hence  $[3, 2] = [1, 0] + [2, 2]$ . The arrows at the right of the Table 4.1 denote the rows containing the elements in  $\Gamma$ .

The Table 4.2 presents the typical output of the method for a KSS curve with  $k = 18$ . The  $t_i$ 's represent temporary variables.

A straight-forward implementation of the vectorial addition chains method requires an increasing number of temporary variables. To reduce the number of temporary variables, we use Algorithm 10. This algorithm uses a matrix  $u$  with the same dimension as the addition

<sup>1</sup>the underlined elements in the sequence were not part of the proto-sequence.

$t_0$	$c_0$	$c_1$	$c_2$	$c_3$	$c_4$	$c_5$	$c_6$	$c_7$	$c_8$	$c_9$	$c_{10}$	$c_{11}$	$c_{12}$	$c_{13}$	$\leftarrow$
$t_1$	1	0	1	0	0	0	0	0	0	0	0	0	0	0	$\leftarrow$
$t_2$	2	2	0	1	0	0	0	0	0	0	0	0	0	0	$\leftarrow$
$t_3$	3	2	1	1	1	0	0	0	0	0	0	0	0	0	
$t_4$	6	4	2	2	2	2	1	0	1	0	0	0	0	0	$\leftarrow$
$t_5$	12	8	4	4	4	4	2	2	0	1	1	0	0	0	$\leftarrow$
$t_6$	18	12	6	6	6	6	3	2	1	1	0	1	1	0	$\leftarrow$
$t_7$	30	20	10	10	10	10	5	4	1	2	1	1	0	0	$\leftarrow$
$t_8$	36	24	12	12	12	12	6	4	2	2	0	2	2	2	$\leftarrow$

Table 4.1: Constructing the vector chain

$t_0$	$\leftarrow$	$x_{27}.x_{28}$	$t_0$	$\leftarrow$	$x_{27}.x_{28}$
$t_1$	$\leftarrow$	$t_0.x_{24}$	$t_1$	$\leftarrow$	$t_0.x_{24}$
$t_2$	$\leftarrow$	$t_0.t_0$	$t_0$	$\leftarrow$	$t_0.t_0$
$t_3$	$\leftarrow$	$t_2.x_{28}$	$t_0$	$\leftarrow$	$t_0.x_{28}$
$t_4$	$\leftarrow$	$t_3.x_{26}$	$t_0$	$\leftarrow$	$t_0.x_{26}$
$t_5$	$\leftarrow$	$x_8.x_{26}$	$t_5$	$\leftarrow$	$x_8.x_{26}$
$t_6$	$\leftarrow$	$t_4.t_4$	$t_0$	$\leftarrow$	$t_0.t_0$
$t_7$	$\leftarrow$	$t_6.t_1$	$t_3$	$\leftarrow$	$t_0.t_1$
$t_8$	$\leftarrow$	$x_{15}.x_{25}$	$t_1$	$\leftarrow$	$x_{15}.x_{25}$
$t_9$	$\leftarrow$	$x_{25}.x_{25}$	$t_0$	$\leftarrow$	$x_{25}.x_{25}$
$\dots$			$\dots$		

Table 4.2: Partial final exponentiation code from the Olivos and Scott et al. method. KSS Curve:  $k = 18$ Table 4.3: Partial final exponentiation code for the same curve, with a reduced number of  $t_i$  elements.

and doubling operations generated from the Olivos method, to keep track of the processed elements. It flags every position (let's say, with a "1") that corresponds to a precomputed value from the original matrix (i.e.: the  $t_0$  from  $t_1 = t_0 \cdot x_{24}$ ).

From bottom-to-top, the first  $t_i$  element which will act as an  $R^2$ -value in the code generation is replaced with a another temporary  $t_j$ , with  $j = 0$  initially. It continues looking to the top for all of the  $t_i$ 's occurrences until the current  $t_i$  is presented as an  $L$ -value, placing a flag on the corresponding position in  $u$  for every replacement. We will use the same  $t_j$  and continue to the top of the matrix recycling memory space. Once the scan reaches the top, we choose the next  $t_i$ , and we repeat the flag-and-replace procedure with a new  $t_j$  from the bottom. At the end, every element of  $u$  will have a flag. See Table 4.3 for the sample

<sup>2</sup>Consult §2.8 for an extended discussion on RH- and LH- values.



---

**Algorithm 10** Reducing terms from the Olivos method.

---

**Input:**  $f$  final exponentiation code as a Matrix, i.e.:  $f_i = [t_0][t_0][x_1]$  ( $t_0 \leftarrow t_0 + x_1$ )

**Output:**  $f$  with a reduced number of  $t_i$ 's

```

1: Let  $u$  be a matrix of the same dimensions as  $f$ 
2: For all  $1 \leq i \leq \#f$  and  $1 \leq j \leq 3$ : set  $u_{i,j}$  to 1 if  $(f_{i,j} \neq t_k)$ 
3: Set  $\text{prev} \leftarrow f_{\#f,1}$  and  $i \leftarrow \#f$ 
4: while ( $u_{i,j}$  has elements not = 1) or ( $i > 0$ ) do
5:   if  $u_{i,j} = 0$  and  $f_{i,1} = \text{prev}$  then
6:      $f_{i,1} \leftarrow \text{current } t_k$ ;  $u_{i,1} \leftarrow 1$ 
7:     Scan bottom-top for  $f_{i,1}$  on  $f_{\forall, [2-3]}$ 
8:     if  $i = 1$  then
9:       Increase  $t_k$ 
10:      Scan bottom-top for  $u_{[\#f..1], [2|3]} = 0$ 
11:      if  $i = 1$  then
12:        Done
13:      else
14:        Rewrite  $t_k$ 
15:        Tag element to 1 in  $u$ 
16:        Decrease  $i$ 
17:      end if
18:    end if
19:  end if
20: end while

```

---

output of this Algorithm.

Finally, we can also recycle the  $x_i$  variables and compute them, either *in-place* or by an assignment *just-in-time* before the operation. The computation for a KSS curve with  $k = 18$  becomes as follows:

$$\begin{array}{lll}
x_A \leftarrow \pi^2\left(\frac{1}{f^x}\right) & t_3 \leftarrow t_0.x_B & t_7 \leftarrow t_6.x_B \\
x_B \leftarrow \pi^2\left(\frac{1}{f}\right) & x_B \leftarrow \frac{1}{f^{x^2}} & t_4 \leftarrow t_2.t_4 \\
t_0 \leftarrow x_A.x_B & t_0 \leftarrow t_3.x_B & x_B \leftarrow \pi(f^{x^6}) \\
x_B \leftarrow \pi^2\left(\frac{1}{f^{x^2}}\right) & t_4 \leftarrow t_3.t_4 & t_2 \leftarrow t_2.x_B \\
t_1 \leftarrow t_0.x_B & x_B \leftarrow \pi^3\left(\frac{1}{f^{x^3}}\right) & t_4 \leftarrow t_4.t_4 \\
t_0 \leftarrow t_0.t_0 & t_0 \leftarrow t_0.x_B & t_4 \leftarrow t_4.t_5 \\
x_B \leftarrow \pi^2\left(\frac{1}{f}\right) & t_3 \leftarrow t_0.t_2 & x_A \leftarrow \pi^4(f^{x^4}) \\
t_0 \leftarrow t_0.x_B & x_B \leftarrow \pi^2\left(\frac{1}{f^{x^3}}\right).\pi^5(f) & x_B \leftarrow \pi^3\left(\frac{1}{f^{x^4}}\right) \\
x_B \leftarrow \pi(f^x) & t_2 \leftarrow t_3.x_B & t_5 \leftarrow x_A, x_B \\
t_0 \leftarrow t_0.x_B & t_3 \leftarrow t_3.t_5 & x_B \leftarrow \pi^3\left(\frac{1}{f^{x^4}}\right) \\
x_A \leftarrow \pi^2\left(\frac{1}{f^{x^5}}\right).\pi^4(f^{x^4}).\pi^5(f^{x^2}) & t_5 \leftarrow t_3.t_2 & t_3 \leftarrow t_3.x_B \\
x_B \leftarrow \pi(f^x) & x_B \leftarrow \frac{1}{f^{x^3}} & x_A \leftarrow \pi(f^{x^5}) \\
t_5 \leftarrow x_A.x_B & t_2 \leftarrow t_2.x_B & x_B \leftarrow \pi(f^{x^5}) \\
t_0 \leftarrow t_0.t_0 & x_A \leftarrow \pi^3\left(\frac{1}{f^{x^6}}\right) & t_6 \leftarrow x_A.x_B \\
t_3 \leftarrow t_0.t_1 & x_B \leftarrow \frac{1}{f^{x^3}} & t_7 \leftarrow t_6.t_7 \\
x_A \leftarrow \pi^2\left(\frac{1}{f^{x^4}}\right).\pi^5(f^x) & t_3 \leftarrow x_A.x_B & x_B \leftarrow \pi^3(f) \\
x_B \leftarrow \pi(f^{x^2}) & t_2 \leftarrow t_2.t_2 & t_6 \leftarrow t_5.x_B \\
t_1 \leftarrow x_A.x_B & t_4 \leftarrow t_2.t_4 & t_4 \leftarrow t_6.t_4 \\
x_A \leftarrow \pi(f^{x^2}) & x_B \leftarrow \pi(f^{x^3}) & x_B \leftarrow \pi^3\left(\frac{1}{f^{x^7}}\right) \\
x_B \leftarrow \pi(f^{x^2}) & t_2 \leftarrow t_1.x_B & t_6 \leftarrow t_6.x_B \\
t_0 \leftarrow x_A.x_B & x_A \leftarrow \pi(f^{x^3}) & t_0 \leftarrow t_4.t_0 \\
x_B \leftarrow \pi^4(f^{x^2}) & x_B \leftarrow \pi^3\left(\frac{1}{f^{x^2}}\right) & x_B \leftarrow \pi^4(f^{x^6}) \\
t_0 \leftarrow t_0.x_B & t_1 \leftarrow x_A.x_B & t_4 \leftarrow t_4.x_B \\
x_B \leftarrow \pi^4(f^x) & t_6 \leftarrow t_2.t_4 & t_0 \leftarrow t_0.t_0 \\
t_2 \leftarrow t_3.x_B & x_B \leftarrow \pi(f^{x^4}) & x_B \leftarrow \pi(f^{x^5}) \\
x_B \leftarrow \pi^2\left(\frac{1}{f^x}\right) & t_4 \leftarrow t_2.x_B & t_0 \leftarrow t_0.x_B \\
t_4 \leftarrow t_3.x_B & x_B \leftarrow \pi^4(f^{x^3}) & t_1 \leftarrow t_7.t_1 \\
t_2 \leftarrow t_2.t_2 & t_2 \leftarrow t_6.x_B & t_4 \leftarrow t_4.t_7 \\
x_B \leftarrow \pi^3\left(\frac{1}{f^{x^2}}\right) & x_B \leftarrow \pi^3\left(\frac{1}{f^{x^5}}\right).\pi^4(f^{x^5}) & t_1 \leftarrow t_1.t_1
\end{array}$$

$t_2 \leftarrow t_1.t_2$	$t_2 \leftarrow t_2.t_2$	$t_0 \leftarrow t_0.t_2$
$t_1 \leftarrow t_0.t_3$	$t_2 \leftarrow t_2.t_4$	$x_B \leftarrow f.\frac{1}{f^{x^7}}$
$x_B \leftarrow \pi^3(\frac{1}{f^{x^3}})$	$t_0 \leftarrow t_0.t_0$	$t_0 \leftarrow t_0.x_B$
$t_0 \leftarrow t_1.x_B$	$t_0 \leftarrow t_0.t_3$	$x_B \leftarrow f.\frac{1}{f^{x^7}}$
$t_1 \leftarrow t_1.t_6$	$t_1 \leftarrow t_2.t_1$	$t_1 \leftarrow t_1.x_B$
$t_0 \leftarrow t_0.t_0$	$t_0 \leftarrow t_1.t_0$	$t_0 \leftarrow t_0.t_0$
$t_0 \leftarrow t_0.t_5$	$x_B \leftarrow \pi(f^{x^6})$	$t_0 \leftarrow t_0.t_1$
$x_B \leftarrow \pi(f^{x^6})$	$t_1 \leftarrow t_1.x_B$	return $t_0$
$t_2 \leftarrow t_2.x_B$	$t_0 \leftarrow t_0.t_0$	

The code sequence should be read top-to-bottom, then left-to-right. The final result of the exponentiation is stored in  $t_0$ . We can use this sequence and instruct a code generator to construct the final exponentiation code.

We tested this computation on the Magma On-line Calculator [IC] with the use of our online tutorial [Dom09] with a security level equivalent to AES-192 and the pairing computation takes approximately 73% less time than direct exponentiation for the hard part.

### 4.3.1 More Examples

In this section, we are applying the algorithm to other families of pairing-friendly elliptic curves, such as the curves mentioned in §2.5.

#### 4.3.1.1 The MNT curves

The MNT pairing-friendly elliptic curves were reported by Miyaji, Nakabayashi and Takano in [MNT01]. For the  $k = 6$  case, the prime  $p$  and the group order  $r$  parameters are:

$$p(x) = x^2 + 1;$$

$$r(x) = x^2 - x + 1;$$

$$t(x) = x + 1.$$

In this case, the hard part of the final exponentiation is to the power of  $(p^2 - p + 1)/r$ .

Substituting from Equation 2.11, the exponentiation is to the power of  $(x^4 + x^2 + 1)/(x^2 - x + 1)$  which is equal to  $x^2 + x + 1$ . Expressing this to the base- $p$ , it becomes simply  $(p + x)$ . The hard part of the final exponentiation can be reduced to:  $f^p \cdot f^x$ , which is an application of the Frobenius operator and a simple exponentiation. This is faster than Algorithm 8, as it requires three Frobenius operations, an inversion, and a multiplication by the remaining part of the final exponentiation.

#### 4.3.1.2 The BN Curves

The BN family of pairing-friendly curves from Barreto and Naehrig [BN06] has an embedding degree of 12, and is parametrised as follows:

$$\begin{aligned} p(x) &= 36x^4 + 36x^3 + 24x^2 + 6x + 1; \\ r(x) &= 36x^4 + 36x^3 + 18x^2 + 6x + 1; \\ t(x) &= 6x^2 + 1. \end{aligned}$$

In this case, the hard part of the final exponentiation is raising to the power of  $(p^4 - p^2 + 1)/r$ . After substituting the polynomials for  $p$  and  $r$ , this can be expressed to the base- $p$  as was already shown in Equation 4.2.

The BN family of curves are very plentiful, and to improve the speed of the Miller loop and the final exponentiation, we may prefer to choose  $x$  to have a low Hamming weight. We present a selection of low Hamming weight  $x$ -parameters in Table A.1 for different curves and subgroup sizes.

Next, we compute  $f^x$ ,  $f^{x^2} = (f^x)^x$ , and  $f^{x^3} = (f^{x^2})^x$ . These are simple exponentiations, and the low Hamming weight of  $x$  ensures that each requires a minimum of multiplications when using a simple square-and-multiply algorithm. We next calculate  $f^p$ ,  $f^{p^2}$ ,  $f^{p^3}$ ,  $(f^x)^p$ ,  $(f^{x^2})^p$ ,  $(f^{x^3})^p$  and  $(f^{x^2})^{p^2}$  using the Frobenius map.

Grouping the elements of the exponentiation together, the expression becomes:

$$[f^p \cdot f^{p^2} \cdot f^{p^3}] \cdot [1/f]^2 \cdot [(f^{x^2})^{p^2}]^6 \cdot [(f^x)^p]^{12} \cdot [f^x / ((f^{x^2})^p)]^{18} \cdot [1/f^{x^2}]^{30} \cdot [f^{x^3} \cdot (f^{x^3})^p]^{36}.$$

**Note:** An element  $\alpha$  is called “unitary” if it has a norm  $N_{\mathbb{F}_{p^k}/\mathbb{F}_{p^{k/d}}}(\alpha) = 1$  [Sco07]. After the easy part of the final exponentiation, the field element is unitary. An unitary element can be inverted by a simple conjugation, retaining this property subsequently [SB04].

The individual components between the square brackets are then calculated with just four multiplications (recalling that division costs the same as a multiplication), and we end up with a calculation of the form:

$$A \cdot B^2 \cdot C^6 \cdot D^{12} \cdot E^{18} \cdot F^{30} \cdot G^{36}.$$

Note that the exponents here are simply the coefficients that arise in the  $\lambda_i$  equations above. Again, we use the Dominguez Perez and Scott method to find an addition sequence<sup>3</sup>:

$$\{1, 2, \underline{3}, 6, 12, 18, 30, 36\}.$$

Note that 3 is the only member of the addition sequence which is not a member of the set of the original proto-sequence. This is an unexpected situation but it means less extra work for the computation.

This part of the calculation requires only nine multiplications and four squarings. We find this approach to the hard part of the final exponentiation for the BN curves to be about 4% faster than the rather ad-hoc method proposed by Devegili, Scott and Dahab [DSD07] (7156 modular multiplications/squarings over  $\mathbb{F}_p$  are required, compared to 7426 using the Nogami et al. recommendation of  $x = 2^{62} + 2^{55} + 1$  [NAS<sup>+</sup>08].)

---

<sup>3</sup>Although, this can be trivially constructed.

### 4.3.1.3 Freeman Curves

In [Fre06] a construction is suggested for pairing-friendly elliptic curves of embedding degree 10. The parameters for this family are as follows:

$$p(x) = 25x^4 + 25x^3 + 25x^2 + 10x + 3;$$

$$r(x) = 25x^4 + 25x^3 + 15x^2 + 5x + 1;$$

$$t(x) = 10x^2 + 5x + 3.$$

These curves are much rarer than the BN curves, and it is not feasible to choose  $x$  to have a particularly small Hamming weight. Nevertheless proceeding as above we find:

$$\lambda_3(x) = 1,$$

$$\lambda_2(x) = 10x^2 + 5x + 5,$$

$$\lambda_1(x) = -5x^2 - 5x - 3,$$

$$\lambda_0(x) = -25x^3 - 15x^2 - 15x - 2.$$

In this case the coefficients form a *perfect addition sequence*<sup>4</sup>:

$$\{1, 2, 3, 5, 10, 15, 25\}.$$

The vectorial addition sequence in this case requires 10 multiplications and 2 squarings.

---

<sup>4</sup>An addition sequence is perfect when its proto-sequence and optimal sequence contain the same elements.

#### 4.3.1.4 The KSS: $k = 8$ Family of Curves

The parameters for the family of  $k = 8$  KSS curves are as follows:

$$p(x) = (x^6 + 2x^5 - 3x^4 + 8x^3 - 15x^2 - 82x + 125)/180;$$

$$r(x) = (x^4 - 8x^2 + 25)/450;$$

$$t(x) = (2x^3 - 11x + 15)/15;$$

$$x \equiv 5 \pmod{30}.$$

For these curves  $\rho = 3/2$ . As in the case of the BN curves,  $x$  can be chosen to have a low Hamming weight. Proceeding as above we find:

$$\lambda_3(x) = (15x^2 + 30x + 75)/6,$$

$$\lambda_2(x) = (2x^5 + 4x^4 - x^3 + 26x^2 - 55x - 144)/6,$$

$$\lambda_1(x) = (-5x^4 - 10x^3 - 5x^2 - 80x + 100)/6,$$

$$\lambda_0(x) = (x^5 + 2x^4 + 7x^3 + 28x^2 + 10x + 108)/6.$$

Again, a minor difficulty arises due to the common denominator of 6, we evaluate the sixth power of the pairing. An addition sequence for this curve is the following:

$$\{1, 2, 4, 5, 7, 10, 15, \underline{25}, 26, 28, 30, \underline{36}, \underline{50}, 55, 75, 80, 100, 108, 144\}.$$

## 4.4 A Note on the Final Exponentiation

In this chapter, we have presented the initial work of Hu, Dong and Pei which makes use of the Frobenius operator. Also, the work of Devegili, Scott and Dahab was presented, which exploits the polynomial structure of the BN family of curves, and the Frobenius map.

Our joint work with Scott et al. not only exploited the polynomial representation of

the curve and the Frobenius map, but also uses addition chains to speed up the multi-scalar exponentiation of the hard part of the computation.

This last work can be combined with our work from Chapter 3 to easily generate the operations needed for the final exponentiation.

The only competitive final exponentiation is the one from Devegili et al. for the BN family of curves, used to compare our method (Scott et al.). Since that implementation only works for that family of curves, we cannot compare against other methods, excluding the Du, Hong and Pei method.

Unless a bad choice of an addition sequence is used, our method is faster. In our tests, our final exponentiation method is approximately 70% faster than direct exponentiation, and approximately 4-5% faster than the Devegili et al. final exponentiation method. The key difference, is that our method can be generalized to any family of pairing-friendly curves.

Future work on the final exponentiation should be build from the Devegili et al. method to avoid the use of excessive temporary space in the computation. For families of curves with larger addition sequences, such as the KSS curves with embedding degree  $k = 36$  which are suitable for very high security levels, an addition sequence would require a lot of temporary space. At the end of §3.6 we show a short sequence for this curve using the AIS method, the number of required  $t_i$  being reduced from 213 to 35 (plus 13 for pre-computation and two  $x_j$  for the  $\lambda$ 's in both cases) which is a good reduction. This is the first implementation at a very high security level.

One important thing to note is that as we go to higher security levels, the final exponentiation starts to take over the Miller loop in terms of computational costs. At lower security levels, the final exponentiation takes less time than the Miller loop, but over high security levels, the final exponentiation takes longer time. This also applies for exponentiation in the final  $G_T$  group.

Traditionally, the pairing-based protocols are designed with the assumption that one pairing computation takes as much time as 3 exponentiation on the final extension field. This ratio has to be revisited for high security levels, as the assumption is not valid there.



## 5 FAST HASHING TO $G_2$

*"The Irish do not want anyone to wish them well; they want everyone to wish their enemies ill." – Harold Nicolson*

IMPLEMENTING identity-based protocols using ordinary pairing-friendly elliptic curves requires two groups, at least one of which is to be of order  $r$ . After hashing an identity to a general point, a scalar multiplication of this point by the cofactor of the particular curve is required. If  $G_1$  is required to be of order  $r$ , the operation is quite simple, and is trivial for families of curves with  $\rho$ -value equal to 1 (or close enough).

If  $G_2$  is required to be of order  $r$  [SK03], the operation becomes much more complex. Thanks to the use of a twisted curve, the operation cost is not exorbitant, but still need to be addressed. In [SBC<sup>+</sup>09a] Scott et al. we suggest an algorithm which make use of some identities for enabling the implementer to use the cheap Frobenius endomorphism and the addition chain method.

As our method (Scott et al.) requires only the parameters of the curve it is possible to use the same approach as in Chapter 3. For this case, the addition sequence generated will be used to obtain addition and doubling operations on points. This can be done since the operation is to accelerate a scalar-point multiplication, similarly to an exponentiation, as described in Chapter 4. Once the addition chain has been obtained, it is easy to generate the code sequence for the cofactor operation.

The rest of this chapter is as follows: We introduce the traditional method for hashing to  $G_2$  in §5.1. §5.2 presents the formulae for the number of points in the twist curves. Our new method is presented at §5.3. Final thoughts on our method are presented in §5.4

## 5.1 Introduction

The traditional method to hash into a curve is the Try-and-Increment in which an element of  $\mathbb{F}_p$  is used as a parameter in the Weierstrass equation to test if a solution exists, increasing by one unit until we succeed. Heuristically, we have a probability of getting a point every  $2^{-k}$  inputs, with  $k$  the embedding degree [Ica09]. Icart [Ibidem.] presented a method that given any elliptic curve  $E$  defined over  $\mathbb{F}_{p^n}$ , maps elements of  $\mathbb{F}_{p^n}$  into  $E$  in deterministic polynomial time (if  $p^n \equiv 2 \pmod{3}$ .)

Some protocols based in the Weil and Tate family of pairings, may require a hash to this point to a prime order point in  $G_2$ . To achieve this, an additional multiplication by a large cofactor may be necessary.

This cofactor  $c = \#E'(\mathbb{F}_{p^d})/r$ , is the number of points on the curve where the group is defined and divided by  $r$ . In §5.2 we present a fast way to compute this cofactor.

## 5.2 Point Counting

Let  $E$  be an elliptic curve defined over  $\mathbb{F}_p$ . The number of points on the curve is defined as  $\#E(\mathbb{F}_p) = p + 1 - t$ , where  $t$  is the trace of the Frobenius.

