Speeding Up Elliptic Curve Cryptography by Optimizing Scalar Multiplication in Software Implementations

Gerwin Gsenger

Institute for Applied Information
Processing and Communications (IAIK)
Graz University of Technology
Inffeldgasse 16a
8010 Graz, Austria

Master's Thesis

Supervisors:
Dipl.-Ing. Christan Hanser
Dipl.-Ing. Dr.techn. Michael Hutter

Assessor:
Univ.-Prof. Dipl.-Ing. Dr.techn. Roderick Bloem

October, 2014

Statutory declaration

I declare that I have authored this thesis independently, that I have not used other than the declared sources / resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

Graz, ________________ ________________________
(date) (signature)

Eidesstattliche Erklärung[1]

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Graz, am ______________ ________________________
(Datum) (Unterschrift)

[1]Beschluss der Curricula-Kommission für Bachelor-, Master- und Diplomstudien vom 10.11.2008; Genehmigung des Senates am 1.12.2008

Acknowledgements

First, I want to thank Roderik Bloem for being the assessor of this thesis.
Second, I want to thank my supervisors Christan Hanser and Michael Hutter for providing their knowledge, experience and patience during the writing of this thesis.
Third, I want to thank my parents Anna and Reinhold for their support and confidence in me, not only while working on this thesis but throughout out my entire life.
Forth, I would like to thank Franziska for being my other half and a wonderful part of my life.
Finally I would like to thank Christian for being a trustworthy friend throughout all those years of studying.

Abstract

Elliptic curve cryptography (ECC) is widely deployed in *public-key cryptographic schemes* and offers high security with small key sizes. In most *elliptic curve public-key cryptography* implementations, *scalar multiplication* is the operation with the highest computational effort. This means that it has a significant influence on the execution time of ECC algorithms. Optimizing this operation is therefore vital for good performance and represents an interesting field for new research.

In this thesis, we discuss elliptic curves used for cryptography, including a new model for elliptic curves called *binary Huff* curves. This curve model provides very fast *differential* addition and *differential* doubling formulas. The discussion is completed with a detailed explanation of some known mathematical attacks on elliptic curve cryptography and an introduction to implementation attacks. We present several approaches to scalar multiplication in order to illustrate different possibilities for scalar multiplication method optimizations. This includes high-speed scalar multiplication methods, an endomorphism-based scalar multiplication method and side channel resistant scalar multiplication methods. The discussed high-speed scalar multiplication methods, such as the fixed-base comb method, use intense precomputations to improve their performance. The endomorphism-based scalar multiplication method exploits an efficiently computable endomorphism available on certain curves to speed up the computation. In contrast to the aforementioned multiplication methods, the *Montgomery ladder* and *Joye's Double-and-Add method* primarily focus on side channel resistance. We implemented all mentioned methods and available optimizations in Java. In this context, we introduce our new *differential Montgomery ladder scalar multiplication* implementation which works on *Huff curves*. This implementation is accompanied by efficient all-in-one, back-and-forth conversion formulas with included y-coordinate recovery. Our *differential Montgomery ladder* on *binary Huff* curves is up to 7.4% faster than our implementation of the fastest known *Montgomery ladder* formulas up to that point. Furthermore, we implemented the so-called *improved fixed-base comb method* for scalar multiplication and give a performance comparison for some of our implementations. Our implementation effort is completed with a more general discussion on how cryptography can be implemented in Java.

Keywords: elliptic curves, ECC, Huff curves, scalar multiplication, fixed-base comb methods, Montgomery ladder, Joye's Double-and-Add, CoZ coordinates, MOV, SSSA, Pohlig-Hellman, Pollard's rho

Kurzfassung

Elliptische-Kurven-Kryptografie (ECC) ist in *Public-Key*-Verschlüsselungsverfahren weit verbreitet und bietet hohe Sicherheit trotz kurzer Schlüssellängen. In den meisten Elliptische-Kurven-Kryptografieimplementierungen ist Skalarmultiplikation jene Operation mit dem höchsten Rechenaufwand. Dies bedeutet, dass sie einen signifikanten Einfluss auf die Ausführungszeit von ECC Algorithmen hat. Daher ist eine Optimierung dieser Operation wichtig für gute Performanz und stellen ein interessantes Feld für neue Forschung dar.