Now, let's consider a curve over an extension field  $\mathbb{F}_{p^k}$ . Let  $L_1(T) = pT^2 - tT + 1$  with  $t = \alpha_1 + \alpha_2$  the trace of the  $p$ -th power Frobenius. Let  $\tau_k = \alpha_1^k + \alpha_2^k$  be the trace of the  $p^k$ -th power Frobenius, then the  $\tau_k$  are given by the recursion  $\tau_1 = t, \tau_2 = t^2 - 2p$  and

$$\tau_k = t \cdot \tau_{k-1} - p \cdot \tau_{k-2}$$

It follows that  $\#E(\mathbb{F}_{p^k}) = p^k + 1 - \tau_k$ . In Scott et al. [SBC<sup>+</sup>09a, Algorithm 1], we describe a simple algorithm to compute this value.

Let  $E$  be an elliptic curve defined over  $\mathbb{F}_p$ . Let  $\#E(\mathbb{F}_p) = p + 1 - t$ . Let the polynomial  $X^2 - tX + p$  factor as  $(X - \alpha)(X - \bar{\alpha})$  we have  $\alpha\bar{\alpha} = p$  and  $\alpha + \bar{\alpha} = t$ .

The number of point for a curve defined over  $\mathbb{F}_{p^2}$  is  $\#E(\mathbb{F}_{p^2}) = p^2 + 1 - (\alpha^2 + \bar{\alpha}^2) =$

$p^2 + 1 - (t^2 - 2p)$  since:

$$\begin{aligned} t^2 &= (\alpha + \bar{\alpha})^2 = \alpha^2 + 2\alpha\bar{\alpha} + \bar{\alpha}^2 \\ \alpha^2 + \bar{\alpha}^2 &= t^2 - 2\alpha\bar{\alpha} \\ &= t^2 - 2p. \end{aligned} \tag{5.1}$$

Similarly,

$$\#E(\mathbb{F}_{p^3}) = p^3 + 1 - (\alpha^3 + \bar{\alpha}^3) = p^3 + 1 - (t^3 - 3pt), \tag{5.2}$$

$$\#E(\mathbb{F}_{p^4}) = p^4 + 1 - (\alpha^4 + \bar{\alpha}^4) = p^4 + 1 - (t^4 - (4pt^2 + 2p^2)), \tag{5.3}$$

$$\#E(\mathbb{F}_{p^6}) = p^6 + 1 - (\alpha^6 + \bar{\alpha}^6) = p^6 + 1 - (6pt^4 - 9p^2t^2 + 2p^3). \tag{5.4}$$

The selection of these curves is based on the fact that curves of even embedding degree are preferred as they have twist of high degree, and that simplify the Miller loop as shown in Chapter 2.

We prefer to define  $G_2$  over the twist of a curve. Let  $E$  be an elliptic curve over  $\mathbb{F}_q$  with  $q = p^{k/d}$  and  $\#E(\mathbb{F}_q) = q + 1 - \tau$ . Let  $\tau = \alpha^{k/d} + \bar{\alpha}^{k/d}$  as in equations 5.1–5.4. If this curve admits a twist  $E'$  of degree  $d$ , then the group order of the twist curves are as follows: [HSV06]

$$\begin{aligned} d = 2 : \#E' &= q + 1 + \tau \\ d = 3 : \#E' &= q + 1 - (\pm 3F - \tau)/2 && \text{with } \tau^2 - 4q = -3F^2 \\ d = 4 : \#E' &= q + 1 \pm F && \text{with } \tau^2 - 4q = -F^2 \\ d = 6 : \#E' &= q + 1 - (\pm 3F + \tau)/2 && \text{with } \tau^2 - 4q = -3F^2 \end{aligned}$$

For example, let  $q = p^3$  and  $\tau = t^3 - 3pt$  then,  $\#E'(\mathbb{F}_q) = q + 1 - (3F + \tau)/2$ , with  $\tau^2 - 4q = -3F^2$ . Solving for  $F$ , one can easily calculate the number of points on the

twisted curve.

### Cofactor for the BN curves

As shown by Barreto and Naehrig in [BN06], the BN curves have a  $k = 12$  embedding degree and it supports a sextic twist. For this case,  $\#E'(\mathbb{F}_{p^2}) = (p + 1 - t)(p - 1 + t)$ , hence the cofactor is  $c = (p - 1 + t)$ . This presents a very nice form, which seems to be a coincidence.

Recalling Equations 2.13, let  $\tau = t^2 - 2p$  and  $d = 6$ , we have:

$$\begin{aligned}
 F &= 36x^4 + 24x^3 + 12x^2 + 4x + 1 \\
 \#E'(\mathbb{F}_{p^2}) &= p^2 + 1 - (-3F + \tau)/2 \\
 &= 1296x^8 + 2592x^7 + 3024x^6 + 2160x^5 + 1152x^4 \\
 &\quad + 432x^3 + 120x^2 + 24x + 4 \\
 c &= 36x^4 + 36x^3 + 30x^2 + 6x + 1
 \end{aligned} \tag{5.5}$$

which is equal to  $(p - 1 + t)$ .

### Cofactor for the KSS: $k = 18$ curves

For the KSS curves with  $k = 18$ , the cofactor is:

$$\begin{aligned}
 c &= (40301641 + 88845918x + 82554234x^2 + 50075833x^3 + 30860595x^4 \\
 &\quad + 19465362x^5 + 9658007x^6 + 4055484x^7 + 1803684x^8 + 757927x^9 \\
 &\quad + 256908x^{10} + 81660x^{11} + 27539x^{12} + 7929x^{13} + 1791x^{14} + 409x^{15} \\
 &\quad + 96x^{16} + 15x^{17} + x^{18})/27
 \end{aligned} \tag{5.6}$$

We could continue to present the cofactors for other useful families of elliptic curves. However, we prefer to leave this as an exercise to the reader. Please note that since we need to get the square root of  $F$ , we will get two different co-factors, one from the positive

solution of the square root, and one from the negative one.

### 5.3 A new method

When calculating cryptographic pairings, at least one of the parameters has to be of order  $r$ . In general, the Tate pairing family takes a point from  $G_2 \subseteq E'(\mathbb{F}_{p^d})[r]$ .

Traditionally, for hashing a point  $G_2$  of order  $r$ , the standard method is to find a general random point in  $G_2$  and then multiply by the co-factor  $c = \#E'/r$ , when using a twist of a curve (which can be easily calculated from the previous section). Hashing to  $G_2$  instead of  $G_1$  is more expensive since we have a larger co-factor of the curve, even on a twist of a curve.

For hashing a point into  $G_2[r]$  one can use the efficiently computable homomorphism:  $\psi^i = \phi^{-1}\pi_p^i\phi$  from §2.1.5, and the addition-chain method to speed-up the multi-scalar-point operation.

The isomorphism  $\phi$  is defined in §2.1.5. One way to compute this map is by pre-computing  $\delta^2$ ,  $\delta^3$  and their inverses ( $\phi^{-1}$ ).

The homomorphism to be exploited is  $\psi$  from [GS08, Lemma 1]. The points in  $G_2$  obey the identity [GS08, Section 8]:

$$\psi^2(P) - [t]\psi(P) + [p]P = 0.$$

The main idea is to first express the co-factor  $c$  to the base  $p$  as:

$$c = c_0 + c_1 \cdot p + c_2 \cdot p^2 \dots$$

and then use the identity

$$[p]P = [t]\psi(P) - \psi^2(P), \tag{5.7}$$

repeatedly if necessary, to reduce the co-factor multiplication to a form

$$[c_0 + c_1 \cdot p + c_2 \cdot p^2 + \dots]P = [g_0]P + [g_1]\psi(P) + [g_2]\psi^2(P) + \dots \quad (5.8)$$

where all of the  $g_i < p$ .

One must be aware that a useful equivalence in some scenarios is:  $\psi^{\frac{k}{2}}(P) = -P$ . Since  $\psi$  is an endomorphism on  $E'$ , for every  $i \in \mathbb{Z}$  we have  $\psi^i(P) = [\lambda_i]P$ , for some integer  $\lambda_i$ .

As  $\psi^i = \phi^{-1}\pi^i\phi$ , we know that  $\psi^k(P) = P$ , since the  $k$ -th power of the Frobenius is the identity on  $\mathbb{F}_{p^k}$ . If  $\psi^{\frac{k}{2}}(P) = \lambda_{\frac{k}{2}}P$ , we have  $P = \psi^k(P) = \psi^{\frac{k}{2}}(\psi^{\frac{k}{2}}(P)) = \lambda_{\frac{k}{2}}^2P$ , so  $\lambda_{\frac{k}{2}}^2 = 1$ .

It is not true that  $\lambda_{\frac{k}{2}} = 1$ , as this would imply that  $P = \psi^{\frac{k}{2}}(P)$  and  $\psi^{\frac{k}{2}}(P) = \phi^{-1}\pi^{\frac{k}{2}}\phi(P)$ , or equivalently  $\phi(P) = \pi^{\frac{k}{2}}(P)\phi(P)$ , i.e., that the  $k/2$ -th power of Frobenius is the identity on  $\mathbb{F}_{p^k}$ , which it is not.

**Example** For the KSS:  $k = 18$  curves, a sextic twist applies. The co-factor in this case is shown in Equation 5.6.

To exploit formula (5.8), we express the co-factor  $c = \#E'(\mathbb{F}_{p^3})/r$  in base  $p$  as:

$$\begin{aligned} c = & (49/3x^2 + 245/3x + 343/3) \cdot p^2 \\ & + (7/3x^6 + 35/3x^5 + 49/3x^4 + 112/3x^3 + 581/3x^2 + 784/3x) \cdot p \\ & + (-5/3x^7 - 26/3x^6 - 98/3x^5 - 127x^4 - 289x^3 - 637x^2 - 1715x - 5774/3) \end{aligned} \quad (5.9)$$

One may prefer to get rid the denominator 3, and instead complete the hashing as

$[3.c(x)]P$ . Let's define <sup>1</sup>:

$$a(x) = 49x^2 + 245x + 343$$

$$b(x) = 7x^6 + 35x^5 + 49x^4 + 112x^3 + 581x^2 + 784x$$

$$c(x) = -5x^7 - 26x^6 - 98x^5 - 381x^4 - 867x^3 - 1911x^2 \\ - 5145x - 5774$$

We know from (5.7) that:

$$[b(x).p(x)]P = b(x).t(x)\psi(P) - b(x)\psi^2(P)$$

Again, from (5.7), we have:

$$[p^2]P = [t^2]\psi^2(P) - [2t]\psi^3(P) + \psi^4(P)$$

---

<sup>1</sup>after the multiplication by 3 in formula (5.9)

Let's define:

$$d(x) = c(x)$$

$$e(x) = b(x).t(x)$$

$$f(x) = a(x).t(x)^2 - b(x)$$

$$g(x) = (-2.a(x).t(x))$$

$$h(x) = a(x)$$

$$i(x) = e(x) \bmod p(x)$$

$$j(x) = \left\lfloor \frac{e(x)}{p(x)} \right\rfloor$$

$$k(x) = f(x) \bmod p(x)$$

$$l(x) = \left\lfloor \frac{f(x)}{p(x)} \right\rfloor$$

and,

$$n(x) = a(x).p^2 + b(x).p + c(x)$$

After a few substitutions, from Scott et al. [SBC<sup>+</sup>09a, algorithm 2] one gets the following formula:

$$\begin{aligned} [n(x)]P &= d(x).P + i(x).\psi(P) + (j(x).t(x) + k(x)).\psi^2(P) \\ &\quad + (l(x).t(x) - j(x) + g(x)).\psi^3(P) + (h(x) - l(x)).\psi^4(P) \end{aligned} \quad (5.10)$$

Let's apply (5.10) with  $[3.c(x)]P$ , where  $P \in G_2$  and  $c(x) = \#E'(\mathbb{F}_{p^3})/r(x)$ . This



yields:

$$\begin{aligned}
[3.c(x)]P &= (-5x^7 - 26x^6 - 98x^5 - 381x^4 - 867x^3 - 1911x^2 - 5145x - 5774).P \\
&+ (-5x^7 - 18x^6 - 38x^4 - 323x^3 - 28x^2 + 784x).\psi(P) \\
&+ (-5x^7 - 18x^6 - 38x^4 - 323x^3 + 1029x + 343).\psi^2(P) \\
&+ (-11x^6 - 70x^5 - 98x^4 - 176x^3 - 1218x^2 - 2058x - 686).\psi^3(P) \\
&+ (28x^2 + 245x + 343).\psi^4(P)
\end{aligned}$$

For the above computation, we can use the following addition-chain:  $\{\underline{1}, \underline{2}, \underline{3}, 5, \underline{7}, \underline{8}, 11, 18, 26, 28, \underline{31}, 38, \underline{45}, \underline{69}, 70, \underline{78}, 98, 176, 245, \underline{253}, 323, 343, 381, \underline{389}, 686, 784, \underline{829}, 867, 1029, 1218, \underline{1658}, 1911, 2058, \underline{4116}, 5145, 5774\}$  [SBC<sup>+</sup>09a]. The computations now becomes as follows<sup>2</sup>:

$x_A \leftarrow -P \cdot x.$	$t_2 \leftarrow t_0 + t_2.$	$t_4 \leftarrow t_7 + t_4.$
$x_B \leftarrow -P.$	$t_4 \leftarrow t_2 + \psi^3(-P).$	$t_3 \leftarrow t_4 + t_3.$
$t_1 \leftarrow x_A + x_B.$	$t_2 \leftarrow t_2 - P \cdot x^5 + \psi^3(-P \cdot x^4).$	$t_4 \leftarrow t_4 + \psi^4(P \cdot x^2) +$
$t_0 \leftarrow \psi^2(P \cdot x) + x_A.$	$t_4 \leftarrow 2 \cdot t_4.$	$\psi(-P \cdot x^2).$
$t_1 \leftarrow 2 \cdot t_1.$	$t_4 \leftarrow t_4 + \psi^2(P) + \psi^4(P).$	$t_3 \leftarrow t_3 + t_7.$
$t_1 \leftarrow t_1 + \psi^3(-P \cdot x).$	$t_6 \leftarrow -P \cdot x^4 + x_B.$	$t_7 \leftarrow t_7 + t_2.$
$t_1 \leftarrow 2 \cdot t_1.$	$t_4 \leftarrow t_6 + t_4.$	$t_1 \leftarrow t_3 + t_1.$
$t_1 \leftarrow t_1 + t_0.$	$t_6 \leftarrow t_6 + t_3.$	$t_6 \leftarrow t_1 + t_6.$
$t_0 \leftarrow -P \cdot x^2 + x_B.$	$t_1 \leftarrow t_4 + t_1.$	$t_6 \leftarrow t_6 + t_0.$
$t_0 \leftarrow 2 \cdot t_0.$	$t_4 \leftarrow t_4 + t_2.$	$t_1 \leftarrow t_6 + t_1.$
$x_B \leftarrow \psi^3(-P \cdot x^2).$	$x_C \leftarrow \psi(-P \cdot x^3) +$	$t_6 \leftarrow t_6 + t_0.$
$t_0 \leftarrow t_0 + x_B.$	$\psi^2(-P \cdot x^3).$	$t_0 \leftarrow t_1 - P \cdot x^6.$
$t_2 \leftarrow t_1 + \psi(P \cdot x).$	$t_2 \leftarrow x_C - P \cdot x^2.$	$t_1 \leftarrow t_1 - P \cdot x^7 +$
$t_1 \leftarrow t_1 + \psi^4(P \cdot x).$	$t_3 \leftarrow \psi^3(-P \cdot x^5) + x_C.$	$\psi(-P \cdot x^7) +$
$t_0 \leftarrow t_0 - P \cdot x^3.$	$t_1 \leftarrow t_2 + t_1.$	$\psi^2(-P \cdot x^7).$
$t_3 \leftarrow \psi(-P \cdot x^4) +$	$t_2 \leftarrow t_2 + x_B.$	$t_0 \leftarrow t_0 + t_4.$
$\psi^2(-P \cdot x^4) - P \cdot x^3.$	$t_7 \leftarrow t_1 + \psi^3(-P \cdot x^3).$	$t_2 \leftarrow t_0 + \psi(-P \cdot x^6) +$

<sup>2</sup>read bottom-to-top, then left-to-right, separate pages are read independently, but are a continuation.

$$\begin{array}{l|l|l}
\psi^2(-P \cdot x^6). & t_0 \leftarrow t_0 + t_5. & t_0 \leftarrow t_0 + t_2. \\
t_0 \leftarrow t_0 + t_7. & t_1 \leftarrow t_2 + t_1. & t_0 \leftarrow 2 \cdot t_0. \\
t_5 \leftarrow t_2 + \psi^3(-P \cdot x^6). & t_2 \leftarrow t_2 + t_4. & t_0 \leftarrow t_0 + t_1. \\
t_2 \leftarrow t_2 + t_6. & t_0 \leftarrow t_1 + t_0. & \text{return } t_0. \\
t_2 \leftarrow t_2 + t_5. & t_1 \leftarrow t_1 + t_3. & \\
t_0 \leftarrow 2 \cdot t_0. & t_0 \leftarrow 2 \cdot t_0. & 
\end{array}$$

where  $t_0$  is  $[3.c(x)]P$ .

We tested this computation in the Magma On-line Calculator [IC] using our online tutorial [Dom09]<sup>3</sup> with a security level equivalent to AES-192, and it takes approximately 35% less time than the traditional method described in §5.1.

## 5.4 Final Thoughts

Before our algorithm was devised, there were only generic methods to hash a point to  $G_2$  which did not exploit any property of the curves at all, such as: sliding windows, NAF or the binary method (double-and-add). For some cryptographic protocols at high security levels (larger embedding degree) this was a problem.

The larger the embedding degree, the larger the  $G_2$ , perhaps this also depends on the supported twist of the curve. In any case, the larger the  $G_2$ , the larger the co-factor, the coefficient used to make the point of the desired order. The larger the co-factor, the more expensive it is to hash a point.

Our method is a solution that reduces the cost of hashing a random point to a specific group exploiting the structure of the co-factor. In our method, we exchange a expensive scalar-point multiplication by some smaller scalar-point multiplications and some map applications, making this operation cheaper.

With the use of an addition sequence, our method defines a sequence of operations. This sequence, is particularly long for the KSS family of curves with embedding degree  $k = 36$ . Using our addition chain solver, we were able to get a sequence  $s$  with  $\#s \approx 780$

<sup>3</sup>For details of our tutorial, please refer to Chapter C.

elements. The search space is quite large for this particular case; however, it is possible to get a shorter chain with more computing resources.

The KSS:  $k = 36$  family of curves supports a sextic twist, hence  $E'$  is defined over  $\mathbb{F}_{p^6}$ . This curve is suitable for a security level of 256-bit. The co-factor for such curve is quite large, and hashing will be very expensive, taking several milliseconds. Using our method, we can reduce the cost of this operation, however we are still in the search of a shorter sequence.

Hashing to  $G_2$  is an operation required at the initialization of the pairing function. This operation is not used during the pairing computation, but required to get a bilinear pairing. Its use depends on the pairing-based protocol.

## 6 CODE GENERATOR

*"I am a drinker with a writing problem." – Brendan Behan*

A cryptographic compiler is meant to generate program code that performs certain cryptographic functions [BMP09, PSM06]. In the case of hardware it depends on the requirements, but there are already compilers which translate programs into the machine code specific to the hardware [Com94]. In this research, the target is a computer program and, indirectly, a multi-precision library. There is therefore a last step that it is usually conferred to the ordinary compiler, which includes the traditional code generator part.

A cryptographic compiler is needed because there are many issues in the implementation of either a protocol or a function, which are added during the process of translating the mathematical background to the implementation itself.

The compiler makes use of several metaprogramming methods, which are covered in §2.8, for instance:

- **Attribute-Oriented Programming** [PP82, §6.5].
- **Reflective Programming** [Smi82].
- **Automatic code generation** [ASU86, 9].
- **Template-based Metaprogramming.** [DD01, 3.2.1]

There are already some build tools in the market: e.g., make, gnumake, nmake, jam, etc. These tools are inherently shell-based. One can extend these tools by using or writing any program for the specific operating system they are currently lying on. Other tools such as Ants do a better job on writing programs by using XML files. All these tools are very useful, but it is preferred to have all the testing code and code sequence generators in the

same line. The compiler uses the techniques listed above, plus a few shell scripts. Instead of using XML files, the compiler uses associative arrays, as the complexity of the options available can be properly handled this way.

## 6.1 Instruction Construction

Any serious computer language has support for the basic arithmetic functions. Every serious multi-precision library designed for ECC has support for the basic set of functions for elliptic curve and finite field arithmetic. The interfaces of these operations in every multi-precision library are multifarious as they depend on the taste of the developer, or on previous decisions in the design of the library. It is necessary to devise specific methods to handle all of the Elliptic Curve (EC) operators.

To support a multi-precision library, a set of parameters which are loaded as separate files were defined. Depending on the multi-precision library, some parameters may not be needed; for example: an inverse operation may be done by prefixing a *minus* symbol, whereas in other, a function call may be required.

To define the position of the affix operators in the library, not only the operation is defined in a variable, but also the position of the operators and operands are defined as another variable. For example, for a named *operation*, it is needed to define an additional *operationPOS* variable, where “POS” stands for position, and its values ranges as:

- 0 for prefix: i.e.,  $-P$
- 1 for suffix: i.e.,  $P \wedge (-1)$
- 2 for infix: i.e.,  $P + = Q$
- 3 for circumfix: i.e.,  $\text{inverse}(P)$
- 4,5 and 6 are special cases
- 7 for transfix: i.e.,  $\text{psi}(A, d2, d3, 6)$  (here, A and 6 are the operands)

following by the required variables: *operationMAP*, *operationMAP2*, etc., which contains the sequence of characters of the corresponding operation in the library.

To handle the different operators and notations from different languages, a set of “Generic” functions that requires, as part of the regular parameters (the LH and RH operands), the operators to be used and their notation were defined. For example, Listing 6.1 defines an assignment operation between 2 operands, the LH- and RH- operand, and their operators. The *operatorPOS* parameter has the following values: “0” for Polish notation, “1” for Reversed Polish notation, “2” for infix notation, “3” for circum-flix notation, and “4” is a special notation to be used when one needs to convert a number in string format. Cases “5” and “6” are the special cases when the RH operand has to be placed on the left (and the LH correspondingly), i.e. in the use of C++ data objects.

Listing 6.1: Code in Magma for a Generic Assignment

---

```
// Input <- LH, RH, notation, operator(s), library definitions
// Output -> Corresponding output code to assign Left <- Right
// Example Input: GenericAssign("fml", "c", lang[tgtRATE] ["powerPOS"], "
    powerMAP", "powerMAP2", "RATEmiracl", lang)
// Example Output: pow(fml, c)
GenericAssign:=function(Left,Right,operatorPOS,operator1,operator2,
    target,language)
    Code:="";
    case operatorPOS:
        when 0:
            Code:=Sprintf("%o_%o_%o", language[target] [operator1], Left, Right
        );
        when 2:
            Code:=Sprintf("%o%o%o", Left, language[target] [operator1], Right);
        when 3:
            Code:=Sprintf("%o%o, %o%", language[target] [operator1], Left,
            Right, language[target] [operator2]);
        when 4:
            Code:=Sprintf("%o%o%o%o", Left, language[target] [operator1], Right
            , language[target] [operator2]);
        when 5:
            Code:=Sprintf("%o%o%o", Left, language[target] [operator1], Right);
        when 6:
            Code:=Sprintf("%o%o%o%o", Right, language[target] [operator1], Left
            , language[target] [operator2]);
    end case;
    return Code;
end function;
```

---

Similarly, Listing 6.2 presents the Magma code for defining a *generic* operation.

Listing 6.2: Code in Magma for a Generic Operation

---

```
// Input  <- LH, RH operands, notation, operator(s), library
//          definitions
// Output -> Corresponding output code for Left <- Right (.) Right2
// Example Input: GenericOperation(tk,Uno,Dos,lang["SUPmiracl"] ["
//                  MultiplyPOS"], "MultiplyOP", "MultiplyOP2", "*"), "SUPmiracl", lang),
//                  lang["FEmiracl"] ["delimiter"])
// Example Output: t1=slope*X;
GenericOperation:=function (Left,Right,Right2,operatorPOS,operator1,
operator2,target,language)
Code:="";
case operatorPOS:
when 0:
Code:=Sprintf("%o_%o_%o_%o", language[target] [operator1],Left,
Right,Right2);
when 3:
Code:=Sprintf("%o%o,%o,%o%", language[target] [operator1],Left,
Right,Right2, language[target] [operator2]);
when 4:
Code:=Sprintf("%o%o%o%o%", Left, language[target] [operator1],
Right, language[target] [operator2], Right2);
when 5:
Code:=Sprintf("%o%o%o%o%", Left, language[target] [operator1],
Right,Right2, language[target] [operator2]);
when 6:
Code:=Sprintf("%o%o,%o,%o%", language[target] [operator1],Left,
Right,Right2, language[target] [operator2]);
end case;
return Code;
end function;
```

---

For example, to generate `w1=(char *) "60300000000000E30"` one can use the following instruction:

```
GenericAssign(Sprintf("w\%o",i), w2i, lang[tgtRATE]
["assignBPOS"], "assignBMap", "assignBMap2", tgtRATE, lang);
```

which is a call from Listing 6.1. Suppose `w2i= 0x60300000000000E30`. Here, the “Sprintf” part selects which  $w_i$  to assign, the `lang[tgtRATE][“assignBPOS”]` contains the instructions on how to store a value into a “Big” number (a multi-precision integer storage location), the “assignBMap” and “assignBMap2” denotes the cast operation “(char \*)” and the apostrophes respectively, which are required by the given “assignBPOS” information.

The `tgtRATE` and `language` are the context environment: the multiprecision library and the code section.

In the case of Listing 6.2, to produce  $m1P = c * m2P$ , which is a scalar-point multiplication of a scalar  $c$  (from the R-ate pairing) and  $m2P$  a point in  $E'(\mathbb{F}_{p^d})$ , one can use the following instruction:

```
GenericOperation("m1P", "c", "m2P", lang[tgtRATE] ["scalar-point
POS"], "scalar-pointMAP", "scalar-pointMAP2", tgtRATE, lang)
```

A more complex example is the following:

```
GenericOperation(tk, Uno, Dos, lang[tgtSUP] [POS], Sprintf("%oOP"
, Operators[j] [2]), Sprintf("%oOP2", Operators[j] [2]), tgtSUP,
lang)
```

where  $tk \leftarrow \text{Uno} \circ \text{Dos}$ , and  $\circ$  is a parametrized operator.

There is an evident similarity between the previous listings: the difference lies in the number of RH-operators. Internally, they behave similarly, it is just the order of the operands and the operators.

As explained by Muchnick [Muc97], when there is a necessity to target several architectures, one may prefer to automatically generate the code generator. In this case, at runtime we know which multiprecision library we would like to generate the code for. It should be easy to generate a more compact and faster version of the `GenericAssign` and `GenericOperation` functions listed above for the specific notation of the target library, but this would be unnecessarily complex.

In both Listings 6.1 and 6.2 a string containing a variable number of operands and operators is constructed based on the number and order of the parameters. We can parametrise the parameters of the code construction but it is expected that the speed gain at runtime would be marginal. Additionally, the cost of constructing such an *automatic generator of code generator*, as Muchnick calls it, would not give the expected gain in terms of the speed of the output code.



Other functions that work with this approach are Normal and GenericTransformation, which are unary functions to Normalize a point on a curve, and to define a generic unary operator.

## 6.2 Attribute-based construction

Another method for code generation used in the compiler is Attribute Code Generation. For this method, the compiler uses “tags”, labels around the code, which are instructions in a meta language for the compiler.

The instructions can be freely chosen. For this research I defined tags for the following operations:

- Assignment.
- Constructing a tuple of variable size in a function call.
- Constructing the Line function of the Miller loop.
- Removing (unnecessary) parts of code.

There are two stages at which the compiler is looking for tags. One stage is the extension field construction dependent code. For this, a function constructs the code depending on how the tower of extension fields was constructed, and searches for the corresponding tags in the code.

The second stage is in the so-called re-writer facility. The re-writer is invoked when we need to substitute basic assignment tags in a specific source code file, as noted in the example 2.17 on page 44. The re-writer facility is also used to rename a template file with the corresponding notation. The notation for the generated output files are in Table 6.1.

The files in Row 1 and Row 2 from Table 6.1 are generated by an automatic code generator. The code sequences for these rows are explained in Chapters 4 and 5 respectively.

In the case where the family of curves supports a bilinear and non-degenerate R-ate pairing, the compiler will make use of the *filename* in Row 3 from Table 6.1. This file re-

#	File name	Description
1	FE c <i>CURVE 1 LIBRARY .extension</i>	Used for the final exponentiation of the Miller loop
2	G2 c <i>CURVE 1 LIBRARY . extension</i>	Used to hash a point into a group $G_2$ of order $r$
3	RATE c <i>CURVE 1 LIBRARY . extension</i>	Used for the R-ate pairing, if available
4	TestPairing c <i>CURVE 1 LIBRARY . extension</i>	Test bed for the pairing code
5	<i>LIBRARY x CURVE . extension</i>	Constructions that depends on the Elliptic Curve
6	<i>LIBRARY t D.MT.K . extension</i>	Constructions that depends on the towering method

Table 6.1: Filename Notation

quires a different method of construction using a hand-crafted sequence and the instruction constructor of the automatic code generator. This relies on the system setup consisting of the curve family, extension field construction and the  $x$ -parameter. If the selected pairing function is the pairing lattice, the ate or the Tate pairing, then the name of the file is changed accordingly.

The files in Rows 4 and 5 are hand-made by the user but require the re-writer. The code will present tags for the attribute-based code generator. Finally, the filename in Row 6 contains code that relies on the towering construction method. It follows the same approach as for Row 3.

### 6.3 Reflective program construction

In the present work the reflective programming paradigm was applied in a limited way, as a way to test the output code as part of the compiler. The Magma language, has the Attach/Detach facility where it compiles and loads the input code into memory. When a piece of code is not giving the desired results (for example: giving a point of the wrong order), we “detach” selected parts of the code and regenerate them. Again, trying to obtain either faster code, or code with smaller footprint, or even both, could be the goal of a code generator. We can also time the code in Magma and select the best option.

Another way to implement the Attach/Detach facility in Magma is with the use of the “load” facility. This facility cannot be used inside loops or inside external files (nested

loads); hence it is typically not what one would need from a compiler if the size of the code makes it unmanageable for the programmer to have it as a single file.

As mentioned before, with reflective programming one should be able to alter specific parts of the code to improve them. The size of the generated code at each stage is small enough to be handled by the currently available computational resources, as the cryptographic compiler generates the pairing function code in segments, which are not necessarily interrelated<sup>1</sup>. As a result, the computational cost to reproduce the code from scratch is minimal compared to the complexity of a granular modification of the generated code segment.

Listing 6.3: Code in Magma for testing the Fast Cofactor function

---

```

deltas:=[];
V:=Name(Fpk,1);
Append(~deltas,V^2);
Append(~deltas,V^3);
Append(~deltas,1/deltas[1]);
Append(~deltas,1/deltas[2]);
R:=Random(ExtT);
Q:=HashG2(R,z,deltas);
if (not IsZero(r*Q)) then
    repeat
        print "\n***Error!***\nRebuilding_fast_cofactor...\n";
        Detach("fasthashing.magma");
        FastCofactor(...);
        Attach("fasthashing.magma");
        Q:=HashG2(R,z,deltas);
    until (IsZero(r*Q));
    printf "Corrected\n";
end if;

```

---

In Listing 6.3 a loop that verifies the order of a point  $Q$  is presented. Here, we have implemented the Scott et al. fast hashing to  $G_2$  [SBC<sup>+</sup>09a] method which takes a Random point  $R$ , the  $x$ -parameter of the curve, and the deltas required by the  $\psi$  function<sup>2</sup> to obtain a point  $Q \in G_2$ . For more details, please refer to Chapter 5. The program continues Attaching/Detaching the code in memory until a Fast Cofactor construction is effective. In general, there are two co-factors from the positive and negative square root in the formula to

---

<sup>1</sup>One can share memory between the code generated. The compiler outputs separate functions for different logical operations, it is up to the implementer to integrate the (logical) functions to recycle the memory usage.

<sup>2</sup>see §2.1.5

get the number of points of the curve, in theory we only run this twice at most, however, we may be looking forward to get a different addition sequence, which would yield a different code sequence. Internally, the FastCofactor function tries to follow a different sequence of code to generate every time as a new addition sequence is provided.

## 6.4 Template-based construction

In the template-based construction, the compiler uses an offline construction, but with the advantage that all of the data types behave similarly; all of them are designed to do elliptic curve and finite field arithmetic.

The code generator needs to know the pattern of the naming of the finite field structures of every multi-precision library we are supporting. In the case of the MIRACL library, the field arithmetic is defined as, for example,  $\text{ZZn2}$ ,  $\text{ZZn3}$ ,  $\text{ZZn4}$  for arithmetic in the fields over  $p^2$ ,  $p^3$  and  $p^4$  respectively. For RELIC, the finite field arithmetic is defined using  $\text{fp2}$ ,  $\text{fp3}$ ,  $\text{fp4}$  for the same fields. We therefore use  $\text{ZZ}$  and  $\text{fp}$  for the fields.

For the finite field extensions, following the Benger and Scott paper [BS09], we can observe that in all the cases, there is only one intermediate field required between the field for the twist and the extension field. We can safely define  $\text{ZZnD}$ ,  $\text{ZZnMT}$  and  $\text{ZZnK}$  as labels for the fields we are using in the implementation, one for the twist, one for the named field, and one for the extension field respectively.

The field in the middle is the field extension constructed as a tower that lies between the extension field of the twist of the curve and the extension field required by the embedding degree, if any. This is: for the BN curves, the tower method is described as  $1-2-4-12$ , we set a quadratic extension on top of a quadratic extension, and we end up with a cubic. This is the default construction. In our example, the “MidTower” is 4. Any final extension field requiring tower uses at most one intermediate tower, we safely named this as mid-tower. This extension field does not affect the construction of the pairing function itself, but may affect the construction of functions around it. The details of its use are out of the scope of this work, but detection of this value is done to facilitate the construction of code

templates.

The compiler uses the re-writer facility after the code is finally generated, it replaces the finite field generic variables with their corresponding variables from the multi-precision library.

Consider the following example:

$$\#include "zznK.h" \tag{6.1}$$

This generic code (6.1) will be replaced, for the case of the MIRACL library with

$$\#include "zzn12.h" \tag{6.2}$$

in the case of the BN curves, as  $k = 12$ . The same approach is used for the variable definitions, function calls, R-ate pairing, etc.

## 6.5 Adding support to a library

To add support for another library, or a set of functions of finite field arithmetic, it is necessary to replicate any of the `languageLANGUAGE.magma` supplied files.

A language file contains 5 sections: the first section contains definitions for generating the code to hash to  $G_2[r]$ ; the second sections is for the final exponentiation; the third sections is for the pairing function, with emphasis on the R-ate pairing; the fourth section contains definitions for the construction of supportive functions, such as instructions on how to get the individual components of a composed number in a finite field, how to perform a Frobenius exponentiation, or how to leave a comment in the code; the last section contains the name of the files defining the finite field arithmetic for every supported finite field extension in the library we are adding support.

### 6.5.1 Hashing to $G_2[r]$

The operations for the generation of hashing to  $G_2$  code are:

Table 6.2: Definitions for  $G_2$

Tag	Description
"element"	Variable used to identify a point in $G_2$
"inversionID"	How to negate a point
"inversionPRECOMPUTE"	Do we need to pre-compute the inverses?
"DegreeOperatorPOS"	$x^i \cdot Q$
"ElementOperator"	P+Q
"assignOperator"	a:=b,a=b
"delimiter"	","
"MapPOS","Map"	$\psi^i$
"MapTempREQ"	Do we need to store intermediate values?
"datatypeF"	ZZn,fp
"datatypeC"	ECn,ec
"datatypef"	zzn,fp
"datatypeec"	ecn,ec
"datatypeB"	Big,bn
"datatypeSUFFIX"	.t
"CodeEXT"	cpp,c,cxx
"HeaderEXT"	h,hxx
"include4Code"	.h file
"include4Header"	header libraries
"header"	Function parameters
"footer"	How to close the function
"begin"	How to start a code block
Continued . . .	

Table 6.2: (continued)

Tag	Description
"end"	How to end a code block
"trinomialSUP"	Do we support several operators per operation?
"binomialSUP"	Do we support inline operations?
"NormalREQ"	Do we need to normalize a point?
"accumulatorOperator"	How to accumulate a value
"m2REQ","dREQ","xREQ"	Do we need to setup the values in the code?
"B-in-HexREQ"	Do we need hexadecimal representation?
"try-and-catchREQ"	Do we need to enclose de code inside a try-and-catch block?
"preInitREQ"	Do we need to initialize the memory prior to use?
"extra-try-and-catchREQ"	Do we need to zero the variables prior to use?
"preExitREQ"	Do we need to explicitly free the memory?
"LOOPLENGTH"	Tag for the loop length
"LibraryID"	Name of the library
End of $G_2$ parameters definition	

### 6.5.2 Final exponentiation

The final exponentiation generation code share the same definitions (with different values) and the following information in table 6.3

Table 6.3: Definitions for the final exponentiation

Tag	Description
"arrayStartsat0"	Self explanatory
"returnArray"	Are we returning elements in array?
Continued . . .	

Table 6.3: (continued)

Tag	Description
"BigDivisionOperator"	Support for big number division
"arrayassignPOS"	Bidimensional array assignment
"varSeparator"	Self explanatory
End of final exponentiation parameters definition	

### 6.5.3 The pairing function

The pairing function also share some definitions, but with different values. It also includes the definitions shown in table 6.4

Table 6.4: Definitions for the pairing function

Tag	Description
"scalar-pointPOS"	How to do a scalar-point multiplication.
"pointAddPOS"	How to add two points
"powerPOS"	How to exponentiate
"GPOS"	How to call the line function
"qpowFrobPOS"	The $p$ -power Frobenius map (deprecated in favor of psi function)
"for1toMAP"	How to construct a for loop from 1 to a fixed parameter
"testifFailedPOS"	Test if the operation failed
"testifNotOKPOS"	Test if an OK is not returned
"IFPOS"	How to do a simple IF conditional
"IFcloseREQ"	Do we need to close a one line IF conditional?
"returnFALSE"	Self explanatory
Continued . . .	



Table 6.4: (continued)

Tag	Description
"returnTRUE"	Self explanatory
"SoftExpoPOS"	How to call the easy part of the final exponentiation
"HardExpoPOS"	How to call the hard part of the final exponentiation
"compiler"	Code compiler and parameters
"Dependencies"	Library dependencies for compile the code in the target multi-precision library
End of pairing function parameters definition	

#### 6.5.4 Others

Another set of definitions establishes the way to set and retrieve specific components of an element of a field, exponentiate, multiply, add, subtract and negate elements in a field. Finally, another set of definitions sets the required libraries to construct an extension field.

#### 6.5.5 Linking the library to the project

Once the definition of the new library is ready, open `autopairing.magma`, add a line to load the new language file next to the other language definition files. The `compileDirectory` variable will need to be set during the execution of the code generator if one would like to compile the output code.

### 6.6 Parameters of the program

The code generator has the following default options:

- `curve="6.8.1"` See [FST10] for the example curve<sup>3</sup>. For other values consult `lam-bda.magma`.

<sup>3</sup>The extra 1 in the notation refers to the "negative" version of the curve.

- $M=100$ . The number of iterations/generations for the repetitive code
- $G=5$ . The number of clones for the repetitive code; it falls-back to 5 if less than 2.
- `target="miracl"`. The multi-precision library. Other values are "relic".
- `level=128`. The equivalent security level. Other values are 80, 192, 256.
- `MAXUNROLLOOP=6`. Trigger set to unroll code.
- `XValueGenerator="Random"`. This generates an  $x$ -parameter of the form:  $2^i + 2^j + c$ , otherwise it uses a combinatorial method.
- `IrreducibleMethod="BengerScott"`. Uses the Benger and Scott towering method to obtain the irreducible polynomial. Otherwise, use the traditional try-and-increment method.
- `compileDirectory="$HOME/Download/miracl"`. Directory containing the libraries of the desired multiprecision library.
- `nocompile="false"`. Compile by default. A test bed example will be compiled for the multi-precision library.

### 6.6.1 Other user-specified options

- The  $x$ -parameter of the curve, represented as  $z$  in the program.
- The  $a$  and  $b$  parameters of the curve in the short Weierstrass representation. This is useful for curves generated externally by the user. For example, curves generated with the CM construction method, which is not currently implemented in this code generator.
- `UserChainFE` and `UserChainCF`. Set the addition sequence for the final exponentiation or for fast hashing to  $G_2$ ; e.g., "`UserChainFE:=[1,2,3,6,12,18,30,36];`"

Figure 6.1: Feedback symbols

```

File Edit View Terminal Tabs Help
robotito@toshitita:~/Documents/PhDThesis  robotito@toshitita:~/speed2010
Loading "getFastHashing.magma"
Loading "Towering.magma"
Loading "TowerDependant.magma"
Loading "getRatePairing.magma"
Loading "getatePairing.magma"
Loading "getTatePairing.magma"
Loading "getEcap.magma"
Generating the curve...
+.....
+.....
+.....
#
p=5%8, p=1%3

Nodes to visit on the addition chain: (final exponentiation edition)
[ 3, 5, 7, 14, 15, 21, 25, 35, 49, 54, 62, 70, 87, 98, 112, 245, 273, 319,
343, 434, 450, 581, 609, 784, 931, 1407, 1911, 4802, 6517 ]
Number of nodes: 47, 46E! 54, 45E! 55E! 57E! 55E! 59
[ 1, 2, 3, 5, 7, 10, 11, 13, 14, 15, 21, 25, 28, 35, 49, 54, 62, 70, 87, 98,
112, 224, 245, 273, 308, 319, 343, 392, 427, 434, 448, 450, 546, 581, 609, 686,
784, 931, 980, 1407, 1715, 1911, 3822, 4802, 6517 ]

```

- UserChainFEcontinue and UserChainCFcontinue. This is a Boolean variable to specify if the user supplied addition sequence should be improved or not. This is useful for addition sequences which can be improved.
- Towering. Specify the tower construction method; e.g., the default construction for the BN curves is  $1 - 2 - 4 - 12$ , "Towering:=[1,2,6,12];" will force the described alternative construction.
- ForcePairing. Specify the desired pairing construction, values ranges from: "optimal", "rate", "ate", "tate".

## 6.7 Feedback symbols in the program

During the code generation, and for feedback, some symbols are output when long or iterative sub-operations are taking place. See figure 6.1 for a screenshot.

### Generation of the parameters of the curve

- "+". A process to find a prime has started.

- “ $\sim$ ”.  $r \notin \mathbb{Z}$ .
- “ $\neg$ ”.  $p \notin \mathbb{Z}$ .
- “.”.  $x \not\equiv b \pmod n$ .
- “ $\wp$ ”. Primality test taking place.

### Finding a short addition chain

- “ $E! \#$ ”. The inclusion of the randomly generated element for substitution will potentially make the chain substantially longer.

### Irreducible tests

- “.”. Attempt to get an irreducible for the Magma implementation.
- “first”. The cofactor got a positive square root.
- “second”. The cofactor got a negative square root.

### R-ate pairing construction

- “.” Testing the bilinearity and non-degeneracy of  $e_{A,B}(Q, P)$  for all valid  $(A, B)$  values.

## 6.8 Code sequence

The code generator has the code sequence shown in Algorithm 11.

The program start by looking up previously assigned values to set the defaults. We make use of “if-not-assigned-then” instructions complemented with “select-case” statements.

### 6.8.1 Default variables and functions load

The default security level for the code generator is 128. If there is no assigned curve, then we choose:

---

**Algorithm 11** Automatic code generator pseudo-code

**Input:** Security level, curve to use (see [FST10]),  $x$ ,  $a$  and  $b$  from the short Weierstrass representation of the curve, word size of the target environment, precomputed addition chains,  $x$ -parameter algorithm to use, method to construct the tower to construct the finite field extensions, and the setting for the artificial immune system part of the code. Defaults to BN curves with a security level of 128 bits.

**Output:** Compressed directory with the final exponentiation and hashing to G2 code, supporting code functions depending on the  $x$ -parameter and on the tower construction, the pairing function code, a test bed, and a script to compile and run the test bed.

- 1: Verify the default values and the user supplied parameters.
  - 2: Get the parameters of the curve, the polynomials describing the family, degree of the twist admitted, the tower construction, the  $x$ -parameter, among others.
  - 3: Construct the final exponentiation and the code for hashing to  $G_2$  for the code generator in Magma and for the requested multi-precision library.
  - 4: Verify that the irreducible polynomial to construct the curve gives the twist of the right order.
  - 5: Compute the cost of the pairing lattice, construct the R-ate pairing and compute its cost.
  - 6: Compare and decide which pairing function to construct for the desired target library: Optimal (pairing lattice), R-ate, ate or Tate pairing. The user may have requested a specific pairing function.
  - 7: Generate the code, update the template files, pack the code and compress it.
  - 8: Optionally: compile an example program.
-

Level	Family	Construction <sup>4</sup>
80	KSS $k = 8$	-
128	BN Curves	6.8
192	KSS $k = 18$	6.12
256	KSS $k = 36$	6.14

After assigning the rest of the defaults values from §6.6, the program loads the following files:

- `languageMIRACL.magma, languageRELIC.magma, language.magma`

Contains the language description: maps, position of the operators, the base for the header construction and extra requirements.

- `lambda.magma`

Contains the definition of the families of curves.

- $k$  the embedding degree
- $D$  the discriminant of the curve
- $px, rx, tx$  polynomials describing the curve
- $cfx$  the cofactor of the curve
- $qx, Tx$  polynomials defining the cofactor of the twist curve
- $nx, bx$  if the  $x$ -parameter of the curve need to be of the form:  $x \equiv n \pmod{b}$
- $ax$  flag to indicate the short Weierstrass form of the curve.
- $dx$  the base field of the twist curve
- $ex$  the divisor of  $r$ , see the  $e$  parameter of the KSS curves [KSS08].

In chapters 4 and 5, a  $\lambda$  expansion from the hard part of the final exponentiation and the cofactor of the twist of curve was needed as part of the algorithms to speed-up these operations. This file also contains a function to get these  $\lambda$  elements.

- `awkward.magma`  
Looks for a divisor of the  $\lambda$  elements as denoted in chapters 4 and 5.
- `getchain.magma`, `approximatechain.magma`, `hypermutation.magma`, `mutate.magma`  
Contains the code for the AIS method to solve an addition chain: see section 3.5
- `codegenerator.magma`  
Contains the code to construct the instruction, as described in this chapter.
- `vectorize.magma`, `reduceterms.magma`  
Generates a code sequence from a vector addition chain and the code optimization.
- `WriteFEMagma.magma`  
Generates the code to be used by the generator itself for the final exponentiation
- `getparametescurve.magma`  
Evaluates the  $x$ -parameter to get a, elliptic curve.
- `LowHamming.magma`  
Contains code with the two methods to get a low Hamming-weight  $x$ -parameter: combinatorial and fixed.
- `setCF.magma`, `lambdaCF.magma`, `WriteCF.magma`  
Gets the  $\lambda$  elements for the hashing to  $G_2$  method 5.3, and evaluates the cofactor of the twist of the curve in polynomial form. Writes the code to be used by the program itself.
- `Rewriter.magma`  
Looks up the code for flags to insert final code or to mutate the program.
- `Pack.magma`  
This program creates a directory containing the generated programs and creates a compressed archive with them.

- `getLineFunction.magma`

Constructs the line function based on a fixed set of code.

To load some of the function code, we need initialize some of the variables to be used.

- `getFinalExpo.magma`, `getFastHashing.magma`

Generates the final code for the final exponentiation and for hashing to  $G_2$ .

- `Towering.magma`

Defines the default way to construct the final extension field using towers.

- `TowerDependant.magma`

Generates the code that depends on the tower construction method.

- `getOptimalPairing.magma`, `getRatePairing.magma`, `getatePairing.magma`, `getTatePairing.magma`

Constructs the Optimal Lattice, R-ate, ate and Tate pairings code.

- `getEcap.magma`

Constructs the pairing function call.

### Get the parametes of the curve

The compilers uses the “GetCurveParameters” procedure function to set the values of the curve in polynomial form. It then reviews if the coefficients can be defined over the rational field or over the integers ring and change the definition to the integers whenever possible.

It looks up the towering construction method for the selected curve, in particular it identifies the tower in the middle.

The  $x$ -parameter is generated accordingly to the methods chosen by the user. First, we estimate the size of  $x$  for the corresponding security level. We also verify the values of the field size  $p$ :  $p \equiv 1 \pmod{3}$ ,  $\equiv \{3, 5, 7\} \pmod{8}$ , etc.



### Generate the final exponentiation and hashing to $G_2$ code

For both the final exponentiation and for hashing to  $G_2$ , the corresponding  $\lambda_i$  values are calculated, then the AIS method is executed, which detects the coefficients, and generates the code sequence. An optimisation of the code takes place before generating the corresponding code: the pre-computing part, the main sequence, and the ending part (freeing memory, if required). Finally, it writes the code in the corresponding files.