In dieser Masterarbeit diskutieren wir elliptische Kurven die in der Kryptografie eingesetzt werden, inklusive einem neuen Model für elliptische Kurven, sogenannte *binäre Huff*-Kurven. Dieses Kurvenmodel bietet sehr schnelle *differenzielle* Additions- und *differenzielle* Verdoppelungsformeln. Die Diskussion wird durch eine detaillierte Erklärung einiger bekannter, mathematischer Attacken auf Elliptische-Kurven-Kryptografie und eine Einführung in Implementierungsattacken abgerundet. Wir stellen mehrere Zugänge zur Skalarmultiplikation vor, um die verschiedenen Möglichkeiten für Skalarmultiplikationsoptimierungen aufzuzeigen. Dies inkludiert Hochgeschwindigkeitsskalarmultiplikationsmethoden, eine Skalarmultiplikationsmethode basierend auf Endomorphismen und seitenkanalattackenresistente Skalarmultiplikationsmethoden. Die diskutierten Hochgeschwindigkeitsskalarmultiplikationsmethoden, wie die fixed-base comb Methode, nutzen aufwendige Vorberechnungen um ihre Performanz zu verbessern. Die endomorphismusbasierte Skalarmultiplikationsmethode nützt einen effizient berechenbaren Endormorphismus auf einigen Kurven aus, um die Berechnung zu beschleunigen. Im Gegensatz zu den zuvor erwähnten Skalarmultiplikationsmethoden, ist der Fokus der *Montgomery Leiter* und der *Joye's Double-and-Add Methode* die Seitenkanalattackenresistenz. Wir haben alle erwähnten Methoden und die zur Verfügung stehenden Optimierungen in Java implementiert. In diesem Kontext stellen wir auch unsere neue, *differenzielle Montgomery Leiter Skalarmultiplikationsmethodenimplementierung*, welche auf *Huff Kurven* arbeitet, vor. Diese Implementierung wird begleitet von effizienten, all-in-one, vor-und-zurück Konvertierungsformeln mit integrierter y-Koordinatenrekonstruktion. Unsere *differenzielle Montgomery Leiter* auf *binären Huff* Kurven ist bis zu 7.4% schneller als unsere Implementierung der, bis zu diesem Zeitpunkt, schnellsten bekannten *Montgomery Leiter* Formeln. Des Weiteren haben wir die sogenannte *verbesserte fixed-base comb Methode* für Skalarmultiplikation implementiert und geben einen Performanzvergleich für einige unserer Implementierungen. Unser Implementierungsaufwand wird durch eine allgemeine Diskussion darüber, wie Kryptografie in Java implementiert werden kann, abgerundet.

Stichwörter: elliptische Kurven, ECC, Huff Kurven, Skalarmultiplikation, fixed-base comb Methoden, Montgomery Leiter, Joye's Double-and-Add, CoZ Koordinaten, MOV, SSSA, Pohlig-Hellman, Pollard's rho

Contents

List of Algorithms

List of Figures

List of Tables

Chapter 1

Introduction

One of the key factors changing our society is the ubiquitous electronic transfer of information. As the transferred information may be of sensitive nature, cryptography is used to protect said information. Cryptography offers different "services" to it's users. Typically one or several properties like *confidentiality, integrity, authenticity, non-repudiation, anonymity* or *privacy* of data are ensured. To be able to cover such a wide range of goals, cryptography is a diverse field of science, although an abstract hierarchy within cryptography exists. *Cryptographic protocols* consist of *cryptographic schemes* and *cryptographic algorithms*, which themselves are founded on *cryptographic primitives* firmly rooted in mathematics. This thesis will focus on topics concerning *cryptographic algorithms*, *cryptographic primitives* and also some aspects of the implementation of those theoretic concepts.

There are two main categories of cryptographic schemes, *symmetric cryptography*, where both parties share a secret key which is used for encryption and decryption, and *asymmetric cryptography*, also called *public-key cryptography*. *Public-key cryptography* denotes cryptographic systems which make use of two keys. One is called a *private-key*, which is used for decryption and has to be kept secret. This key is accompanied by, and mathematically related to, a so-called *public-key*, which is used for encryption. The *public-key* can be unveiled to the public without compromising the security of the scheme, as it must be mathematically hard to infer the *private-key* if one knows the *public-key*. Together, *public-* and *private-key* form a so-called *keypair*.