There is however, an extra step on the hashing to  $G_2$  part, the need to construct the twist of the curve. The compiler uses either the Benger and Scott method [BS09] to find the irreducible polynomial to construct the final extension field as a tower of smaller extension fields, or the classical try-and-increment method to find the irreducible polynomial. In both methods an irreducibility test takes places. However, in the former, the irreducible polynomial tends to be smaller.

There is a need to test the results before proceeding, so that if one get the wrong twist, one can try with a different code sequence. It is always possible to get a different sequence from the AIS method if the coefficients are sparse enough. For example, in the KSS  $k = 18$  curves, there are plenty of possibilities. However, in the BLS curves [BLS02][FST10, Construction 6.6] there are not many.

### Pairing construction and cost estimate

For the pairing lattices, to get the optimal pairing, the Weak Popov form described in 2.9.1 is preferred. For the R-ate pairing, since the optimal pairing will reach at least the R-ate pairing type I, one can safely stick strictly to R-ate pairing type III: the  $T_i - T_j$  construction.

Here, the following files need to be loaded:

- millerRGengood2

Contains a generic implementation of the R-ate pairing. This pairing reverse its internal behavior depending on the length parameters  $(T_i - T_j)$ .

- `TiCombo`  
Evaluates the R-ate pairing to find bilinear and non-degenerate  $T_i - T_j$  pairs.
- `ChooseTiTj`  
Estimates the R-ate pairing cost.
- `millerate`, `millerTate`, `millerOptimal`  
Contains the ate, Tate and pairing lattice pairings implementations
- `OptimalPairingCombo`  
Finds the optimal pairing lattice.
- `ChooseOP`  
Estimates the pairing lattice cost (all vectors)
- `getOptimalPairing`  
Constructs the optimal pairing lattice code.

To estimate the cost of the pairings, since both the pairing lattice, R-ate and ate pairings have the same elliptic curve arithmetic complexity, one can safely rely on the loop length and the Hamming-weight of their parameters, plus the additional overload of the “extra” operations in the corresponding pairing function. In some cases, the family of curves presents a very small trace, when the pairing lattice matches the ate pairing, we must use it as it requires a simpler construction. The user of the compiler can always force the construction of a particular pairing function.

At the end, the code is packaged in compressed form, ready to be compiled with a sample test run. However, please note that since some parts of the code are generated as separate files, the developer may prefer to take the pieces of code and embed them in their code, rather to have it as separate programs. This way, some memory can be saved as we need new temporary variables for every piece of generated code.

## 6.8.2 Sample generated code

For more examples, please visit the mini-website: [\[Dom\]](#)

Listing 6.4 show the basic Miller function for the MIRACL library. Alternatively, this function can be constructed using the Costello et al.  $2^n$ -tuple-and-add pairings [\[CBNW10\]](#). However, the use of affine coordinates is preferred for this research, as recommended by Lauter, Montgomery and Naehrig [\[LMN10\]](#).

Listing 6.4: Basic Miller function

---

```
void Miller(ECn3 &P, ZZn &Qx, ZZn &Qy, ZZn18 &f3, ECn3 &m2P, Big &m2)
{
    int i;
    int nb=bits(m2);
    f3=1;
    m2P=P;
    for (i=nb-2;i>=0;i--) {
        f3*=f3;
        f3*=g(m2P,m2P,Qx,Qy);
        if (bit(m2,i)) f3*=g(m2P,P,Qx,Qy);
    }
}
```

---

In some cases, the Miller function can reuse temporary states for cases with the same parameters but different loop length, as in the KSS curves with  $k = 18$ . For those cases, a Miller function as shown in Listing 6.5 would be appropriate.

Listing 6.5: Miller function with reusing parameters

---

```
void Miller(ECn3 &P, ZZn &Qx, ZZn &Qy, ZZn18 &f3, ECn3 &m2P, Big &m2,
ZZn18 &fd, ECn3 &dP, Big &d){
    int i;
    int nb=bits(m2);
    int nb2=bits(d);
    f3=1;
    m2P=P;
    for (i=nb-2;i>=0;i--) {
        f3*=f3;
        f3*=g(m2P,m2P,Qx,Qy);
        if (bit(m2,i)) f3*=g(m2P,P,Qx,Qy);
        if (i==nb-nb2) {dP=m2P; fd=f3;}
    }
}
```

---

### 6.8.2.1 KSS $k = 8$

To generate the code for this family of curves, use:

```
level:=80;
load "autopairing.magma";
```

Listing 6.6, listing 6.7 and listing 6.8 show a selection of the generated code.

Listing 6.6: Optimal Pairing Lattice for curve KSS "k=8"

---

```
#include "OPTIMALc8lMIRACL.h"

BOOL Pairing(ECn2 &P, ZZn &Qx, ZZn &Qy, ZZn8 &r, ZZn2 &X, ZZn2 &d2,
             ZZn2 &d3, Big &x, Big w1, Big w2)
{
    ZZn8 t0;
    ZZn8 f3;
    ECn2 m1P;
    ECn2 T1;
    ECn2 T2;
    ECn2 T3;
    f3=1;
    Miller(P, Qx, Qy, t0, m1P, w1);
    f3*=t0;
    Miller(P, Qx, Qy, f3, m1P, w2);
    f3*=t0;
    T3=psi(P, d2, d3, 0);
    T3=w1*T3;
    T1=T3;
    T3=psi(P, d2, d3, 1);
    T3=w2*T3;
    T2=T3;
    f3*=g(T1, T2, Qx, Qy);
    T3=psi(P, d2, d3, 1);
    T3=w2*T3;
    T1+=T3;
    T3=psi(P, d2, d3, 2);
    T2=T3;
    f3*=g(T1, T2, Qx, Qy);
    SoftExpo(f3, X);
    HardExpo(r, f3, X, x);
    if (f3.iszero()) {
        return FALSE;
    }
    return TRUE;
}
```

---

Listing 6.7: Hard part of the Final Exponentiation for the KSS "k=8" curves

---

```

#include "Fec81MIRACL.h"
void HardExpo(ZZn8 &r, ZZn8 &f3x0, ZZn2 &X, Big &x){
//vector=[ 1, 2, 4, 5, 7, 10, 15, 26, 28, 30, 55, 75, 80, 100, 108,
144 ]
    ZZn8 xA;
    ZZn8 xB;
    ZZn8 t0;
    ZZn8 t1;
    ZZn8 t2;
    ZZn8 t3;
    ZZn8 t4;
    ZZn8 t5;
    ZZn8 t6;
    ZZn8 f3x1;
    ZZn8 f3x2;
    ZZn8 f3x3;
    ZZn8 f3x4;
    ZZn8 f3x5;

    f3x1=pow(f3x0,x);
    f3x2=pow(f3x1,x);
    f3x3=pow(f3x2,x);
    f3x4=pow(f3x3,x);
    f3x5=pow(f3x4,x);

    xA=f3x0;
    xB=Frobenius(inverse(f3x0),X,2);
    t0=xA*xB;
    xB=Frobenius(inverse(f3x1),X,1);
    t1=t0*xB;
    xB=f3x2;
    t0=t0*xB;
    xB=Frobenius(f3x0,X,1);
    t2=t1*xB;
    xB=Frobenius(f3x0,X,3);
    t1=t2*xB;
    xB=Frobenius(inverse(f3x2),X,1)*Frobenius(inverse(f3x4),X,1);
    t2=t2*xB;
    xB=Frobenius(inverse(f3x1),X,2);
    t6=t1*xB;
    xB=Frobenius(f3x0,X,1);
    t1=t1*xB;
    t4=t6*t0;
    xA=Frobenius(f3x2,X,2);
    xB=Frobenius(inverse(f3x0),X,2);
    t3=xA*xB;
    xA=f3x1*Frobenius(inverse(f3x3),X,1);
    xB=Frobenius(inverse(f3x0),X,2);
    t5=xA*xB;
    xA=Frobenius(f3x1,X,3);
    xB=Frobenius(f3x1,X,3);

```

```

t0=xA*xB;
xB=Frobenius (f3x2,X,3);
t0=t0*xB;
t6=t4*t6;
xB=f3x5*Frobenius (inverse (f3x3),X,2);
t4=t4*xB;
t3=t6*t3;
t4=t4*t6;
t0=t3*t0;
t1=t1*t1;
t1=t1*t5;
t3=t0*t3;
xB=Frobenius (f3x4,X,2);
t0=t0*xB;
t1=t3*t1;
t3=t3*t4;
t1=t1*t1;
t1=t1*t2;
xB=f3x3;
t1=t1*xB;
xA=f3x4*Frobenius (f3x5,X,2);
xB=f3x3;
t2=xA*xB;
t0=t1*t0;
t1=t1*t3;
t0=t0*t0;
t0=t0*t2;
t0=t0*t0;
t0=t0*t1;

r=t0;

}

```

Listing 6.8: Line function for the KSS "k=8" curves

```

ZZn8 line(ECn2 &A, ECn2 &C, ECn2 &B, ZZn2 &slope, ZZn2 &extra, BOOL
    Doubling, ZZn &Qx, ZZn &Qy){
    ZZn8 w;
    ZZn4 a;
    ZZn4 b;

    ZZn2 X,Y;
    ZZn2 t0;
    ZZn2 t1;
    ZZn2 t2;

#ifdef AFFINE
    A.get(X,Y);
    t0=slope*Qy;
    t1=slope*X;

```

---

```

    t1=Y-t1;
    a.set(t0,t1);

    t0=-Qy;
    b.set(t0);

    w.set(a,b);
#endif

    return w;
}

```

---

### 6.8.2.2 KSS $k = 18$

To generate the code for this family of curves, use:

```

FFE=["zzn", "zzn3", "zzn6", "zzn18"];
UserChainFE=[1, 2, 3, 4, 5, 7, 8, 14, 15, 16, 21, 25, 28,
35, 42, 49, 54, 62, 70, 87, 98, 112, 147, 245, 273, 294,
319, 343, 392, 434, 450, 581, 609, 784, 931, 1162, 1407,
1862, 1911, 3724, 4655, 4802, 6517];
UserChainFEcontinue=false;
UserChainG2=[1, 2, 3, 5, 7, 8, 11, 18, 26, 28, 31, 38, 45,
69, 70, 78, 98, 176, 245, 253, 323, 343, 381, 389, 686, 784,
829, 867, 1029, 1218, 1658, 1911, 2058, 4116, 5145, 5774];
UserChainG2continue=false;
curve="6.12";
load "autopairing.magma";

```

Listing 6.9 and listing 6.10 show a selection of the generated code.

Listing 6.9: R-ate pairing for curve KSS "k=18"

---

```

#include "RATEc6.12lMIRACL.h"

BOOL Pairing(ECn3 &P, ZZn &Qx, ZZn &Qy, ZZn18 &r, ZZn3 &X, ZZn3 &d2,
ZZn3 &d3, Big &x,Big &m2,Big &d)
{
    ZZn18 fm2;

```

---

```

ZZn18 fm1;
ECn3 m1P;
ECn3 m2P;
int nb;
int i;
ZZn18 fd;
ECn3 dP;
Miller(P, Qx, Qy, fm2, m2P, m2, fd, dP, d);
fd*=fm2;

m1P=m2P;
fd*=g(m1P, dP, Qx, Qy);
fm2*=Frobenius(fd, X, 6);
m1P=psi(m1P, d2, d3, 6);
fm2*=g(m1P, m2P, Qx, Qy);
if (fm2.iszero()) {
return FALSE;
}
SoftExpo(fm2, X);
HardExpo(r, fm2, X, x);
return TRUE;
}

```

---

Listing 6.10: Hard part of the Final Exponentiation for the KSS "k=18" curves

---

```

#include "FEC6.12lMIRACL.h"
void HardExpo(ZZn18 &r, ZZn18 &f3x0, ZZn3 &X, Big &x){
//vector=[ 3, 5, 7, 14, 15, 21, 25, 35, 49, 54, 62, 70, 87, 98, 112,
          245, 273, 319, 343, 434, 450, 581, 609, 784, 931, 1407, 1911,
          4802, 6517 ]
ZZn18 xA;
ZZn18 xB;
ZZn18 t0;
ZZn18 t1;
ZZn18 t2;
ZZn18 t3;
ZZn18 t4;
ZZn18 t5;
ZZn18 t6;
ZZn18 t7;
ZZn18 f3x1;
ZZn18 f3x2;
ZZn18 f3x3;
ZZn18 f3x4;
ZZn18 f3x5;
ZZn18 f3x6;
ZZn18 f3x7;

f3x1=pow(f3x0, x);
f3x2=pow(f3x1, x);
f3x3=pow(f3x2, x);
f3x4=pow(f3x3, x);

```



```

f3x5=pow(f3x4,x);
f3x6=pow(f3x5,x);
f3x7=pow(f3x6,x);

xA=Frobenius(inverse(f3x1),X,2);
xB=Frobenius(inverse(f3x0),X,2);
t0=xA*xB;
xB=Frobenius(inverse(f3x2),X,2);
t1=t0*xB;
t0=t0*t0;
xB=Frobenius(inverse(f3x0),X,2);
t0=t0*xB;
xB=Frobenius(f3x1,X,1);
t0=t0*xB;
xA=Frobenius(inverse(f3x5),X,2)*Frobenius(f3x4,X,4)*Frobenius(f3x2,
X,5);
xB=Frobenius(f3x1,X,1);
t5=xA*xB;
t0=t0*t0;
t3=t0*t1;
xA=Frobenius(inverse(f3x4),X,2)*Frobenius(f3x1,X,5);
xB=Frobenius(f3x2,X,1);
t1=xA*xB;
xA=Frobenius(f3x2,X,1);
xB=Frobenius(f3x2,X,1);
t0=xA*xB;
xB=Frobenius(f3x2,X,4);
t0=t0*xB;
xB=Frobenius(f3x1,X,4);
t2=t3*xB;
xB=Frobenius(inverse(f3x1),X,2);
t4=t3*xB;
t2=t2*t2;
xB=Frobenius(inverse(f3x2),X,3);
t3=t0*xB;
xB=inverse(f3x2);
t0=t3*xB;
t4=t3*t4;
xB=Frobenius(inverse(f3x3),X,3);
t0=t0*xB;
t3=t0*t2;
xB=Frobenius(inverse(f3x3),X,2)*Frobenius(f3x0,X,5);
t2=t3*xB;
t3=t3*t5;
t5=t3*t2;
xB=inverse(f3x3);
t2=t2*xB;
xA=Frobenius(inverse(f3x6),X,3);
xB=inverse(f3x3);
t3=xA*xB;
t2=t2*t2;
t4=t2*t4;
xB=Frobenius(f3x3,X,1);

```

```

t2=t1*xB;
xA=Frobenius (f3x3,X,1);
xB=Frobenius (inverse (f3x2),X,3);
t1=xA*xB;
t6=t2*t4;
xB=Frobenius (f3x4,X,1);
t4=t2*xB;
xB=Frobenius (f3x3,X,4);
t2=t6*xB;
xB=Frobenius (inverse (f3x5),X,3)*Frobenius (f3x5,X,4);
t7=t6*xB;
t4=t2*t4;
xB=Frobenius (f3x6,X,1);
t2=t2*xB;
t4=t4*t4;
t4=t4*t5;
xA=inverse (f3x4);
xB=Frobenius (inverse (f3x4),X,3);
t5=xA*xB;
xB=Frobenius (inverse (f3x4),X,3);
t3=t3*xB;
xA=Frobenius (f3x5,X,1);
xB=Frobenius (f3x5,X,1);
t6=xA*xB;
t7=t6*t7;
xB=Frobenius (f3x0,X,3);
t6=t5*xB;
t4=t6*t4;
xB=Frobenius (inverse (f3x7),X,3);
t6=t6*xB;
t0=t4*t0;
xB=Frobenius (f3x6,X,4);
t4=t4*xB;
t0=t0*t0;
xB=inverse (f3x5);
t0=t0*xB;
t1=t7*t1;
t4=t4*t7;
t1=t1*t1;
t2=t1*t2;
t1=t0*t3;
xB=Frobenius (inverse (f3x3),X,3);
t0=t1*xB;
t1=t1*t6;
t0=t0*t0;
t0=t0*t5;
xB=inverse (f3x6);
t2=t2*xB;
t2=t2*t2;
t2=t2*t4;
t0=t0*t0;
t0=t0*t3;
t1=t2*t1;

```

```

t0=t1*t0;
xB=inverse(f3x6);
t1=t1*xB;
t0=t0*t0;
t0=t0*t2;
xB=f3x0*inverse(f3x7);
t0=t0*xB;
xB=f3x0*inverse(f3x7);
t1=t1*xB;
t0=t0*t0;
t0=t0*t1;

r=t0;

}

```

---

To get the code for a security level equivalent to AES-128, using a BN curves with the Nogami et al. parameter, run:

```

z:=0x4080000000000001;
load "autopairing.magma";

```

To get the pairing code for the default curve, single-try for low Hamming weight, try-and-increment method to find an irreducible polynomial, and an optimal pairing lattice, run:

```

XValueGenerator:="LowHamming";
ForcePairing:="optimal";
IrreducibleMethod:="Traditional";
load "autopairing.magma";

```

To get the code for a security level equivalent to AES-256, KSS:k=36 curve, with pre-computed addition sequences, run:<sup>5</sup>

```

curve:="6.14";
level:=256;
UserChainFE:=[ 1, 2, 4, 5, 7, 8, 10, 11, 14, 19, 20, 21,

```

---

<sup>5</sup>**WARNING!** Code construction takes a while. My pentium 4, use to run out of memory, better use a 64-bit computer, or a high memory kernel. Alternatively, use construction 6.6 with  $k = 24$  for this security level.

```
29, 37, 48, 49, 60, 62, 67, 87, 98, 112, 124, 125, 127,
146, 147, 157, 196, 245, 248, 268, 288, 294, 348, 385, 394,
434, 448, 460, 560, 561, 609, 676, 686, 784, 882, 931, 1029,
1078, 1372, 1519, 1529, 1540, 1617, 1862, 1876, 1989, 2009,
2401, 2547, 2695, 2724, 2744, 3528, 3724, 3871, 4018, 4116,
4478, 4578, 4802, 5362, 5369, 5488, 5613, 6174, 6517, 7203,
9604, 9761, 9998, 10290, 10976, 13034, 13279, 13377, 14063,
15092, 16709, 16807, 19208, 22932, 24010, 25539, 25807,
26075, 27048, 28622, 28910, 29449, 33614, 40817, 48020,
52136, 58898, 63476, 67228, 71099, 76712, 79436, 83914,
88392, 104272, 105644, 106673, 117649, 123137, 128499,
129654, 142198, 151959, 152635, 154007, 168070, 176784,
178773, 180649, 182525, 186782, 253624, 283073, 285719,
299782, 308014, 310415, 333347, 365050, 428526, 433895,
444185, 549829, 597849, 621859, 645869, 899493, 907578,
936488, 949767, 1099658, 1126706, 1291738, 1478520, 1507142,
1817557, 1858374, 1981511, 2033647, 2333429, 2487436,
3037265, 3189900, 4067294, 4184943, 4353013, 5260591,
5428661, 6074530, 6204184, 6648369, 12722899, 17983490,
35966980, 38000627 ];
UserChainFEcontinue:=false;
UserChainG2:=[ 0x1, 0x2, 0x3, 0x4, 0x5, 0x6, 0x7, 0x8, 0x9,
0xA, 0xB, 0xC, 0xD, 0x10, 0x11, 0x12, 0x14, 0x15, 0x17,
0x19, 0x1B, 0x1C, 0x1E, 0x1F, 0x20, 0x21, 0x23, 0x24, 0x25,
0x26, 0x29, 0x2D, 0x2E, 0x2F, 0x31, 0x34, 0x35, 0x39, 0x3A,
0x3D, 0x3F, 0x42, 0x43, 0x4E, 0x4F, 0x56, 0x5D, 0x5F, 0x6F,
0x74, 0x77, 0x79, 0x7E, 0x82, 0x85, 0x89, 0x8D, 0x97, 0x9D,
0xAD, 0xAE, 0xB8, 0xBD, 0xC4, 0xC5, 0xC6, 0xD0, 0xDA, 0xDD,
```

0x106, 0x10D, 0x116, 0x120, 0x121, 0x131, 0x138, 0x145,  
0x148, 0x14D, 0x14E, 0x151, 0x154, 0x158, 0x174, 0x181,  
0x19C, 0x1A0, 0x1A6, 0x1BA, 0x1C3, 0x1C5, 0x1CF, 0x1DB,  
0x1DE, 0x1EA, 0x1F4, 0x200, 0x21F, 0x226, 0x232, 0x238,  
0x2B6, 0x2C0, 0x2CD, 0x32C, 0x34F, 0x374, 0x394, 0x3A1,  
0x3A3, 0x3CC, 0x3EC, 0x419, 0x424, 0x431, 0x443, 0x4BC,  
0x4EA, 0x4F0, 0x511, 0x542, 0x57B, 0x5B5, 0x5F2, 0x610,  
0x61A, 0x633, 0x634, 0x649, 0x74F, 0x766, 0x7F3, 0x850,  
0x87F, 0x890, 0x896, 0x9EE, 0xA17, 0xA8B, 0xBA1, 0xBD6,  
0xC4D, 0xC5E, 0xC9D, 0xCC1, 0xD46, 0xD95, 0xDC6, 0xDD6,  
0xE18, 0xE4C, 0xFEC, 0x1042, 0x10CB, 0x10D1, 0x118E, 0x1225,  
0x1282, 0x1403, 0x147A, 0x1587, 0x158D, 0x16AD, 0x16B6,  
0x16CB, 0x1779, 0x1849, 0x190F, 0x1A57, 0x1A5B, 0x1C2A,  
0x1DC6, 0x1FEC, 0x2140, 0x221D, 0x23FB, 0x24BF, 0x2853,  
0x29C7, 0x2B81, 0x2CB2, 0x2F68, 0x355A, 0x3792, 0x3801,  
0x3CEB, 0x3E39, 0x3EFE, 0x401E, 0x41E1, 0x428E, 0x42EB,  
0x47A7, 0x48C7, 0x4964, 0x4AB1, 0x4E00, 0x5243, 0x5C5A,  
0x5C80, 0x5E5B, 0x63D6, 0x648E, 0x64DC, 0x687D, 0x7113,  
0x7537, 0x7A27, 0x7FDC, 0x80FD, 0x81D7, 0x857A, 0x86AB,  
0x896B, 0x9541, 0x9549, 0x96EF, 0x9A63, 0x9AA6, 0xA0DA,  
0xA61C, 0xAA4D, 0xAF5E, 0xB32A, 0xBA90, 0xBB12, 0xBF2B,  
0xC15D, 0xCDAA, 0xDF38, 0xE122, 0xECC3, 0xF1D4, 0xFFEC,  
0x10D81, 0x113CA, 0x119BC, 0x1224C, 0x12638, 0x1265C,  
0x126D0, 0x128EF, 0x13E7C, 0x146CC, 0x14CE6, 0x15565,  
0x156B6, 0x1647C, 0x1893B, 0x193C6, 0x1950B, 0x1A730,  
0x1A930, 0x1B123, 0x1B268, 0x1C2AA, 0x1C9F9, 0x1E0C4,  
0x1E6D4, 0x268AB, 0x27548, 0x29688, 0x2B44E, 0x2B71B,  
0x2CDD1, 0x2FD39, 0x36E4C, 0x39069, 0x3A2EB, 0x3DA7D,

0x3F4D4, 0x414C0, 0x489F7, 0x48A36, 0x4980C, 0x4B11B,  
0x4C7C8, 0x568A2, 0x59DFC, 0x5B575, 0x5C2BB, 0x5D2A7,  
0x610E0, 0x612D4, 0x61F32, 0x666D9, 0x6CB67, 0x760A8,  
0x774AB, 0x80F0E, 0x83309, 0x85FBB, 0x8708C, 0x88AE7,  
0x8911A, 0x8A963, 0x8F2C7, 0x93555, 0x97240, 0x986BA,  
0x9A2E4, 0x9A41C, 0x9B9A3, 0xA6901, 0xA75C2, 0xAB7A3,  
0xAEFA4, 0xB17F7, 0xB5AE2, 0xC288C, 0xC327A, 0xCD896,  
0xDCA6A, 0xE7D94, 0xEA75B, 0xF887D, 0x101F6C, 0x10A8D7,  
0x11437D, 0x1151C9, 0x117D4A, 0x11CF8D, 0x12E949, 0x1347A4,  
0x13DCED, 0x14873A, 0x14C638, 0x179409, 0x180E30, 0x184E4E,  
0x1A2F12, 0x1CA45A, 0x1DE2D6, 0x204B81, 0x215F4B, 0x23461F,  
0x238EE6, 0x241460, 0x24F398, 0x255874, 0x26ADD9, 0x2768EB,  
0x27C56B, 0x291251, 0x2CA2BA, 0x2E6564, 0x2EC93A, 0x2FEB86,  
0x30AAB1, 0x325D19, 0x336A9A, 0x374517, 0x3D55F7, 0x41DFEE,  
0x41E1B3, 0x41F27E, 0x437BB9, 0x43C9B9, 0x466041, 0x478679,  
0x4AF4C5, 0x515B9E, 0x52A26A, 0x564555, 0x5DBA00, 0x5F6B23,  
0x5FB5D4, 0x6035B0, 0x69BC6A, 0x6FCF3E, 0x7A4500, 0x7B44EC,  
0x7C31AF, 0x80C9BB, 0x80CCE7, 0x823163, 0x87F41E, 0x891A7A,  
0x918B06, 0x957FDA, 0x9714E5, 0x977D62, 0x99475B, 0x9A9E11,  
0xA91BA5, 0xAFE70C, 0xB58508, 0xB60BB3, 0xBFC556, 0xD3A243,  
0xD7B703, 0xDF17AB, 0xE0AB71, 0xE99E38, 0xEFBD6A, 0xF07EC7,  
0xF9F107, 0xFB17D7, 0x10580D8, 0x10DB3E1, 0x1163EC8,  
0x1227142, 0x12F49D8, 0x137DAF2, 0x1417F0E, 0x142A7FD,  
0x1572F37, 0x166B7B4, 0x1849A8A, 0x18924C0, 0x1947FA2,  
0x1B4CB23, 0x1C7B46C, 0x1C86EFC, 0x1FFB413, 0x2047BDB,  
0x21C0FE4, 0x2254539, 0x2369702, 0x23B481D, 0x244EB01,  
0x2618F5B, 0x26F59C5, 0x290B910, 0x29BA8B4, 0x29EA5ED,  
0x2A95D90, 0x2AF1305, 0x2B772C0, 0x2BA29DB, 0x2BFFC82,

0x2CC250E, 0x2CCA60B, 0x2CD0265, 0x354F683, 0x366C610,  
0x37A0DB4, 0x39F014C, 0x3C29032, 0x472773E, 0x4B4572C,  
0x4B70B7A, 0x4DE7465, 0x51BCA5C, 0x59DFBBF, 0x61A2D6E,  
0x63E41CE, 0x63FE8FE, 0x66696D7, 0x671AECE, 0x6FAC948,  
0x791DE2D, 0x7AA2C7B, 0x809979E, 0x92C08E0, 0xA0CB451,  
0xAF65289, 0xB38343C, 0xB49B186, 0xBDF3160, 0xC3EE734,  
0xC479097, 0xC58D414, 0xC9C9DCD, 0xCA4ACDB, 0xD9E9DE2,  
0xDF1404C, 0xE429BEA, 0xE6A6155, 0xEC81B55, 0x11951DBA,  
0x1254E310, 0x12858DC1, 0x14C0D5DE, 0x14C63E80, 0x150C9EC1,  
0x153EFBDA, 0x156D613E, 0x1607FF4F, 0x179C7EF1, 0x17AB264C,  
0x186131FF, 0x19A2B10D, 0x19A45A3D, 0x19EBE0B6, 0x1A00A6EE,  
0x1B2FF0C6, 0x1BD90C6B, 0x1DD8C07E, 0x1E703DE0, 0x1EC68335,  
0x1ED6A2A1, 0x1FDC2379, 0x22210E7A, 0x2363B677, 0x24193B7F,  
0x24F0F282, 0x2781AB92, 0x288F5F73, 0x290AA45F, 0x29FA61C9,  
0x2A292B03, 0x2A39D3DA, 0x2BC2F89A, 0x2BEF9B54, 0x2EBBC062,  
0x2EE118D6, 0x2F4AD540, 0x2FCBA227, 0x343E1965, 0x3800A997,  
0x3D1C73F3, 0x40716A76, 0x428D7A5A, 0x44B2BF93, 0x482CCD47,  
0x4BCBCE93, 0x4D50691D, 0x500A92F8, 0x52CA8F7A, 0x535C1A80,  
0x53F561DB, 0x55BDD0D7, 0x55E6E328, 0x5AC5578D, 0x5ADD65BD,  
0x5BD87D94, 0x5DA03200, 0x5FA4ADDB, 0x60147D19, 0x6283D6DE,  
0x6304A099, 0x645BCFD0, 0x6FA58156, 0x725CF416, 0x7EA484AD,  
0x815397B2, 0x895D2F50, 0x96FBCD32, 0x9D6D7C00, 0x9ED433B4,  
0xA67E602F, 0xA7520272, 0xA76C3184, 0xB34B62E4, 0xB3C5A7E4,  
0xB4109CA9, 0xB5011B70, 0xCEECFC26, 0xE141DF36, 0xEDE68C11,  
0xFF7BA9CB, 0x1070D87F8, 0x1084562EA, 0x108875568,  
0x10E255127, 0x1148BE7FE, 0x11B86B146, 0x11BBA1BE0,  
0x11E55C494, 0x14850265D, 0x166C0643D, 0x1672099ED,  
0x17216EC76, 0x180598860, 0x183262E6B, 0x184DAF98E,

0x19C77787F, 0x1A6842CD0, 0x1CF8ED12F, 0x1CFB2174E,  
0x1CFF59307, 0x1E50231C8, 0x1E868F7D8, 0x1EB079DC5,  
0x1ED692D20, 0x1FFEEBAE1, 0x20EB6D636, 0x232D011B5,  
0x24EA91E20, 0x251527BB0, 0x25F43BBFC, 0x26DAE1D51,  
0x28559439D, 0x2A332041B, 0x2C822F69D, 0x2D461DDD1,  
0x2EE06380E, 0x2F2BD4388, 0x3082AA4C6, 0x3222B4BB4,  
0x3458F022B, 0x3755AA452, 0x37B98E620, 0x3A4284593,  
0x3A5075D3E, 0x3B9C8331C, 0x3E913085C, 0x3F56BDC70,  
0x3FA20339C, 0x4055867D8, 0x411F505A5, 0x42734017F,  
0x58E549B6C, 0x5946EC8DA, 0x5AF9EB9A0, 0x60269491A,  
0x655C5639A, 0x6A5CFF692, 0x73CCBC3C4, 0x792B2A6EC,  
0x7EE8B2480, 0x7F7B72D60, 0x857CBAA79, 0x86DD3A9C8,  
0x89555555A, 0x8C4366E30, 0x9630AA1E4, 0x9A7BD6177,  
0xA5B08C45B, 0xB3C4AA391, 0xC0B379FB7, 0xC2B13C330,  
0xC7F09250B, 0xCFDADA9B8, 0xD3ACA1DAB, 0xDEEDABA54,  
0xE2318D3B9, 0xE3CBB84C6, 0xEBE0F1C78, 0xEFE8086EE,  
0xF5E2534C9, 0xFAB759DE6, 0xFD5AF71C0, 0x10019F0D14,  
0x102D6205AE, 0x11FD141CFC, 0x12FC8FC6C7, 0x12FEC65DC9,  
0x1361CAFE62, 0x1531C09169, 0x15D83EF198, 0x17BD412360,  
0x19421C1CEE, 0x1BC775608B, 0x1C0A02DAE5, 0x1C7C5FCEFB,  
0x1C9B265230, 0x1CC54F7D33, 0x1E61C6F5B2, 0x1F0918F824,  
0x20D8A7C953, 0x23A0CABFF0, 0x243E383BF0, 0x2546BF9158,  
0x2654E4E27F, 0x28639BB8B5, 0x2C7590BE5A, 0x2C97B1CCD4,  
0x2D9FF72FBE, 0x2E8DDDBBCF, 0x2FF49E200C, 0x3253E1DC08,  
0x343C4AD3E0, 0x37B1A57832, 0x3F44581F1E, 0x44F3F6D8BE,  
0x4761A4F60F, 0x487FFABAA3, 0x4933C06287, 0x4B33AF1D68,  
0x4B8E8C8325, 0x4CAA13346B, 0x4CC8E9D70C, 0x555E3F2C66,  
0x5B60A87580, 0x6E5D383C47, 0x72846C3DC6, 0x7F037562D1,



0x85593AC66B, 0x95869CCC19, 0x95BE9D75B0, 0x9861CF79CB,  
0xAA5EE396C7, 0xAB064FC84B, 0xAE0E7A6D11, 0xB7B637CE88,  
0xBB700001A4, 0xBF59130A00, 0xC5FEE30092, 0xC64AAECF25,  
0xC792FEF582, 0xD7949E0296, 0xD97FA5A05B, 0xD9C7D26DA2,  
0xDADC5E55A0, 0xF00E1EE709, 0xFA6927AB64, 0x116E5877A5F,  
0x11FA9BDE88F, 0x12CE488063A, 0x153396CE8B9, 0x16176286D7F,  
0x1916AC68D8B, 0x1AE30160ABE, 0x1C12EDC6887, 0x1EECED39845,  
0x20FA77B6198, 0x25ADB2A7F00, 0x29FCF2157BE, 0x2B9113D74AC,  
0x2C86F62A975, 0x2E6D1299F27, 0x3339A137633, 0x35BFDAF2EE8,  
0x3618C03CA54, 0x361E1C19B2B, 0x36A398D45A4, 0x3FFC02A11BD,  
0x40E7E392E35, 0x41412A7F70F, 0x4826FE03356, 0x53A261801DE,  
0x5484930D597, 0x593D7BD58BC, 0x59FE2F4F873, 0x5BBAA6A58FE,  
0x5DAB3835122, 0x5F08BC242BA, 0x6ABFBC2445E, 0x6B3F37971BE,  
0x78DBB4BDF60, 0x8381A2F7627, 0x8542432510C, 0x8C6A89E8ED2,  
0x90F28994975, 0x93BC04B1649, 0xA03534A0BCB, 0xA1FEE705DFB,  
0xB7327DD46B4, 0xBC8861C731A, 0xC478991D5EB, 0xDD8F4586376,  
0x113AD619FEA1, 0x1148C4F4B8F5, 0x115C26BFB757,  
0x130A56D5C215, 0x132E950DFE05, 0x14EFC3EA668C,  
0x1606A971E0EB, 0x1962A7210FD3, 0x1B5175F4A818,  
0x1F51361EB9D5, 0x1F9DE031EE40, 0x20CAC4B9F47A,  
0x29916D58834C, 0x304560D1F50A, 0x3E1E552A5880,  
0x3E6788EABB07, 0x4887775B1902, 0x4B4FE6BDC277,  
0x4F63F965B986, 0x5029F848BA18, 0x52E309862EC4,  
0x73ADCE40233E, 0x7DB1218A2F09, 0x82EB47A230E7,  
0x8C2707ED4730, 0x8CAC61280D9B, 0xABFD9746C770,  
0xFC278F8F8188, 0x11779058429A0, 0x166DCFEE9E326,  
0x1F3896011F0C1, 0x21D1ACD6A740D, 0x29ACBEEF4A316,  
0x346C9863B6A86, 0x442F115CAEC0E, 0x4C5DC5D6D1CF5,

Curve	$\text{Log}_2(r)$	$\text{Log}_2(p)$	Pairing function	CPU cycles				Time in ms
				M	FE	extra	Tot	
6.8.1	$\sim 254$	$\sim 254$	R-ate	4.5	3.2	0	7.7	3.088
6.12	$\sim 379$	$\sim 512$	R-ate	16.8	38.7	0.2	55.7	22.06
6.12	$\sim 379$	$\sim 512$	P. Lattice	24.5	38.9	0.1	63.5	25.16

Table 6.5: Timing the generated code

```

0x62CB95C57001B, 0x6A0672A972359, 0xCCD2086EE2374];
UserChainG2continue:=false;
load "autopairing.magma";

```

### 6.8.3 Timings

When the multi-precision library supports the desired elliptic curve and finite field arithmetic, the code generator can compile a test bed with the pairing functions. One can force to skip the compiling part with a “`nocompile:=true;`” statement before running the generator. This is particularly useful when the requested curve or finite field arithmetic construction has a significantly different construction for the irreducible polynomial than the ones from the MIRACL library <sup>6</sup>. In that case, please skip the compilation, assign your own polynomial, and compile it by hand with the provided script when desired.

The first row in table 6.5 presents the timings for the BN curve: that’s construction 6.8, negative version from the [FST10]. For that curve, the R-ate pairing was generated. The second and third row is for the KSS  $k = 18$  curve, with the R-ate and optimal pairing lattice function.

The CPU cycles are described as follows: column M means the Miller function and any other operation before the final exponentiation, the column FE is for the final exponentiation, and TOT is the total. The data is presented in millions of CPU cycles per operation. The last column is the number of milliseconds. Column extra denotes the operations around

<sup>6</sup>For further details, consult the vendor’s manual, or generate a curve and examine the code.

the Miller loop and the final exponentiation, such as a line function, Frobenius operations, exponentiations, etc.

To estimate the number of CPU cycles, we manually added the “ebats” code [eba] as follows:

- We compiled the “cpucycles” code just after Miracl in the target computer.
- We added `#include "cpucycles.h"` in the corresponding `PAIRINGFUNCTIONcCURVEILIBRARY.cpp` file
- After the variables initialization, we got a snapshot of the current processor cycles:

```
long long ct2,ct3;
long long ct1=cpucycles();
```

- Just before the final exponentiation, we set up another check-point:

```
ct2=cpucycles()-ct1;
```

- Immediately after the final exponentiation, we got the final timing with:

```
ct3=cpucycles()-ct2-ct1;
ct1=ct3+ct2;
cout << "<<<ct2<<","<<<ct3<<":"<<<ct1<<>"<<endl;
```

For the estimation of the time needed to compute the pairing we used the Bernstein timing code for his “Writing really fast code” lab session [Ber] as follows:

In the `ECAPcCURVEIMIRACL.cpp`:

- At the start of the file:

```
#include<time.h>
long long nanoseconds(void){
    struct timespec t;
```

```

    clock_gettime(CLOCK_MONOTONIC, &t);

    return t.tv_sec * 1000000000LL + t.tv_nsec;
}

```

- Surrounding the pairing function call:

```

long long t[20], tot=0; int i;
for (i=0; i<20; i++) {
    OK=Pairing(P, Qx, Qy, r, X, d2, d3, x, w1, w2);
    t[i]=nanoseconds();
}
for (i=1; i<20; i++) tot+=t[i]-t[i-1];
tot/=19;
cout << "Time: " << float(tot/1000)/1000 << endl;

```

The processor used for timing the code was an Intel(R) Core(TM) i5 CPU M 540 @ 2.53GHz quad core, with 4 GB of ram.

The code was originally generated using an Intel(R) Pentium(R) 4 CPU 2.80GHz, with 1 GB of ram. The time needed to generate the code is minimal if we supply an addition sequence for both the final exponentiation (see Chapter 4) and for the hashing to  $G_2$  code (see Chapter 5).

The compiler managed to generate the code for a KSS curve with  $k = 36$  using an Intel(R) Core(TM)2 Duo CPU E6850 @ 3.00GHz with 3GB of ram. However, there are neither the elliptic curve nor the finite field arithmetic available of writing to run the code.

With the corresponding addition sequence supplied for both the final exponentiation and for hashing to  $G_2$  code, the code generation, particularly the optimisation of the code for the hashing to  $G_2$ , took several hours. The time taken was consumed in the sorting stage of the bi-dimensional matrix containing the code, which is used for optimizing it and reducing the temporary variables.

Library	CPU cycles	Time
PBC	160.22	53.37 ms
Our implementation	10.85	3.63 ms
MIRACL	9.24	3.17 ms

Table 6.6: Comparison of the generated code, BN curves. CPU cycles in millions.

A shorter addition sequence than the *initial sequence* took around 24 hours to be calculated, using an Intel Core 2 Duo processor (a few hours to get modest results). This was impossible to perform using a Pentium 4 system due to the low memory resources available.

Depending on the structure of the family of curves, the R-ate pairing code generation can take several seconds, as two pairing operations are taken in place to verify the bilinearity of the parameters [LLP08]. This is not necessary when constructing the Optimal Pairing Lattice.

A comparison of the timings from the automatically generated code (for the MIRACL library), with the general purpose PBC library [Lyn], and with the highly optimized hand-written implementation bundled with the MIRACL library is now presented. For comparison purposes a BN  $k = 12$  curve at the 128-bit security level was chosen. Please note that the overall operation count for the MIRACL implementation is very close to that reported recently by Beuchat et al., [BGM<sup>+</sup>10] in their recent record setting implementation (although the latter uses a significantly faster specially tailored implementation of  $\mathbb{F}_{p^2}$  arithmetic). The timings were collected on an Intel Core 2 Duo E6850 3GHz and are presented in table 6.6.

This demonstrates the validity of our approach, and shows that automatically generated code is nearly as fast as carefully hand-crafted code, and is significantly faster than a generic pairing implementation.

## 6.9 Final Thoughts

We have adapted and created several construction methods for automatically generate cryptographic pairing functions code to ease the job of the protocol implementer.

The generated code uses a slightly larger number of variables in the overall pairing computation compared to hand-written implementations.

Our tool shows that automatically generated code is nearly as fast as carefully hand-crafted code, as it was shown in section 6.8.3, the generated code is competitive in terms of speed and the number of CPU cycles. There are very few places where the generated code can be improved, however those check-points depend on the multi-precision library, as they use characteristics very specific to the programming language that may not be generalised.

The generation time for the addition sequence in some cases can be very long, but as it was shown for a particular case in Chapter 3, it can be worth the wait.

This work can be extended to other multi-precision libraries or for its use over a set of specialized hand-crafted functions. An interesting target library would be one based on JavaME. We also recommend as a future work, the inclusion of automatic finite field arithmetic generation.

Extra tuning by hand can be done to close the gap against the speed record of the specific multi-precision library we are generating the code.

For example, our generated code for the MIRACL library took around 3 milliseconds to compute a pairing, the MIRACL library also took around 3 milliseconds. If we add support for a set of functions that takes around 1 millisecond to compute the pairing, then the compiler will produce code that also would take approximately 1 millisecond to compute the pairing, as the finite field and elliptic curve arithmetic will be the same.

## 7 CONCLUSIONS

*"Never before have so many people with so little to say, said so much to so few." – Despair, Inc., paraphrasing Winston Churchill in the context of blogging.*

THE PhD process began with an exploratory exercise in a joint work with Kachisa and Scott [DKS]; we were looking for missing parts or inefficient methods occurring in the *implementation* of pairings. Our main idea was to create a tutorial in Magma for those interested in the implementation of cryptographic pairings.

With the exploration of the R-ate pairing, I discovered the possibility of computing the pairing function in parallel. Since the R-ate pairing may require several Miller loop functions, we may find an  $(A, B)$  pair that, in total, requires a larger number of Miller loop iterations, but where the individual Miller loops are short; if they can be executed in parallel, then we could produce a shorter R-ate pairing.

In addition to our Magma tutorial, I also developed an online version, to demonstrate the implementation of a pairing function for people without access to a Magma license.

The final exponentiation is a critical part of the Tate pairing function family, whose previous implementations were not exploiting the fact that we have a polynomial representation of the parameters of the curve. Our new method exploits this fact and with the use of the Frobenius exponentiation we can get a fast implementation. Our method can be applied to any family of pairing-friendly elliptic curves [SBC<sup>+</sup>09b]. This method, the result of joint work with Scott et al., also uses addition chains to speed-up the multi-scalar exponentiation of the hard part of the computation.

Efficiently hashing a random point to a group  $G_2$  of order  $r$  was previously achieved with the use of sliding windows, the diverse NAF techniques or with the simple scalar-point multiplication. Our method, published in Scott et al. [SBC<sup>+</sup>09a], uses a similar approach to

the final exponentiation method; however, we make use of the endomorphic map suggested by Galbraith and Scott [GS08]. This resulted in a speed-up that was significantly faster than the traditional method.

The two operations for performing the final exponentiation and for the fast hashing to  $G_2$  require several exponentiations in their respective groups of data, and arithmetic laws: Finite field or elliptic curve arithmetic. I used Artificial Intelligence techniques to find an addition sequence used to perform these exponentiations. I have implemented an Artificial Immune System based on the Cruz-Cortez et al. method [CCRHC08] for our project, and for comparison purposes I implemented a second algorithm based on simple but fast bit-XOR operations. None of these algorithm are perfect. The binary method gives a single-run solution whose output cannot be feedback to get an improved with the same method, while the artificial immune system may take from milliseconds to several hours to get a reduced sequence.

This method, which was presented in [DS09], produces short addition sequences without requiring tedious pen-and-paper computation.

The main contribution is the creation of a cryptographic compiler, which generates the pairing function and supportive code in either Magma, C or C++ programming language. The compiler is competitive against hand-crafted implementations as we produce code almost as fast as hand-made code. It should be possible to achieve a faster implementation with the use of finite field and elliptic curve arithmetic highly tied to a specific CPU processor.

A mini-language defining the operations would be a good idea to make the code generator more flexible for future improvements in the pairing function. This mini-language would translate the instructions into the target programming language and multi-precision library. I believe this is possible to do in a future and it will lead to code constructions with a smaller foot-print or with the inclusion of mid- or low-level optimisations.



## Open Problems

Since the R-ate pairing may require several Miller loop functions, one may find an  $(A, B)$  pair that, in total, generates a longer number of Miller loop iterations, but that the individual Miller loops are shorter; if they can be executed in parallel, then we can produce a shorter R-ate pairing, but no combination with such characteristics has been discovered so far.

The online tutorial is an ongoing work. In a future version, I intend to include support for other pairing-friendly elliptic curves; displaying the timings of the computations in seconds, in addition to the relative timings; better support for the exponentiation method; and a better comparison of the implementation, as well as presenting the full Magma code in a downloadable version.

Future work on the final exponentiation will seek to optimise the number of exponentiations, which can be expensive in some cases as it uses a lot of temporary storage. For families of curves with a larger addition sequences, such as the KSS curves with embedding degree  $k = 36$  which are suitable for very high security levels, an addition sequence would require a particularly enormous temporary space.

Currently the compiler was not able to produce a *short* addition sequence for the final exponentiation of the KSS family of curves with embedding degree  $k = 36$ . Exploring other mechanisms for the addition chain construction, particularly the addition of more elements of the Auto Immune System and Artificial Intelligence is an interesting future work.

The work done in the compiler can be improved by refining the use of the computational paradigms here presented.

In particular, it would be interesting to replace the hand-crafted parts of the code generator with generic instructions and a set of instructions using a designed syntax, as it would make our compiler more flexible. This task would require not only the implementation of more parsers, but also I believe that a neutral grammar needs to be defined and probably a primitive language to associate the pairing operations to the corresponding multi-precision library, which opens the possibility of further improvements specific to the library.

The main limit of the compiler is the need of an implementation of the elliptic curve arithmetic and the finite extension field arithmetic corresponding to several others embedding degrees. In particular, I am interested in the construction of the KSS family of curves with embedding degree  $k = 36$ , which requires elliptic curve arithmetic over  $\mathbb{F}_{p^6}$ . I believe that it is possible to extend the code generator to include an automatic finite field arithmetic generator, which can be used in conjunction with the generated code if there is no arithmetic defined for the target curve.



# A EXAMPLE PARAMETERS OF THE CURVE

## A.1 Hamming Weight

The values presented at Table A.1 are proposed  $x$ -parameters with low-Hamming weight for several security levels.

#	$k$	$\text{HW}(x)$	ID	$p \bmod 8$	$\text{Log}_2(p)$	$\text{Log}_2(r)$	$x$ -parameter
Approximate 128-bit security level							
1	12	4	6.8.1	3	254	254	0x4000000000820001
2	12	2	6.8.1	1	262	262	0x10000040000000000
3	12	4	6.8.1	3	254	254	0x4000000020002001
4	12	3	6.8.1	3	254	254	0x4080000000000001
Approximate 192-bit security level							
5	18	6	6.12	1	508	376	0x100400000010000A4
6	18	6	6.12	1	508	376	0x100600000C0000004
7	18	7	6.12	5	508	376	0x100000048000AC000
8	18	12	6.12	5	512	379	0x16500000000209528
Approximate 256-bit security level							
9	36	8	6.14	3	596	496	0xC00010D0108
10	36	8	6.14	1	588	489	0x802084002A1
11	36	9	6.14	7	588	489	0x80C0005802C

Table A.1: Useful Low Hamming-Weight Values

The  $x$ -parameter from row number 4 is the BN curve discovered by Nogami et al. [NAS<sup>+</sup>08]. Row 8 is included as part of the Miracl library, from Scott [Sco].