In the 1970s, two major concepts in *public-key cryptography* were published; firstly, in 1976, the authors of [DH76] introduced a concept later named *Diffie-Hellman key exchange*. This key exchange relies only on the authenticity of the exchanged keys, to establish a shared secret known to both participants of the key exchange. The introduction of the *Diffie-Hellman key exchange* was followed in 1978 by the publication of [RSA78], where the authors introduced the so-called *RSA public-key cryptosystem*. The RSA scheme can be used to encrypt and to digitally sign data. It builds its security on the so-called the *RSA problem*. Currently, the most effective way to solve the *RSA problem* is to factor the RSA modulus which is a big integer consisting of two large primes. For factoring big integers, a subexponential time algorithm exists, namely the *general number field sieve* (GNFS) (more information on the GNFS is given in [LHWL93]). Cryptography which relies on the assumption that factoring big integers is hard is called *integer factorisation cryptography* (IFC).

The mathematical concept of elliptic curves has been known for a long time, and was mostly a tool used in theoretical mathematics. In 1985, elliptic curves became more of practical use when Neil Koblitz in [Kob87] and Victor S. Miller in [Mil86] introduced *elliptic curve cryptography* (ECC), which in turn inspired a lot of cryptography related research on this topic. The ECC cryptographic schemes build their security on the assumption that it is hard to solve the *elliptic curve discrete logarithm problem* (explained in more detail in Section 3.4.1). To this day, there is no publicly known algorithm that solves the *elliptic curve discrete logarithm problem* in polynomial time on a *classical* computer if the curve is chosen properly.

ECC offers the same level of security as competing cryptographic schemes (e.g., RSA) with considerably lower key sizes, which makes it especially interesting for systems with restricted computing power and/or memory. A detailed comparison of the estimated security gained by different key sizes in various cryptographic systems is available in [LV01]. NIST gives in [BBB$^+$12] ready to use numbers as given in Table 1.1 (taken from [BBB$^+$12, Table 2]). Here one can clearly see how much slower the suggested key sizes for ECC based schemes, compared to IFC based schemes, grow.

Table 1.1: Comparison of key size and achieved bits of security, for *Diffie-Hellman* (D-H), *integer factorisation cryptography* (IFC) and *elliptic curve cryptography* (ECC)

Bits of Security	**FCC (e.g., DSA, D-H) key size in bits public / private**	**IFC (e.g., RSA) key size in bits**	**ECC (e.g., ECDSA) key size in bits**
80	1024 / 160	1024	160
112	2048 / 224	2048	224
128	3072 / 256	3072	256
192	7680 / 384	7680	384
256	15360 / 512	15360	512

In the past, trust in the ECC has been shattered several times. The first time it concerned the mathematical side of ECC. In 1991 the so-called *MOV attack* came out, which attacked ECC on so called *supersingular* elliptic curves, closely followed by the *SSSA attack* which attacked curves, where the curve order is equal to the order of the underlying finite field (both attacks are explained in Section 3.4.3). Quite recently, trust in the standardizing body surrounding ECC was shaken. It was shown that the NIST standardised cryptographically secure pseudorandom number generator `Dual_EC_DRBG` had been standardized in June 2006, although the authors of [Bro06] and [SS06] had shown in March and May 2006 respectively that it had weaknesses that could result in a backdoor. This raised questions concerning the legitimacy of the NIST standardized curve parameters.

1.1 Contribution

In most *elliptic curve public-key cryptography* implementations *scalar multiplication* is the operation with the highest computational effort. This means that it has a significant influence on the execution time of ECC algorithms. Therefore, optimizing this operation is vital for good performance and represents an interesting field for new research.

We implemented two different general approaches to scalar multiplication. First, we implemented several different highly regular scalar multiplication methods, with very small memory footprint and acceptable performance. Those are important building blocks for side channel resistant implementations. Second, we implemented different high performance scalar multiplication algorithms which use precomputations and give excellent performance in cases where no side channel resistance is needed. All implementations were benchmarked and evaluated.