## B CERTICOM CHALLENGE

ELLIPTIC curve cryptography is becoming the standard public-key primitive for applications. Advantages are the higher security per bit in comparison to RSA, and the faster implementations. The National Institute of Standards and Technology (NIST) has a signature standard specifying elliptic curves for several security levels which are now among the best studied curves from an implementation perspective. Their security, however, is less well understood. In particular, for each security level, three curves are given, suggesting that all three curves have the same security level. One of the curves, however, is a randomly chosen curve over a field of characteristic 2, while another curve is a *Koblitz curve* over the same field. The Koblitz curve has an efficiently computable endomorphism that can be used for faster implementation than a random curve, but can also be used to speed up attacks. While it is known that these curves are somewhat weaker, the exact loss of security is unclear and needs to be studied experimentally.

The ECDLP has received a lot of attention since the suggestion of elliptic curves for use in cryptography by both Koblitz and Miller independently in the 1980s. In 1997, Certicom issued several challenges [Cer] in order “to increase the cryptographic community’s understanding and appreciation of the difficulty of the ECDLP”. The Challenge is to compute the ECC private keys from the given list of ECC public keys and associated system parameters. This is the type of problem facing an adversary who wishes to completely defeat an elliptic curve cryptosystem. This Challenge will help confirm comparisons of the security levels of systems such as ECC, RSA and DSA that have been based primarily on theoretical considerations. Our team [BBB<sup>+</sup>09b] also hope that it will provide additional information to users of elliptic curve public-key cryptosystems in terms of selecting suitable key lengths for a desired level of security. Breaking a specific system will give the community a realistic estimation of the resources required for an attacker to get the EC private keys. Level I chal-

Challenges are meant to be computationally feasible with current technology and knowledge, Level II challenges are believed to be computationally infeasible. Using the appropriate resources and technology, a negative result may suggest that current ECC parameters are stronger than believed, and the security parameters may be changed accordingly.

The challenges over fields of size less than 100 bits were posed as exercises and have been solved quickly. All challenges over fields of 109 bits have also been solved, with the last one being solved in April 2004 in a distributed effort. The next open challenges are over fields of size of 131 bits. In particular there are two challenges over a curve defined over  $\mathbb{F}_{2^{131}}$  and a third challenge for a curve over  $\mathbb{F}_p$ , where  $p$  is a 131-bit prime. Of the two curves defined over binary fields, one is a Koblitz curve [Kob91], that is, its coefficients are drawn from  $\mathbb{F}_2$ . Koblitz curves are of interest for implementations because they have particularly efficient scalar multiplication. The NIST has standardized elliptic curves for several security levels; for each of these levels one Koblitz curve and one curve over a prime field is given. It is therefore very important to have a clear understanding of the security of these curves.

It is commonly believed that the strongest attacks against the ECDLP are generic attacks based on the Birthday Paradox. Improvements by Wiener and Zuccherato [WZ98], van Oorschot and Wiener [vW99], and by Gallant, Lambert, and Vanstone [GLV00] showed how the computation can be efficiently parallelized and how group automorphisms can speed up the attack.

Bailey et al. [BBB<sup>+</sup>09a] improved these theoretical attacks on elliptic Koblitz curves and have highly optimized implementations for a challenge problem posed by Certicom. The challenge is to solve a DLP on the curve ECC2K-130, a Koblitz curve over  $\mathbb{F}_{2^{131}}$ . There is a certain amount of prestige associated with carrying out a successful attack on this reasonably strong standard curve. In principle, there is no difficulty predicting the average speed of the attack, it should take  $\sqrt{\pi 2^{131} / (2 \cdot 4 \cdot 131)}$  iterations —but it is not known how long these iterations take and how exactly they are done. In particular it is unclear how the step function in Pollard’s rho method [Pol78] should be chosen to avoid fruitless

short cycles and how to define the step function to work on orbits under the automorphisms. Certicom’s documentation consequently states that one of the objectives of their challenge is “to determine whether there is any significant difference in the difficulty of the ECDLP for random elliptic curves over  $\mathbb{F}_{2^m}$  and the ECDLP for Koblitz curves.”

## B.1 Methodology

**Step 1.** As the Certicom challenge would require a huge amount of computational power, the attack should be distributed among several clusters around the world. Our team has fine tuned the field arithmetic in  $\mathbb{F}_{2^{131}}$  needed to run the attack, improved the choice and analysis of the iteration function in the parallelized Pollard rho attack, improved the communication infrastructure for the distributed attack, and fine-tuned the implementation of the attack on a multitude of platforms, covering different CPUs, the Cell Broadband Engine, Graphics Processing Units (GPUs) and Field Programmable Gate Arrays (FPGAs).

Our team have gathered access to several other clusters of CPUs, Cells, GPUs and FPGAs and are currently attacking the ECC2K-130 challenge; we (anonymously) report progress on the attack at [www.ecc-challenge.info](http://www.ecc-challenge.info). Our estimate is that the attack will be finished this year, which means that we are taking up Certicom’s claim “The 131-bit Level I challenges are expected to be infeasible against realistic software and hardware attacks, unless of course, a new algorithm for the ECDLP is discovered” (see [Cer, p.22,p.23]).

The job of each client around the Internet is to compute pairs  $(s, h)$  and send those pairs to the (8) central servers. Here  $s$  is a random seed, and  $h$  is the hash of the corresponding distinguished point.

Each client sends a stream of reports to each server. The client buffers 64 reports for the server; collects the 64 reports into a 1024-byte block; adds an 8-byte nonce identifying the client site, the stream (for sites sending many streams in parallel), and the position of the block within the stream (0, 1, 2, etc.); and sends the resulting packet to the server. The server sends back an 8-byte acknowledgement identifying the client site, the stream, and

the next block position.

**Step 2.** At the server side, given enough data from the clients, the servers will find the desired discrete logarithm from the gathered data. The servers each have enough room for  $2^{37.8}$   $(s, h)$ -pairs.

## B.2 Technical Requirements

The code is “embarrassingly parallel” and requires very little communication and memory between the cores. It can be run in the background and use idle time. The program generates a small but steady stream of useful results, approximately 256 bytes per minute on a typical CPU core, to be reported to a small group of central servers. Check-pointing the program is fast and requires only about 256 kilobytes of storage per CPU core.

## B.3 Running the Attack from Ireland

The code for the attack is written in C/C++. It is not dependent on a specific number of CPUs but adapts to the number provided by the scheduler. There is no dependency between nodes during execution. This limited communication between nodes limits the network and memory loads.

Shared access were granted to the *Stokes* cluster at ICHEC [ICH] for a total of 500,000 cpu-hours. This cluster has 2560 cores in 320 nodes (each computer node has two Intel (Harpertown) Xeon E5462 quad-core processors and 16GB of RAM). There is a total of 5120GB of RAM available for jobs.

The nodes are interconnected via two planes of ConnectX Infiniband (DDR) providing high bandwidth and low latency for both computational communications and storage access. Storage is provided to the compute nodes via a Panasas ActiveStor 5200 cluster with 84TB (formatted) of capacity using the PanFS filesystem.

Using the *ssh* facility to connect to the ICHEC [ICH] inside the HEAnet network [HEA] and submit the jobs for execution. The ICHEC cluster uses the *qsub* command, which are

submitted via a *PBS* file.

Since the code is embarrassingly parallel, we can request any number of cores in the cluster <sup>1</sup>. To increase the chances of getting a job in the queue to be active, we restricted a job to 128 cores and 24 hours of execution. The service level provided permits up to 256 cores, but, in empirical tests, a job of small size is usually moved forward in the jobs queue; however, we prefer a job size handy enough for micro-managing the attack: this comprises 1 or two connections per day to the cluster, 1-3 jobs always in the queue, and one results upload process per week.

There are days when there is no active job; this is a shared cluster that depend on the demand of other users. The output data was around 90 MB of effective data every per week.

The following is the system usage of a single 128 core job running for 24 hours. The job shown has been working for almost 17 hours working.

```
ldominguez@stokes1:~> qutil -j 236304
=====
Job No: 236304   Queue: ShortQ   Username: ldomingu   Time: 16:50 / 24:00

Node   | Load 1 | Load 5 | Load 15 | Mem Used (MB)
-----|-----|-----|-----|-----
M r4i1n15   8.01    8.02    8.00     262.8
  r4i1n11   8.00    8.00    8.00     191.9
    r4i0n1   8.00    8.00    8.00     173.4
  r3i3n13   8.00    8.00    8.00     176.1
    r3i3n7   8.00    8.00    8.00     182.4
    r3i1n9   8.00    8.00    8.00     173.0
  r1i3n10   8.00    8.00    8.00     186.2
    r1i3n8   8.00    8.01    8.00     185.5
    r1i3n3   8.00    8.00    8.00     175.0
    r1i2n8   8.07    8.04    8.00     179.7
  r1i1n11   8.00    8.00    8.00     207.0
  r1i1n10   8.00    8.01    8.00     186.7
    r1i1n5   8.00    8.00    8.00     203.4
  r1i0n11   8.00    8.00    8.00     180.7
  r1i0n10   8.03    8.03    8.00     189.9
```

<sup>1</sup>We can request nodes in multiples of 8, as required by the system.



```
rli0n1      8.00      8.00      8.00      203.4
```

Current efficiency = 1.00

Each core requires the following cpu cycles per iteration:

```
157695355606446 cycles for 274877908992 iterations; 573 cycles/iteration
90542150916 cycles (0.0574159%, high level) for 12958 reads; 6987355 cycles/read
963428441822 cycles (0.610943%, high level) for 2147650530 weights; 448 cycles/weight
155751883910343 cycles (98.7676%, high level) for 2147483664 ecs; 72527 cycles/ec
```

Currently, we have no peer-reviewed publication other than [BBB<sup>+</sup>09b] as we are still awaiting positive results. The team expect to publish at a major conference on the completion of the attack.

THE cryptographic pairing functions have received a lot of attention in recent years. Particularly, the implementation issues associated with pairings have been of particular focus. Recent work [DSD07, GS08, SBC<sup>+</sup>09b, BS09] has focused on implementing pairings over the BN curves [BN06].

The BN curves are a well-known family of pairing friendly elliptic curves. They are easy to implement and are suitable for 128- and 192-bits security levels [FST10]. They are not the only family of curves suitable for pairings; there are numerous other constructions, including the new families of pairing friendly elliptic curves introduced by Kachisa, Schaefer and Scott in [KSS08].

In [DKS], Dominguez, Kachisa and Scott analysed some issues related to the implementation of pairings over the KSS family of curves with embedding degree  $k = 18$ . We decided to present an online tutorial to help the reader of the paper to understand the concepts.

From Duhrkopf [Duh93], a tutorial should be constructed as a computerized course, an online version of a book or article, with the same chapters. It should include links to the source. Also, some graphic media, such as graphs or videos may be included. The most important part is that it must include interaction: if a question is not answered correctly, a link to the corresponding topic should be provided.

It is not practical to ask the reader to implement the pairing function to determine his level of understanding as he may not have a copy of the Magma software, which was used in the Dominguez, Kachisa and Scott paper. In a tutorial, we can show the reader how the computation is being done, and provide some interaction, such as: what happens if we choose this other polynomial? Can I have a different  $x$ -parameter of the curve? Is the Miller loop at the R-ate function really the shortest?

This chapter is organized as follows: In §C.1 we explain the need for an online tutorial. In §C.2 we introduce the Magma Online Calculator. In §C.3 the concept of Asynchronous JavaScript and XML is introduced, and §C.4 presents the YUI library, which are a set of technologies required to provide basic interaction and code execution for our online tutorial. The construction of our tutorial is detailed in §C.5. A discussion on the computations that are related to the user interaction is presented in §C.6. Finally, the closing remarks on this tutorial are presented in §C.7.

## C.1 Introduction

The result of joint work with Dominguez, Kachisa and Scott is a step-by-step tutorial where the reader is able to take pieces of the code shown in the paper and try them directly on their computer.

There was an issue with the tutorial, that the user required a license of use for the Magma software, which not everybody can afford. This is a major draw-back as the code presented is in Magma and its translation into another computer language will require some expertise from the reader; whereas the target audience was not limited to expert computer programmers, but more generally, for people interested in pairings [DKS]. Regardless of this restriction, most of the code in the paper can be executed with the Magma Online Calculator. The Magma Calculator is a webpage which hosts a small script which takes small Magma programs and executes them. This service, however, is restricted to 20 seconds of computation and has a memory limit. Unfortunately, some parts of the tutorial require more computing time than the 20 seconds allowed in the webpage, hence, we decided to create a small web site where all of the related computations can be executed.

To access the online tutorial, please visit the [Implementing Cryptographic Pairing, the Online Tutorial](#) website at: [Dom09]. To retrieve a copy of the tutorial, please refer to [DKS].

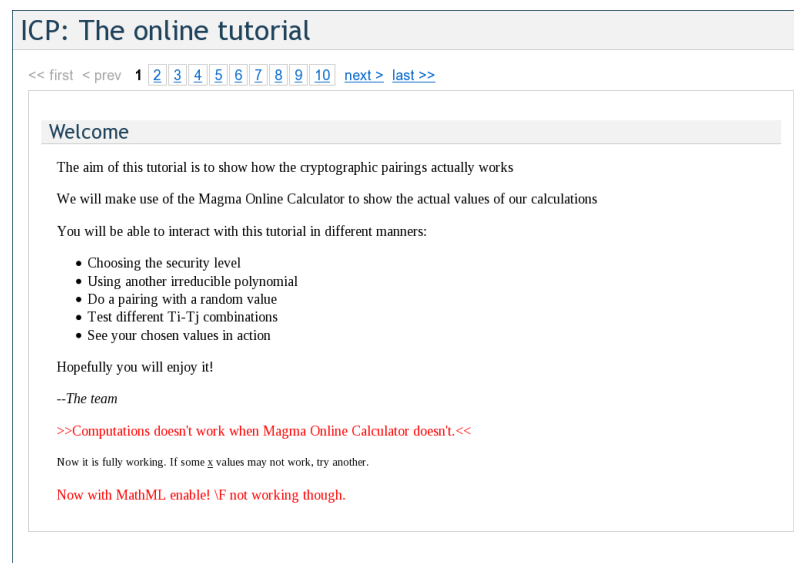


Figure C.1: Screenshot of the welcome screen.

## C.2 Magma Online Calculator

The algebraic computer system Magma is designed to provide a software environment for computing mathematical structures. The software lets the user define his own structures presenting a playground where one can test many types of maps and transformations, potentially to discover new properties on them. Magma is also the programming language used to interact with the computational system. The language is interpreted, and the Magma distribution is the only implementation of the language available.

Magma is a non-commercial system. However, they are not sponsored and require a means to recover the costs incurred through the elaboration and maintenance of the system; Magma therefore requires the purchase of a license. For an educational institution, the corresponding price is attractive and should represent only a minor cost for any university. For those who cannot afford it, there is a student version available and also a special discount to universities located in newly industrialized and developing countries, which should suffice for its acquisition if desired. The student version, however, has some limitations which we are not going to discuss in the present work. Finally, there is a preview version available

online: the Magma Calculator.

The source code for a similar system is available online. We were able to setup our own private Magma Online Calculator server with less restrictions: this gave us more time and memory. There were two problems with this setup: Not only did the computer used have less computing power than the Magma server, but we also believe that this is not covered by the Magma license.

Nonetheless, we are allowed to make use of the Magma Calculator to any extent, as long as we conform to the memory and time limits. We decided to construct an online tutorial using the Magma Calculator for those who cannot afford a license themselves. This tutorial is intended for demonstration purposes only.

Our tutorial will make use of the AJAX technology and the YUI library, which are described in the following sections. The use of AJAX is because it permits to automate the interaction of the webpage with the Magma Online Calculator, whereas, the use of YUI is because of its maturity by the time of this research, as it will be pointed out later.

### C.3 AJAX

AJAX is an acronym which stands for Asynchronous JavaScript and XML. AJAX is not a technology, but a set of several technologies, such as XML, XMLHttpRequest, CSS and DOM, combined with JavaScript [Gar05a].

Here, we introduce the components of the AJAX technology [VH07]:

**Definition 32.** *The Extensible Markup Language (XML) is a W3C recommendation for a general purpose markup language that has no predefined tags. It is a simplified subset of the Standard Generalized Markup Language (SGML).*

**Definition 33.** *An XMLHttpRequest is a JavaScript class with specific protocols and methods which allow HTTP requests in JavaScript code and it is available in most internet browsers.*

**Definition 34.** *A Webservice is an extension of the client/server paradigm to the web. The details of a webservice are not relevant to this thesis.*

**Definition 35.** *The Cascading Style Sheets (CSS) allow the definition of styles for the elements of structured documents which may make use of XML.*

**Definition 36.** *The Document Object Model (DOM) is the description that tells the browser how to represent HTML elements as a tree structure of objects.*

We can produce applications in the client/server model using the internet browser as the client side. Traditionally, an internet browser was used to interact in a server-side paradigm. With the advent of the AJAX technologies it is possible to implement a client/server paradigm with an internet browser as a client. Traditionally, this was possible with the use of external plug-ins, such as *ActiveX*, *Adobe Flash*, or *Mozilla Chrome*.

As mentioned in the previous section, a user without a Magma license cannot use to the full extent the tutorial we developed; a client/server model does not seem appropriate to this scenario.

Using the Magma Online Calculator we can execute some Magma code, but in a very limited way, and so, following the examples in our tutorial using the online calculator would be painful. It is, however, possible to automate the use of the Magma Online Calculator to run the examples from our tutorial. A client/server model can be implemented, where the client is our internet browser, and with the use of the AJAX technologies have some interaction with the examples. We leave the interaction with the Magma Online Calculator to the client side. This way, the reader is able to try the examples from our tutorial without the difficulty that prevents the use of programs that do not fit the restrictions of the online calculator.

We stress that the intention of our client is to demonstrate the pairings in Magma and we encourage the reader to construct more complex programs, which will require a full Magma license.

## C.4 YUI

YUI stand for *Yahoo! User Interface* Library. It is an open source collection of utilities and controls. It is written mainly in JavaScript but there is also some code in CSS [We108].

The use of a JavaScript library can save the web developer a great amount of effort and hides some of the tricky parts of the AJAX program implementation. Since the focus of this research is not web programming, we decided to use an already established and mature library.

When we started the design of our online tutorial (2008), YUI appeared to be the most mature of the free AJAX libraries.

Our implementation of the tutorial is based on YUI 2.0. At this time YUI 3.0 (which is the current version) was already available. However, since it was an early beta version we took the mature option. This version of the library was already well documented.

### C.4.1 YUI Structure

The core of the YUI library consists of the following three components:

- Yahoo Global Object,
- DOM collection,
- Event utility.

The *Global Object* establishes a namespace where all of the operations are linked as a whole application. *DOM collection* permits access to the elements in their class or context, and handles the details of the different implementations of the DOM in the different internet browsers, which varies from vendor to vendor.

In addition to the core elements, we used the following plug-ins:

- Paginator,
- Panel container,

- Dialog.

## Paginator

The purpose of the Paginator Control is to reduce the size of a webpage by breaking large sets of data into discrete pages. It offers a control to navigate around the mini-pages [[Yah](#)].

In the case of our tutorial, we decided to use the Paginator to separate the different stages of the implementation. The parts we defined for our project are as follows.

1. Welcome.	8. Point Generation.	16. R-ate pairing construction.
2. KSS Curves.	9. Tate Pairing.	
3. Security level.	10. Pairing Execution.	17. Final Exponentiation.
4. Curve Generation.	11. The ate pairing.	18. Hashing to $G_2$ .
5. Twist of a Curve.	12. Frobenius map.	19. Multiexponentiation in $G_2$ .
6. Sextic Twist.	13. Pairing Execution.	
7. Irreducible Polynomial.	14. R-ate pairing.	20. Comparison.
	15. Pairing Execution.	21. Conclusion.

The purpose of the sections are either to explain or compute something in the tutorial, and to organize the online tutorial with the same structure as the paper version of our tutorial. Each page, or slide, is contained in separate HTML/PHP files. In the case of slides that require interaction with the server, for example: the pairing execution or curve construction, comparison, etc., some extra code is needed.

## Panel and Dialog Container

The Overlay control facilitates the creation of modular content absolutely positioned above the page. The Panel container, which is derived from the Overlay control, acts as an operating system window where we can position other elements.



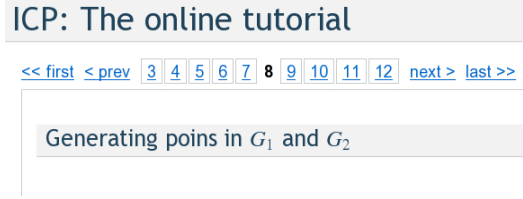


Figure C.2: Screenshot of the paginator

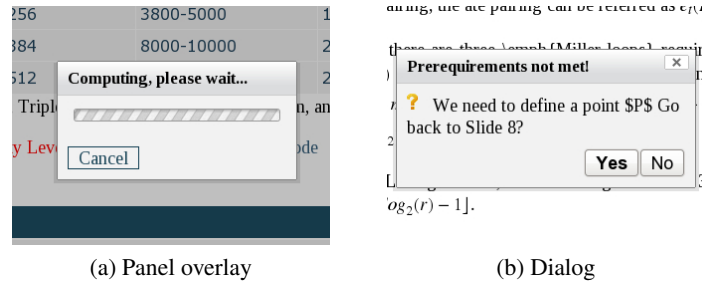


Figure C.3: Screenshot of the Panel and Dialog

In the case of the tutorial, we place a panel overlay on top of the screen when the computations are being executed by the remote server. This way we avoid queuing the server while there are computations taking place.

The dialog lies on top of the panel and it is used in the tutorial to interact with the user mainly reacting to illegal operations, such as: trying to generate an elliptic curve before defining its parameters, or requesting a comparison of the three pairings supported before they are actually constructed.

## C.5 Our Construction

For the tutorial construction, we followed the requirements from [Yah] for setting up the basic environment: inserting the links for the JavaScript code, the CSS files, the plug-ins, and the setup of our YUI webpage environment.

We defined an “id” for each individual slide and defined the variables to store our computations: the  $x$ -parameter, the field size, the subgroup, the equation of the elliptic curve (the  $b$  parameter), the irreducible polynomial, the points and the parameters of the R-ate

pairing. For simplicity, the variables to store are defined as plain text. Since the tutorial is meant to be consulted using a computer or a high-end internet device, memory consumption is not an issue here.

As mentioned before, the Magma Online Calculator has a time and memory space restriction. In the beginning, we find the parameters of the curve in the initial slides. The next computations constructs the corresponding function with the already stored parameters in the client (the internet browser, in plain text). For example, once we have obtained an  $x$ -parameter and an irreducible polynomial, computing a pairing will require the construction of a magma program which includes the finite fields, explicitly expressed, and a randomization of the points involved. For the code construction, we used a similar approach as in 2.8.4 and send the code to the server using an XMLHttpRequest.

### C.5.1 The XMLHttpRequest

In Listing C.1 we present the preparations for the code execution. The virtual HTML element with “id=entrada” contains, at that point, the code in Magma to be executed by the server. We added, at the end of the element “(d.value+=...)”, the instructions of what to do with the code, in this case we are requesting the generation of a KSS  $k = 18$  curve and for the parameters to be stored into the Magma Calculator memory, following a “Printf” command with the formatted output to retrieve them.

Listing C.1: Preparing the XMLHttpRequest

---

```

if(httpObject.readyState == 4){
    document.getElementById("entrada").value = httpObject.
    responseText.replace(/put/g, "");
    var d=document.getElementById('entrada');
    d.value+=' \np,r,t,c,x,h:=KSSCurves(' +old+', ' +nextbX+');Printf("<
p:%o>\n<r:%o>\n<t:%o>\n<o:%o>\n<x:%o>\n<h:%o>",p,r,t,c,x,h)';
    getAnswer(setAnswer3);
}

```

---

The code delivery and interaction with the Magma Calculator is done by our webserver and not by the client itself, this is because we are implementing a multi-layer client/server environment. We need this computing model because it is a common well-known practice

to hide the computer network in a company inside a proxy server, and the requests may not go through the network restrictions of the end user. The function “getAnswer” uses the code at Listing C.2 to send the requests and to receive the responses from the Calculator.

Listing C.2: The XMLHttpRequest and Response

---

```

ini_set('display_errors',1);
error_reporting(E_ALL);
require_once "../HTTP/Client.php";
$headers='';
$params['proxy_host'] = 'your_proxy';
$params['proxy_port'] = 21;
$client = &new HTTP_Client($params, $headers);
$client->enableHistory(false);
if (strcmp($_POST["commands"], '')) {
    $to_post=array('commands'=>$_POST["commands"]);
} else {
    $to_post=array('commands'=>'x:=1;x;');
}
$client->post( "http://magma.maths.usyd.edu.au/calc/", $to_post );
$res = $client->currentResponse();
$body = $res['body'];
$found=array();
preg_match_all("/id=\"output\">(.*?)</textarea/smU", $body, $found
);
echo $found[1][0];

```

---

The code in Listing C.2 shows how can we have a proxy server within our internal network. It is a common well-known practice in IT departments to stop traffic leaving the network from web scripts. This way, in case of an infection, we prevent the malicious code attacking external sites. In this case, the code connects from and into our local network, which is safe. Typically, the local network is only actively monitored by the company’s proxy server upon suspicious activities. It is safe to use the network resources from our script as this is a constructive use.

The “proxy\_host” and “proxy\_port” varies from network to network and must be changed prior the use. The reader who would like to implement his own online tutorial should ask to their IT department for further details. We hand-coded the link that points to the Magma Calculator, but it can be easily changed into something else.

### C.5.2 Reading the Output

The output code from C.1, requires the use of regular expressions. This is supported in JavaScript and creates no major issue for the programmer. We include the Listing C.3, which shows the typical code for extracting the response from the server. The first line detects the error messages sent from the server and terminates the execution of the script. Line 4 breaks the response into a matrix, each containing one output value. The next lines assign the elements in the matrix to their corresponding variable. In the few last lines of the code we assign a null value to the variables.

Listing C.3: Parsing the output response from the Magma calculator

---

```

if (e.match(/WARNING/g)) { terminate(); alert('Please, _try_with_
    another_x-parameter'); }
else {
    e=e.match(/<\w:[0xA-F\d]+\s*[0xA-F\d]*\s*[0xA-F\d]*\s*[0xA-F\d]*>/g
    );
    ec_p=e[0].replace(/\D*/g, "").match(/\d+/g);
    ec_r=e[1].replace(/\D*/g, "").match(/\d+/g);
    ec_t=e[2].replace(/\D*/g, "").match(/\d+/g);

    ...

    ec_b=null;
    ec_chi=null;
    ec_P=null;
    ec_Q=null;

```

---

### C.5.3 Displaying the Results

Once the computation is finished, we can later assign the corresponding values into the DOM structure of our web page: “oCell.innerHTML=ech;”. In this example, the JavaScript code writes the Hamming weight  $h$  into a cell in a table.

The code shown in Listing C.4 is invoked every time the user enters the Security Level slide from Table C.4.1 if we already have computed the parameters of the curve. This code is only for displaying purposes and makes no computation. The “wrapping2(.)” function in the second last block of code of the listing is just to ensure that the value stays inside the table. See Figure C.4 for a sample output.

## Listing C.4: Displaying the results from the computation

```

function showprtcx() {
  var d=document.getElementById('idSlide');
  d.parentNode.removeChild(d);
  d=document.createElement('p');
  d.setAttribute('id','idSlide');

  var vTable = document.createElement("table");
  vTable.setAttribute('border','1');
  vTable.setAttribute('cellpadding','1');
  vTable.setAttribute('class','results');
  vTable.setAttribute('width','80%');
  vTable.setAttribute('style','table-layout:auto');

  ...

  var vTBody = document.createElement("tbody");
  oRow = document.createElement("tr");
  oCell = document.createElement("td");
  oCell.setAttribute('style','text-align:center');
  oCell.innerHTML='p(x)';
  oRow.appendChild(oCell);
  oCell = document.createElement("td");
  oCell.innerHTML=wrapping2(ec_p);
  oRow.appendChild(oCell);
  vTBody.appendChild(oRow);

  ...

  vTable.appendChild(vTBody);
  d.appendChild(vTable);
  document.getElementById('elementpage').appendChild(d);
}

```

Variable	
p	5873078483844521707805606
r	1078995676249030927351773
t	1586516194664014881453358
c	5443097329417863331821835
x	5772788380868
ham(x)	12
<div> <div>&lt;&lt;</div> <div>x</div> <div>&gt;&gt;</div> </div>	

Figure C.4: Screenshot of the displayed stored values

## C.6 Internal Comparisons

One of the purposes of the online tutorial was to benchmark the pairing construction for the parameters of the user. The idea was to make evident how the choice of the parameters affects the speed of the pairing computation.

In the actual version of the online tutorial, there are benchmarks for the following operations:

- Final Exponentiation,
- Hashing to  $G_2$ ,
- Multiexponentiation in  $G_2$ ,
- Comparison of pairings,
- R-ate pairing construction.

For the final exponentiation, we time our implementation, given in [SBC<sup>+</sup>09b], of the computation of a particular pairing against a non efficient implementation of the same pairing. For the hashing to  $G_2$ , we timed the traditional way against our method [SBC<sup>+</sup>09a]. For the multi-exponentiation, we used the [GS08] method for the KSS  $k = 18$  family of curves.

The pairing comparison is done by comparing the Miller loop length of the Tate, ate and R-ate pairings; this depends on the  $x$ -parameter chosen and the R-ate pairing construction.

Miller-length in iterations	
$e_r(P, Q)$	244
$e_t(P, Q)$	165
$e_{\{A, B\}}(P, Q)$	39

Figure C.5: Screenshot of the Miller loop length computation for a random KSS:  $k = 8$  curve.

Finally, the R-ate pairing comparison is done by calculating the different Miller loop lengths. Figure C.5 presents a comparison of the Tate, ate and R-ate pairings.

Polynomials (click on them)			
Ti	A	Tj	B
4	3	1	x
17	$3/7*x$	1	$2/7*x^2$
2	$5/49*x^2$	3	$3/49*x^2$
2	$5/49*x^2$	6	$8/49*x^2$
5	$8/49*x^2$	6	$3/49*x^2$
13	$3/7*x$	6	$2/7*x$
7	$2/7*x$	12	$3/7*x$
17	$3/14*x$	7	$1/14*x^2$
14	$8/49*x^2$	12	$5/49*x^2$
17	$3/49*x^2$	12	$8/49*x^2$
17	$3/49*x^2$	15	$5/49*x^2$

Pairing	
Bilinearity	true
Non-degeneracy	true
Your selection would take: 0% <u>less</u> doublings than our proposed TITj combination,	
however it would require: 0% <u>less</u> additions in the Miller loop.	

(a) Good choice

Polynomials (click on them)			
Ti	A	Tj	B
4	3	1	x
17	$3/7*x$	1	$2/7*x^2$
2	$5/49*x^2$	3	$3/49*x^2$
2	$5/49*x^2$	6	$8/49*x^2$
5	$8/49*x^2$	6	$3/49*x^2$
13	$3/7*x$	6	$2/7*x$
7	$2/7*x$	12	$3/7*x$
17	3/14*x	7	1/14*x^2
14	$8/49*x^2$	12	$5/49*x^2$
17	$3/49*x^2$	12	$8/49*x^2$
17	$3/49*x^2$	15	$5/49*x^2$

Pairing	
Bilinearity	true
Non-degeneracy	true
Your selection would take: 100.00% <u>more</u> doublings than our proposed TITj combination,	
also it would require: 100.00% <u>more</u> additions in the Miller loop.	

(b) Bad choice

Figure C.6: Screenshot of the R-ate pairing comparison

## **C.7 Final Thoughts about the Online Tutorial**

This online version of the tutorial aims to demonstrate an actual implementation of a pairing function. We have dealt with the problem of the end user not having a Magma license and we have overcome the restrictions of the Magma Online Calculator.

Our online tutorial is an alive webpage. In a future version, we intend to include support for other pairing-friendly elliptic curves, use seconds for the speed comparison, other multi-exponentiation methods, among other ideas.

We have used a mixture of PHP and JavaScript code for the transmission of the data. There is Magma code for the pairing function execution. The client stores the parameters of the curve, field and pairing construction, and the user can modify the pairing construction.

Since the main page of the tutorial is based in JavaScript, anybody can view and download it. The PHP code for the transmission is shown in Listing C.2. We stress that this code can be used as a framework for constructing Online Calculators for other programming languages.



# D CODE GENERATOR FUNCTION LIST

In this appendix there is a selection of functions, procedures and their input/output parameters from the code generator.

Table D.1: List of functions with Input/Output description for the Code Generator

Filename	Function name	Input	Output
getchain	getPos	$f$ element to find, $c$ sequence	$i$ in $f = c_i$
	CreateVectorChain	$c$ proto-sequence, $d = \{1, 2\} \cap c$	vector $e$ with solved elements, $f = c \cap \{e\}$ , sequence $\{e\} \cup f$
approximatechain	max_two_elements	elements $u, d, v$	$\{u, d, v\} \cap \min\{u, d, v\}$
	Approximatechain	$c$ chain	$w$ sequence
hypermutation	findPos	$f$ element to find, $c$ chain	$i$ in $c_i \leq f < c_{i+1}$
	MyInclude	$e$ element, $c$ chain	$c \cup \{e\}$
	DirectSolvable	$e$ element, $Cl$ sequence	true if there exists some $i, j$ such that $e = Cl_i + Cl_j$
	hypermutation	$Cl$ sequence, $r = \text{low/high}$ , $c$ proto-sequence	new $Cl$ version containing upto 1 $Cl_i$ not in previous $Cl$
	AllSolvable	$Cl$ sequence	Vector $v$ containing $v_i = 1$ if there is no such $i, j$ that holds $Cl_k = Cl_i + Cl_j$ for $0 \leq k < \#Cl$ . $f = \#v$
	DetectUnused	$Cl$ sequence, $c$ proto-sequence	Vector $v$ constructed as $v_i = 1$ if $Cl_i \in B$ , where $A = \{Cl_k \cap c\}$ and $B \cup c$ for all $k$ . If there exists $n$ such that $Cl_i = A_n + Cl_j$ then $A \cap \{Cl_i, Cl_j\}$ and $B \cup \{A_n, Cl_j\}$ for all $n$ .
	CountOperations	$Cl$ sequence	A counter $d$ for $i = j$ and a counter $a$ for $i \neq j$ in $Cl_k = Cl_i + Cl_j$
mutate	AtomicInclusion	$f$ element, $C$ sequence, $c$ proto-sequence	$C \cup \{C_i + C_i\}$ if $f - e = C_i + C_i$ and $e \in C$ . If $e \notin C$ , but $e = C_i + C_j$ then $C \cup \{e, C_i, C_j\}$
	Mutate	$M$ number of mutations to execute, $G$ number of chains to mutate, $c$	proto-sequence, a $\#c \times G$ matrix $w$ with for all $W_i = c$ & $c \times n$ matrix $s$ with the sortest elements.
Continued . . .			

Table D.1: (continued)

Filename	Function name	Input	Output
vectorize	AtomicInclusion	$f$ element, $C$ sequence, $c$ proto-sequence	$C \cup \{C_i + C_i\}$ if $f - e = C_i + C_i$ and $e \in C$ . If $e \notin C$ , but $e = C_i + C_j$ then $C \cup \{e, C_i, C_j\}$
	Mutate	$M$ number of mutations to execute, $G$ number of chains to mutate, $c$	proto-sequence, a $\#c \times G$ matrix $w$ with for all $W_i = c$ & $c \times n$ matrix $s$ with the shortest elements.
getVector		$s$ sequence, $c$ proto-sequence	matrix with a generic code sequence for $s$
reduceterms	setpendings	$m \times n$ matrix $f$ with code sequence	$m \times n$ matrix $u$ with $u_{i,j} = 1$ for every $f_{i,j}$ containing an $x_i$ element.
	countpendings	$m \times n$ matrix $u$ , as output from the previous function	#elements in $u$ equal to 0
	IsUsedBefore	$fe$ code sequence, $i$ , target= $t_j$ , $u$ as in the previous function	true if the target element is used as a RH-value in $fe_k$ , where $0 \leq k < i$
	GetNext	$fe$ , $u$ and $i$ as in the previous function	An element $t_j$ not used in $fe_k$ , where $0 \leq k < i$
	SetToCurr	$fe$ , $u$ and $i$ as in the previous function, $t_j$ target, $t_k$ new element	For all $fe_k$ containing $t_j \leftarrow t_k$ , and corresponding $u_k \leftarrow 1$ , where $0 \leq k < i$
	getCurr	Current $i$	$t_i$
	getNewPrev	$m \times n$ matrix $u$	$i$ if $u_i j = 0$
	reduceterms	$fe$ code sequence	$fe$ with less distinct $fe_{ij}$ elements
	generateCodeFE—CF *FMagma	$fe$ code sequence	Corresponding Magma Code, used in the code generator
	getimaxdegree	$\lambda$ as in Equation 4.3	max degree of $x_i$
	associateXiFEMagma *CFMagma	$\lambda$ as in previous function, $c$ proto-sequence	Magma code for the pre-computation, used in the codegenerator
Continued . . .			

Table D.1: (continued)

Filename	Function name	Input	Output
codegenerator	GenericAssign	LH and RH operands, suffix position, operators, target library	$LH \leftarrow RH$
	GenericOperation	LH, RH and RH2 operands, suffix position, operators, target library	$LH \leftarrow RH \circ RH2$
	Assign	LH and RH operands, target library	$LH \leftarrow RH$
	Accumulate	LH and RH operands, target library	$LH + = RH$ code
	Normal	LH operand, target library	Code for normalize a point
	genericTransformation	LH operand, suffix operator, operator, target library	Code to Transform LH; i.e., -P
	genericTransformation2	LH operand, suffix operator, operator, target library	Code to Transform LH; i.e., $\text{psi}(A, d2, d3, 6)$
	AssociateXi	$\lambda$ as in AssociateXiFEMagma, $c$ proto-sequence	For every operand of a given $\lambda_i$ , print a vector=[LH, degree of the $x$ exponent, sign of the exponent, degree of the Frobenius operator]. For example, given: $x_3 = \pi^2(-f^{x^4}).f^{x^7}$ , output vectors: [[x3,4,-,2],[x3,7,+,0]]
	getNextXiCode	Partial codesequene, target library	code to compute $x_i$
	getXiCodeM	code sequence, element to output, target library	vector with code to compute
	invertedElement	$x_i$ element to invert, suffix position, operator and target library	Inverted $x_i$ code; i.e., “negate(P)”
	degreeElement	same as invertedElement, plus a flag if we need to invert the element	exponentiation code for $x_i$
Continued . . .			

Table D.1: (continued)

Filename	Function name	Input	Output
	Precompute	$x_i$ max degree, target library	code to compute $x_i$
	Try-and-Catch	$x_i$ max degree, total number of $t_j$ , target library	$x_i$ code with corresponding try-and-catch operators
	Finalize	Same as in previous function	Finalize code; i.e., freeing memory, return value, etc.
	getAllXiCode	Code sequence, target library	Magma code for the $x_i$ assignments, used by the code generator
	tagXi	Code sequence	just-in-time macro instruction for the code sequence
	getAllCode	Code sequence, pre-computing code sequence, target library	Unified Code
WriteFEMagma		$x_i$ and final exponentiation code	a final exponentiation code loadable into memory
getparameterscurve	ham. Hamming Weight		
	getprtcx	suggested $x$ -parameter, $n$ and $b$ parameters of the family of curves	$p, r, t, c, x$ and the Hamming-weight of $x$
	CurveGenA and B	$p, r, c$ parameters of the curve	$E, \mathbb{F}_p, a, b$ in short Weierstrass form and a point
LowHamming	SearchLowHammingweightX	$M, \text{level}, p, r, t, n, b$	Low Hamming-weight $x$ -parameter of the security level
	GetRandomX	$\text{level}, p, r, t, n, b$	$2^i + 2^j + c$ of the desired security level
SetCF		$(p, k, t)$ defining the Elliptic Curve	$h$ , the cofactor
lambdaCF		$p, t, k$	$\lambda$
WriteCFMagma		Same as WriteFEMagma but for hashing to $G_2$	
Continued . . .			

Table D.1: (continued)

Filename	Function name	Input	Output
Rewriter		Input/Output code Filenames, $k, d$ degree of the twist, $mt$ degree of the intermediate field between the twist and the extension field, target library and elliptic curve	Code with template tags rewritten
Pack			Compressed file containing the necessary code
getLineFunction	SetLineFunction		Vectors containing Reversed Polish notation of the line function, and a table matching an arithmetic symbol to its operator name
	LookUpOperators	$o$ operators table, $t$ search target	$i$ in $o_i = t$
	getLineCode	Line code in reversed polish notation, Operator table, line function type, target library	vector containing line function
Towering	SetTowering		Default extension field constructions
	GetMidTower	extension field construction, $k$	extension degree between $k$ and the twist
	GetAlphaBeta	$\Delta$ discriminant, $k$ , $p$ , $\chi$ -parameter used by Magma	Irreducible polynomial
	WriteTowering	Filename, Irreducible Polynomial	File rewritten
	GetToweringType	extension field construction, $k$ , twist	quadratic then cube, cube then quadratic extension field construction order of the fields
TowerDependant	RewritePSI	Filename, extension fields, extension field construction order, target library	psi function rewritten
	RewriteH2	Filename, extension fields, target library	H2 function rewritten
	Rewrite ppowerfrobenius	Filename, extension fields, target library	$p$ =power Frobenius function rewritten
Continued . . .			

Table D.1: (continued)

Filename	Function name	Input	Output
	TowerDependant	Filename, extension fields, extension field construction order, target library	See previous functions
getFinalExpo	getHex	$e$ element	$e$ in hexadecimal representation
	RemoveOx	$e$ in hexadecimal representation	$e$ in implicit hexadecimal representation
	WriteFinalExponentiation	curve, extension fields, initialization code, pre-compute code, code, finalization code, target library	Final Exponentiation code
	FinalExponentiation	parameters of the curve, parameters of the generator, user defined chain, Magma code for the Final Exponentiation, target library	Final Exponentiation code
getFastHashing	WriteFastHashing	Same as in the Final Exponentiation	Idem
	FastCofactor	Same as in the Final Exponentiation	Idem
getRatePairing	GetPreInitMap	target library	Initialize variables
	ReusableMillerM2	R-ate parameters of the curve, $x$ -parameter	true if we can reuse intermediate values of the Miller loops
	WriteRate	Same as in the Final Exponentiation	Idem
getIrreducible	GetnpFpd	$(p, k, t)$ defining the Elliptic Curve	The number of rational points on the twisted curve over $\mathbb{F}_{p^3}$
	GenerateFpk	flag, finite field $F_{p^d}$ , polynomial $\chi$	The extension field
	GenerateExtT	flag, finite field $F_{p^d}$ ; $a, b$ from the EC definition; polynomial $\chi$	The EC in the twist ( $E'$ )
	GeneratePointTw	twisted and extended EC's; polynomial $\Delta$ ; $r$ , number of points	a point $Q$
Continued . . .			

Table D.1: (continued)

Filename	Function name	Input	Output
	GetIrreducible	parameters of the curve	Irreducible polynomial, and Elliptic curve and finite field object for Magma
millerRGengood2	pairing (generic R-ate pairing)	$P \in G_1, Q \in G_2$ , R-ate pairing parameters to test	$G_T$
TiCombo	GetTiTjCombo	parameters of the curve	$m \times n$ matrix containing the R-ate parameters
ChooseTiTj	ChooseTiTj	$m \times n$ matrix containing the R-ate parameters	OptimalTiTj index for the previous matrix with the recommended R-ate parameters
millerate and millerTate	generic ate and Tate pairing to test	$P, Q, r$ or $t - 1$	true if bilinear and non-degeneracy
WriteHardExpo	WriteHardExpo	Magma code for the Final Exponentiation	File written
The End			



# BIBLIOGRAPHY

- [AF96] Niclas Andersson and Peter Fritzson. Overview and industrial application of code generator generators. *Journal of Systems and Software*, 32(3):185–214, 1996. One citation on page 39.
- [AP] Diego F. Aranha and Conrado Porto Lopes Gouvêa. RELIC: an efficient library for cryptography. <http://code.google.com/p/relic-toolkit/>. One citation on page 38.
- [ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers. Principles, Techniques, and Tools*. Computer Science. Addison-Wesley, 1986. There are 2 citations on pages 39 and 103.
- [Bab86] László Babai. On Lovasz lattice reduction and the nearest lattice point problem. *Combinatorica.*, 6(1):1–13, 1986. There are 2 citations on pages 46 and 48.
- [Bar] Paulo S. L. M. Barreto. Personal communication. One citation on page 50.
- [BBB<sup>+</sup>09a] Daniel V. Bailey, Brian Baldwin, Lejla Batina, Daniel J. Bernstein, Peter Birkner, Joppe W. Bos, Gauthier van Damme, Giacomo de Meulenaer, Junfeng Fan, Tim Güneysu, Frank Gurkaynak, Thorsten Kleinjung, Tanja Lange, Nele Mentens, Christof Paar, Francesco Regazzoni, Peter Schwabe, and Leif Uhsadel. The Certicom Challenges ECC2-X. *SHARCS'09 - Special-purpose Hardware for Attacking Cryptographic Systems*, pages 51–82, 2009. <http://www.hyperelliptic.org/tanja/SHARCS/record2.pdf>. One citation on page 152.
- [BBB<sup>+</sup>09b] Daniel V. Bailey, Lejla Batina, Daniel J. Bernstein, Peter Birkner, Joppe W. Bos, Hsieh-Chung Chen, Chen-Mou Cheng, Gauthier van Damme, Giacomo

- de Meulenaer, Luis Julian Dominguez Perez, Junfeng Fan, Tim Güneysu, Frank Gurkaynak, Thorsten Kleinjung, Tanja Lange, Nele Mentens, Ruben Niederhagen, Christof Paar, Francesco Regazzoni, Peter Schwabe, Leif Uhsadel, Anthony Van Herrewege, and Bo-Yin Yang. Breaking ECC2K-130. Cryptology ePrint Archive, Report 2009/541, 2009. <http://eprint.iacr.org/>. There are 2 citations on pages 151 and 156.
- [BBH<sup>+</sup>63] D. W. Barron, J. N. Buxton, D. F. Hartley, E. Nixon, and C. Strachey. The main features of CPL. *The Computer Journal*, 6(2):134–143, 1963. One citation on page 40.
- [BC89] Jurjen Bos and Matthijs Coster. Addition chain heuristics. In G. Goos and J. Hartmanis, editors, *Advances in Cryptology – CRYPTO 89 Proceedings*, volume 435 of *Lecture Notes in Computer Science*, pages 400–407, 1989. There are 2 citations on pages 59 and 63.
- [BCP97] Wieb Bosma, John J. Cannon, and Catherine Playoust. The Magma algebra system. i. The user language. *J. Symbolic Comput.*, 24(3–4):235–265, 1997. Computational algebra and number theory (London, 1993). One citation on page 38.
- [BCS97] Peter Bürgisser, Michael Clausen, and Mohammad Amin Shokrollahi. *Algebraic Complexity Theory*. Springer-Verlag Berlin Heidelberg New York, 1997. One citation on page 37.
- [Ber] Daniel J. Bernstein. 2010.08.16 lab sessions: "writing really fast code". <http://cr.yp.to/highspeed/20100816.html>. One citation on page 142.
- [Ber09] Daniel J. Bernstein. Optimizing linear maps modulo 2. *SPEED-CC 2009*, pages 3–18, 2009. Workshop: Software Performance Enhancement for Encryption and Decryption and Cryptographic Compilers. <http://www.>

[hyperelliptic.org/SPEED/record09.pdf](http://hyperelliptic.org/SPEED/record09.pdf). There are 4 citations on pages 59, 60, 62, and 74.

- [BGHS07] Paulo S. L. M. Barreto, Steven D. Galbraith, Colm Ó' Héigeartaigh, and Michael Scott. Efficient pairing computation on supersingular abelian varieties. *Des. Codes Cryptography*, 42(3):239–271, 2007. One citation on page 19.
- [BGM<sup>+</sup>10] Jean-Luc Beuchat, Jorge Enrique Gonzalez Diaz, Shigeo Mitsunari, Eiji Okamoto, Francisco Rodriguez-Henriquez, and Tadanori Teruya. High-speed software implementation of the optimal ate pairing over Barreto-Naehrig curves. In Marc Joye, Atsuko Miyaji, and Akira Otsuka, editors, *Pairing-Based Cryptography – Pairing 2010*, volume 6487 of *Lecture Notes in Computer Science*, pages 21–39. Springer-Verlag, 2010. <http://eprint.iacr.org/2010/354>. There are 2 citations on pages 3 and 144.
- [BLKS02] Paulo S. L. M. Barreto, Ben Lynn, Hae Y. Kim, and M. Scott. Efficient algorithms for pairing-based cryptosystems. In Moti Yung, editor, *Advances in Cryptology – CRYPTO 2002*, volume 2442 of *Lecture Notes in Computer Science*, pages 354–368, 2002. One citation on page 18.
- [BLS01] Dan Boneh, Ben Lynn, and Hovav Shacham. Short signatures from the Weil pairing. In Colin Boyd, editor, *Advances in Cryptology – ASIACRYPT 2001*, volume 2248 of *Lecture Notes in Computer Science*, pages 514–532. Springer-Verlag, 2001. One citation on page 1.
- [BLS02] Paulo S. L. M. Barreto, Ben Lynn, and Michael Scott. Constructing elliptic curves with prescribed embedding degree. *Lecture Notes in Computer Science, Security in Communication Networks. SCN 2002*, 2576:263–273, 2002. One citation on page 124.

- [BMP09] M. Barbosa, A. Moss, and D. Page. Constructive and destructive use of compilers in elliptic curve cryptography. *Journal of Cryptology*, 22(2):259–281, April 2009. One citation on page 103.
- [BN06] Paulo S. L. M. Barreto and Michael Naehrig. Pairing-friendly elliptic curves of prime order. In Bart Preneel and Stafford Tavares, editors, *Selected Areas in Cryptography – SAC 2005*, volume 3897 of *Lecture Notes in Computer Science*, pages 319–331, 2006. There are 8 citations on pages 10, 30, 34, 39, 78, 87, 95, and 157.
- [BS04] Seluçk Baktir and Berk Sunar. Optimal tower fields. *Computers, IEEE Transactions on*, 53(10):1231–1243, oct. 2004. One citation on page 9.
- [BS09] Naomi Benger and Michael Scott. Constructing tower extensions for the implementation of Pairing-Based Cryptography. In M. Anwar Hasan and Tor Helleseth, editors, *Arithmetic of Finite Fields - WAIFI 2010*, volume 6087 of *Lecture Notes in Computer Science*. Springer-Verlag, 2009. There are 5 citations on pages 9, 10, 111, 124, and 157.
- [CBNW10] Craig Costello, Colin Boyd, Juan Manuel Gonzalez Nieto, and Kenneth Koon-Ho Wong. Avoiding full extension field arithmetic in pairing computations. In Daniel J. Bernstein and Tanja Lange, editors, *Progress in Cryptology – AFRICACRYPT 2010*, volume 6055 of *Lecture Notes in Computer Science*, pages 223–240. Springer-Verlag, 2010. <http://eprint.iacr.org/>. One citation on page 126.
- [CCRHC08] N. Cruz-Cortes, F. Rodriguez-Henriquez, and C.A. Coello Coello. An Artificial Immune System heuristic for generating short addition chains. *Evolutionary Computation, IEEE Transactions on*, 12(1):1–24, Feb. 2008. There are 8 citations on pages 3, 59, 63, 65, 67, 68, 69, and 147.

- [Cer] Certicom. Certicom ECC Challenge. [http://www.certicom.com/images/pdfs/cert\\_ecc\\_challenge.pdf](http://www.certicom.com/images/pdfs/cert_ecc_challenge.pdf). There are 2 citations on pages 151 and 153.
- [CF06] Henri Cohen and Gerhard Frey. *Hanbook of Elliptic and Hyperelliptic Curve Cryptography*. Chapman & Hall/CRC, 2006. There are 4 citations on pages 6, 7, 17, and 82.
- [Chu96] Alonzo Church. *Introduction to mathematical Logic*. Annals of Mathematical Studies. Princeton University Press, 10th reprint edition, 1996. Page 38. One citation on page 41.
- [Com94] IEEE Computer Society. Design Automation Technical Committee. *IEEE standard VHDL language reference manual : circuits and devices, communications technology*. IEEE Std ; 1076-1993. Institute of Electrical and Electronics Engineers, 1994. One citation on page 103.
- [CV06] Guilherme P. Coelho and Fernando J. Von Zuben. omni-ainet: An immune-inspired approach for omni optimization. In Hugues Bersini and Jorge Carneiro, editors, *Artificial Immune Systems – ICARIS 2006*, volume 4163 of *Lecture Notes in Computer Science*, pages 294–308. Springer-Verlag, 2006. One citation on page 66.
- [DD01] Harvey M. Deitel and Paul J. Deitel. *C++, How to program*. How to Program Series. Prentice Hall, 3 edition, 2001. There are 3 citations on pages 42, 45, and 103.
- [DH76] W. Diffie and M. Hellman. New directions in cryptography. *Information Theory, IEEE Transactions on*, 22(6):644–654, nov 1976. One citation on page 5.
- [DKS] Luis J. Dominguez Perez, Ezekiel J. Kachisa, and Michael Scott. Implementing cryptographic pairings: a magma tutorial. Cryptography ePrint Archive,

- Report 2009/072. <http://eprint.iacr.org/>. There are 6 citations on pages 2, 26, 49, 146, 157, and 158.
- [DLS81] Peter Downey, Benton Leong, and Ravi Sethi. Computing sequences with addition chains. *SIAM Journal on Computing*, 10(3):638–646, 1981. One citation on page 58.
- [Dom] Luis J. Dominguez Perez. Automatic code generator website. <http://www.computing.dcu.ie/~ldominguez/phdproject.html>. One citation on page 126.
- [Dom09] Luis J. Dominguez Perez. Implementing cryptographic pairings: The online tutorial, 2009. <http://www.computing.dcu.ie/~ldominguez/pairings>. There are 3 citations on pages 86, 101, and 158.
- [DS09] Luis J. Dominguez Perez and Michael Scott. Automatic generation of optimised cryptographic pairing functions. SPEED-CC 2009 Workshop memories, 2009. <http://www.hyperelliptic.org/SPEED/record09.pdf>. There are 5 citations on pages 3, 59, 63, 76, and 147.
- [DSD07] Augusto J. Devegili, Michael Scott, and Ricardo Dahab. Implementing cryptographic pairings over Barreto-Naehrig curves. In Tsuyoshi Takagi, Tatsuaki Okamoto, Eiji Okamoto, and Takeshi Okamoto, editors, *Pairing-Based Cryptography – Pairing 2007*, volume 4575 of *Lecture Notes in Computer Science*, pages 197–207. Springer-Verlag, 2007. Pairing 2007. There are 5 citations on pages 2, 76, 78, 88, and 157.
- [Duh93] Richard Duhrkopf. Tutorial software. *The American Biology Teacher*, 55(2):123–124, 1993. One citation on page 157.
- [eba] Ebats. <http://www.ecrypt.eu.org/ebats/cpucycles.html>. One citation on page 142.

- [ECR] ECRYPT. Ecrypt ii yearly report on algorithms and key sizes (2009-2010). One citation on page 13.
- [Fre06] David Freeman. Constructing pairing-friendly elliptic curves with embedding degree 10. In Edwin Sha, Sung-Kook Han, Cheng-Zhong Xu, Moon Hae Kim, Laurence T. Yang, and Bin Xiao, editors, *Embedded and Ubiquitous Computing*, volume 4096 of *Lecture Notes in Computer Science*, pages 452–465. Springer-Verlag, 2006. In Algorithmic Number Theory Symposium ANTS-VII. There are 2 citations on pages 34 and 89.
- [FST10] David Freeman, Michael Scott, and Edlyn Teske. A taxonomy of pairing-friendly elliptic curves. *Journal of Cryptology*, 23(2):224–280, April 2010. There are 8 citations on pages 3, 10, 33, 116, 120, 124, 141, and 157.
- [Gal05] Steven D. Galbraith. Pairings. In Ian F. Blake, Gadiel Seroussi, and Nigel P. Smart, editors, *Advances in Elliptic Curve Cryptography*, volume 317 of *London Mathematical Society. Lecture Note Series*, pages 183–214. Cambridge University Press, 2005. One citation on page 76.
- [Gar05a] Jesse James Garrett. Ajax: A new approach to web applications, 2005. <http://www.adaptivepath.com/ideas/essays/archives/000385.php>. One citation on page 160.
- [Gar05b] Simon M. Garrett. How do we evaluate Artificial Immune Systems? *Evol. Comput.*, 13(2):145–177, 2005. One citation on page 65.
- [GHO<sup>+</sup>07] Robert Granger, Florian Hess, Roger Oyono, Nicolas Thériault, and Frederik Vercauteren. Ate pairing on hyperelliptic curves. In Moni Naor, editor, *Advances in Cryptology - EUROCRYPT 2007*, volume 4515 of *Lecture Notes in Computer Science*, pages 430–447. Springer-Verlag, 2007. One citation on page 28.

- [GLV00] Robert P. Gallant, Robert J. Lambert, and Scott A. Vanstone. Improving the parallelized Pollard lambda search on anomalous binary curves. *Mathematics of Computation*, 69(232):1699–1705, 2000. One citation on page 152.
- [GLV01] Robert P. Gallant, Robert J. Lambert, and Scott A. Vanstone. Faster point multiplication on elliptic curves with efficient endomorphisms. In Joe Kilian, editor, *Advances in Cryptology – CRYPTO 2001*, volume 2139 of *Lecture Notes in Computer Science*, pages 190–200. Springer-Verlag, 2001. Crypto 2001. One citation on page 46.
- [GMV07] Steven D. Galbraith, James F. McKee, and Paula C. Valença. Ordinary abelian varieties having small embedding degree. *Finite Fields and Their Applications*, 13(4):800–814, 2007. One citation on page 34.
- [GPS06] Robert Granger, Daniel Page, and Nigel P. Smart. High security Pairing-Based Cryptography revisited. In Florian Hess, Sebastian Pauli, and Michael Pohst, editors, *Algorithmic Number Theory – ANTS-VII*, volume 4076 of *Lecture Notes in Computer Science*, pages 480–494. Springer-Verlag, 2006. There are 2 citations on pages 16 and 78.
- [GS08] Steven D. Galbraith and Michael Scott. Exponentiation in pairing-friendly groups using homomorphisms. In Steven D. Galbraith and Kenneth G. Paterson, editors, *Pairing-Based Cryptography – Pairing 2008*, volume 5209 of *Lecture Notes in Computer Science*, pages 211–224. Springer-Verlag, 2008. Pairing 2008. There are 10 citations on pages 2, 46, 47, 48, 49, 50, 96, 147, 157, and 169.
- [HDP05] Lei Hu, Jun-Wu Dong, and Ding-Yi Pei. Implementation of cryptosystems based on tate pairing. *J. Comput. Sci. Technol.*, 20(2):264–269, 2005. There are 2 citations on pages 76 and 77.



- [HEA] HEAnet. Ireland’s national education & research network. <http://www.heanet.ie/>. One citation on page 154.
  
- [Hes08] Florian Hess. Pairing lattices. In Steven D. Galbraith and Kenneth G. Paterson, editors, *Pairing-Based Cryptography – Pairing 2008*, volume 5209 of *Lecture Notes in Computer Science*, pages 18–38, Berlin, Heidelberg, 2008. Springer-Verlag. <http://www.math.tu-berlin.de/~hess/personal/pairing-lattice.pdf>. There are 4 citations on pages 16, 17, 20, and 29.
  
- [HMOV04] Darrel Hankerson, Alfred Menezes, and Scott Vanstone. *Guide to elliptic curve cryptography*. Springer-Verlag, 2004. ISBN 0-387-95273-X. There are 3 citations on pages 6, 7, and 12.
  
- [HSV06] Florian Hess, Nigel P. Smart, and Frederick Vercauteren. The Eta pairing revisited. *IEEE Trans. Information Theory*, 52(10):4595–4602, oct. 2006. There are 6 citations on pages 11, 12, 19, 22, 28, and 94.
  
- [Ica09] Thomas Icart. How to hash into elliptic curves. In Shai Halevi, editor, *Advances in Cryptology – CRYPTO 2009*, volume 5677 of *Lecture Notes in Computer Science*, pages 303–316. Springer-Verlag, 2009. One citation on page 93.
  
- [ICH] ICHEC. Irish Center for High-End Computing. <http://www.ichec.ie/>. One citation on page 154.
  
- [Jou00] Antoine Joux. A one round protocol for tripartite Diffie–Hellman. In Wieb Bosma, editor, *Algorithmic Number Theory – ANTS-IV*, volume 1838 of *Lecture Notes in Computer Science*, pages 385–394. Springer-Verlag, 2000. One citation on page 1.
  
- [Kai80] Thomas Kailath. *Linear Systems*. Prentice-Hall, Inc., 1980. There are 3 citations on pages 36, 50, and 51.

- [KMM04] Tomasz Kaczynski, Konstantin Mischaikow, and Marian Mrozek. *Computational Homology*. Applied Mathematical Sciences, vol. 157. Springer-Verlag New York, Inc., 2004. One citation on page 37.
- [Kob87] Neal Koblitz. Elliptic curve cryptosystems. *Mathematics of Computation*, 48(177):203–209, 1987. One citation on page 5.
- [Kob91] Neal Koblitz. CM-curves with good cryptographic properties. In J. Feigenbaum, editor, *Advances in Cryptology – Crypto 1991*, volume 576 of *Lecture Notes in Computer Science*, pages 279–287. Springer-Verlag, 1991. One citation on page 152.
- [KSS08] Ezekiel Kachisa, Edward F. Schaeffer, and Michael Scott. Constructing Brezing-Weng pairing friendly elliptic curves using elements in the cyclotomic field. In Steven D. Galbraith and Kenneth G. Paterson, editors, *Pairing-Based Cryptography – Pairing 2008*, volume 5209 of *Lecture Notes in Computer Science*, pages 126–135. Springer-Verlag, 2008. Pairing 2008. There are 7 citations on pages 27, 30, 34, 47, 79, 121, and 157.
- [IC] The Magma On line Calculator. <http://magma.maths.usyd.edu.au/calc>. There are 2 citations on pages 86 and 101.
- [Len97] Arjen K. Lenstra. Using cyclotomic polynomials to construct efficient discrete logarithm cryptosystems over finite fields. In Vijay Varadharajan, Josef Pieprzyk, and Yi Mu, editors, *Information Security and Privacy*, Lecture Notes in Computer Science. Springer-Verlag, 1997. Second Australasian Conference, ACISP’ 97. One citation on page 16.
- [LL96] Tsit-Yeun Lam and Ka Hin Leung. On the cyclotomic polynomial  $\phi_{pq}(x)$ . In John H. Ewing, editor, *The American Mathematical Monthly*, volume 103, pages 562–564. Mathematical Association of America, 1996. One citation on page 16.

- [LLP08] Hyang-Sook Lee, Eunjeong Lee, and Cheol-Min Park. Efficient and generalized pairing computation on Abelian varieties. Cryptology ePrint Archive, Report 2008/040, 2008. <http://eprint.iacr.org/>. There are 5 citations on pages 22, 23, 24, 25, and 144.
  
- [LMN10] Kristin Lauter, Peter L. Montgomery, and Michael Naehrig. An analysis of affine coordinates for pairing computation. In Marc Joye, Atsuko Miyaji, and Akira Otsuka, editors, *Pairing-Based Cryptography – Pairing 2010*, volume 6487 of *Lecture Notes in Computer Science*, pages 1–20. Springer-Verlag, 2010. <http://eprint.iacr.org/2010/363>. There are 3 citations on pages 3, 7, and 126.
  
- [Lyn] Ben Lynn. PBC library - the Pairing-Based Cryptography library. <http://crypto.stanford.edu/pbc/>. One citation on page 144.
  
- [Men93] Alfred Menezes. *Elliptic Curve Public Key Cryptosystems*. Kluwer Academic Publishers., 1993. One citation on page 14.
  
- [Mil86] Victor S. Miller. Use of elliptic curves in cryptography. In Hugh C. Williams, editor, *Advances in Cryptology – CRYPTO 1985*, volume 1440 of *Lecture Notes in Computer Science*, pages 417–426. Springer-Verlag, 1986. One citation on page 5.
  
- [MKHO07] Seiichi Matsuda, Naoki Kanayama, Florian Hess, and Eiji Okamoto. Optimised versions of the ate and twisted ate pairings. Cryptology ePrint Archive, Report 2007/013, 2007. <http://eprint.iacr.org/>. One citation on page 28.
  
- [MNT01] Atsuko Miyaji, Masaki Nakabayashi, and Shunzuo Takano. New explicit conditions of elliptic curve traces for FR-reduction. *IEICE Trans. Fundamentals*, E84:1234–1243, 2001. There are 2 citations on pages 33 and 86.

- [MS03] T. Mulders and A. Storjohann. On lattice reduction for polynomial matrices. *J. Symb. Comput.*, 35(4):377–401, 2003. One citation on page 52.
- [Muc97] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufman Publishers, Inc., 1997. There are 2 citations on pages 40 and 107.
- [NAS<sup>+</sup>08] Yasuyuki Nogami, Masataka Akane, Yumi Sakemi, Hidehiro Kato, and Yoshitaka Morikawa. Integer variable  $\chi$ -based ate pairing. In Steven D. Galbraith and Kenneth G. Paterson, editors, *Pairing-Based Cryptography – Pairing 2008*, volume 5209 of *Lecture Notes in Computer Science*, pages 178–191, Berlin, Heidelberg, 2008. Springer-Verlag. There are 2 citations on pages 88 and 150.
- [NIS] NIST. Recommendation for key management part 1: General sp 800-57, may 2006. One citation on page 13.
- [NNS10] Michael Naehrig, Ruben Niederhagen, and Peter Schwabe. New software speed records for cryptographic pairings. In *Progress in Cryptology - LAT-INCRIPT 2010*, volume 6212 of *Lecture Notes in Computer Science*, pages 109–123. Springer-Verlag, 2010. One citation on page 3.
- [NV99] Leandro Nunes de Castro and Fernando Jose Von Zuben. Artificial Immune Systems: Part i - Basic theory and applications. Technical Report TR-DCA 01/99, Department of Computer Engineering and Industrial Automation at the School of Electrical and Computer Engineering, State University of Campinas, Brazil, 1999. There are 2 citations on pages 64 and 65.
- [Oli81] Jorge Olivos. On vectorial addition chains. *Journal of Algorithms*, 2:13–21, 1981. There are 3 citations on pages 63, 76, and 82.
- [Pol78] John M. Pollard. Monte Carlo methods for index computation (mod  $p$ ). *Mathematics of Computation*, 32:918–924, 1978. One citation on page 152.

- [PP82] Thomas Pittman and James Peters. *The art of compiler design: theory and practice*. Prentice-Hall, Inc., 1982. There are 2 citations on pages 40 and 103.
- [PSM06] Lee Pike, Mark Shields, and John Matthews. A verifying core for a cryptographic language compiler. *ACM International Conference Proceeding Series*, 205:1–10, 2006. One citation on page 103.
- [RSA78] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126, 1978. One citation on page 5.
- [SB04] Michael Scott and Paulo S.L.M. Barreto. Compressed pairings. In Matt Franklin, editor, *Advances in Cryptology – CRYPTO 2004*, volume 3152 of *Lecture Notes in Computer Science*, pages 73–82. Springer Berlin / Heidelberg, 2004. 10.1007/978-3-540-28628-8\_9. One citation on page 88.
- [SBC<sup>+</sup>09a] Michael Scott, Naomi Benger, Manuel Charlemagne, Luis J. Dominguez Perez, and Ezekiel J Kachisa. Fast hashing to  $G_2$  on pairing-friendly curves. In Hovav Shacham and Brent Waters, editors, *Pairing-Based Cryptography – Pairing 2009*, volume 5671 of *Lecture Notes in Computer Science*, pages 102–113. Springer-Verlag, 2009. Pairing 2009. There are 8 citations on pages 2, 92, 93, 99, 100, 110, 146, and 169.
- [SBC<sup>+</sup>09b] Michael Scott, Naomi Benger, Manuel Charlemagne, Luis J. Dominguez Perez, and Ezekiel J Kachisa. On the final exponentiation for calculating pairings on ordinary elliptic curves. In Hovav Shacham and Brent Waters, editors, *Pairing-Based Cryptography – Pairing 2009*, volume 5671 of *Lecture Notes in Computer Science*, pages 78–88. Springer-Verlag, 2009. There are 7 citations on pages 2, 63, 72, 76, 146, 157, and 169.

- [Sco] Michael Scott. MIRACL – Multiprecision Integer and Rational Arithmetic C/C++ Library. <http://ftp.computing.dcu.ie/pub/crypto/miracl.zip>. There are 3 citations on pages 38, 44, and 150.
- [Sco02] Mike Scott. Authenticated id-based key exchange and remote log-in with simple token and pin number. Cryptology ePrint Archive, Report 2002/164, 2002. <http://eprint.iacr.org/>. One citation on page 4.
- [Sco07] Michael Scott. Implementing cryptographic pairings. In Tsuyoshi Takagi, Tatsuaki Okamoto, Eiji Okamoto, and Takeshi Okamoto, editors, *Pairing-Based Cryptography – Pairing 2007*, volume 4575 of *Lecture Notes in Computer Science*, pages 177–196. Springer-Verlag, 2007. Invited Talk. <ftp://ftp.computing.dcu.ie/pub/resources/crypto/pairings.pdf>. There are 4 citations on pages 18, 20, 78, and 88.
- [Sil86] Joseph H. Silverman. *The Arithmetic of Elliptic Curves*. Springer-Verlag, 1986. There are 3 citations on pages 14, 16, and 18.
- [SK03] Ryuichi SAKAI and Masao KASAHARA. ID based cryptosystems with pairing on elliptic curve. Cryptology ePrint Archive, Report 2003/054, 2003. <http://eprint.iacr.org/>. One citation on page 92.
- [Smi82] Brian C. Smith. *Procedural reflection in programming languages*. PhD thesis, Massachusetts Institute of Technology. Dept. of Electrical Engineering and Computer Science., 1982. There are 2 citations on pages 45 and 103.
- [Sti95] Douglas R. Stinson. *Cryptography. Theory and Practice*. Discrete Mathematics and Its Applications. CRC Press, 1995. One citation on page 5.
- [Ver10] Frederik Vercauteren. Optimal pairings. *IEEE Transactions on Information Theory*, 56(1):455–461, jan. 2010. <http://www.cosic.esat>.

- [kuleuven.be/publications/article-1039.pdf](http://kuleuven.be/publications/article-1039.pdf). There are 3 citations on pages 29, 30, and 31.
- [VH07] Gottfried Vossen and Stephan Hagemann. *Unleashing Web 2.0: from concepts to creativity*. Morgan Kaufmann, 2007. One citation on page 160.
- [VVV93] Jan Van den Bussche, Dirk Van Gucht, and Gottfried Vossen. Reflective programming in the relational algebra. In *PODS '93: Proceedings of the twelfth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 17–25, New York, NY, USA, 1993. ACM. One citation on page 45.
- [vW99] Paul C. van Oorschot and Michael J. Wiener. Parallel collision search with cryptanalytic applications. *Journal of Cryptology*, 12:1–28, 1999. One citation on page 152.
- [Wel08] Dan Wellman. *Learning the Yahoo! User Interface Library*. Packt Publishing, March 2008. One citation on page 162.
- [Wil02] S. Gill Williamson. *Combinatorics for Computer Science*. Dover Publications, Inc., Dover edition, 2002. edition, 2002. 1985. One citation on page 36.
- [WZ98] Michael J. Wiener and Robert J. Zuccherato. Faster attacks on elliptic curve cryptosystems. In Stafford Tavares and Henk Meijer, editors, *Selected Areas in Cryptography – SAC 1998*, volume 1556 of *Lecture Notes in Computer Science*, pages 190–200. Springer-Verlag, 1998. One citation on page 152.
- [Yah] Yahoo Inc. YUI Library. BSD License. <http://developer.yahoo.com/yui/>. There are 2 citations on pages 163 and 164.

- [ZZH07] Chang-An Zhao, Fangguo Zhang, and Jiwu Huang. A note on the ate pairing. Cryptology ePrint Archive, Report 2007/247, 2007. <http://eprint.iacr.org/>. There are 2 citations on pages 22 and 28.



# SUBJECT INDEX

Addition Chain, 57

Addition Sequence, 59

ate pairing, 19

$\chi$ , 12

Certicom, 151

$\Delta(E)$ , 6

$\delta$ , 12

$\#E$ , 6

$\varphi$ , 15

ECC, 5

ECDLP, 12

Echelon Form, 36

elliptic curve, 5

$E$ , 5

$E'$ , 11

ordinary, 7

supersingular, 7

twist curve, 11

embedding degree, 7

$k$ , 7

extension field, 7

$\mathbb{F}_p$ , 5

$\mathbb{F}_{p^k}$ , 7

Final Exponentiation, 17

Frobenius endomorphism, 14

$j(E)$ , 6

L-value, 41

Line function, 18

Miller algorithm, 16

$\mathcal{O}$ , 6

$\langle P \rangle$ , 6

$\phi$ , 14

$\pi$ , 14

$\psi$ , 14

parameters of the curve

$k$ , 7

$p$ , 5

$r$ , 6

$t$ , 6

point-at-infinity, 6

proto-sequence, 59

Public-Key Cryptosystem, 5

$\rho$ , 33

R-ate pairing, 22

R-value, 41

Reverse Polish notation, [42](#)

Vector Chain, [58](#)

Tate pairing, [17](#)

trace of Frobenius, [6](#)