To our knowledge, we were the first to implement so-called *Huff curves* and publicly state the curve parameters of the *Huff curves* corresponding to NIST curves to reduce the implementation effort for others interested in this curve type. Furthermore, we introduced all-in-one, back-and-forth conversion formulas with included y-coordinate recovery for *differential Huff curve Montgomery ladders*. This was the theoretic basis for a *Montgomery ladder* implementation, which is up to 7.4% faster than our implementation of the, up to that point, fastest known *Montgomery ladder* formulas. We also published our research in a scientific paper, [GH13], which was presented in a workshop during the ARES 2013 conference.

1.2 Outline

The reminder of this thesis is organized as follows. We start with Chapter 2 by building the necessary theoretical background, including groups, fields and elliptic curves, which is needed to understand the work documented in this thesis. In Chapter 3, various flavors of elliptic curves, including *Huff curves*, are explained. Furthermore, we give generic and state-of-the-art attacks on ECC. Later on, in Chapter 4, scalar multiplication methods are introduced and explained. We discuss high performance scalar multiplication methods which use *precomputations* as a time-memory trade-off to speed up multiplications. Those are in contrast to *Montgomery ladder* type multiplication methods which have a very small memory footprint and are highly regular. Chapter 5 gives an insight into how cryptography is typically implemented in Java™. This chapter is followed by Chapter 6, where the results of the practical implementations are given and discussed. The thesis concludes with Chapter 7.

Chapter 2

Preliminaries

In this chapter, we try to give all readers a brief introduction to the mathematical principles necessary to understand elliptic curves and the work done in this thesis. This chapter is based on information published in the following publications: [HMV04], [CF05], [Sil92], [Sma02], [BSS05] and [Han10] as well as lecture notes from [LZM09] and [OL09].

The chapter is structured as follows. In Section 2.1, an introduction to groups and fields is given. Both types of finite fields, prime and binary fields, which act as mathematical foundations for elliptic curves, are explained. In Section 2.2, elliptic curves and important structural concepts, for example the group law, are introduced. In Section 2.2.1, the theoretical concept of *projective* coordinate systems is given. This chapter concludes with a short summary in Section 2.3.

2.1 Elementary Algebraic Structures

In this section, we introduce all, necessary elementary algebraic structures.

Section 2.1.1 focuses on groups, followed by Section 2.1.2 which introduces fields in general, followed by Section 2.1.4 which concentrates on finite fields. Furthermore, we also state important rules and theorems necessary for working with those algebraic structures.

2.1.1 Groups

A *group* $(G, \cdot)$, is a set G with a mapping $\cdot : G \times G \to G$ such that:

1. G is *closed* with respect to $\cdot$, meaning that if $a, b \in G$ then $a \cdot b \in G$,
2. $\cdot$ is *associative* that is for $a, b, c \in G,\ a \cdot (b \cdot c) = (a \cdot b) \cdot c$ holds,
3. $(G, \cdot)$ has a *unit (or neutral) element* $e \in G$ such that $a \cdot e = e \cdot a = a$ for all $a \in G$,
4. every element $g \in G$ has an *inverse* g^{-1}, i.e., $g^{-1} \cdot g = g \cdot g^{-1} = e$.

A group G is called *commutative* or *Abelian* if for every two elements $a, b \in G$ the property $a \cdot b = b \cdot a$ is satisfied. A group is called *cyclic* if every element in the group can be expressed as a power g^n with $n \in \mathbb{Z}$ of some element $g \in G$, which is called the *generator* of the group. A *finite* group G has a *finite* number of elements, called the *order* of G and is denoted by $|G|$. If G is *infinite* then the order of the group is ∞. The order of an element $g \in G$ is the smallest number $k \in \mathbb{N}$ for which $g^k = 1$, denoted by $|g| = k$. Furthermore, the order of every element $g \in G$ divides the group order. Again, if no such number exists, the *order of g in G* is ∞. An example to illustrate the workings of a generator in a multiplicative cyclic group can be found below.

Example 2.1.1 (*Generator*)**.** $\mathbb{Z}_3^* = \mathbb{Z}_3 \setminus \{0\} = \{1, 2\}$ where $\mathbb{Z}_3$ are integers mod 3, with generator $g = 2$ the group elements are expressed as $2^1 \equiv 2 \bmod 3,\ 2^2 \equiv 1 \bmod 3$.

A *subgroup* $(H, \cdot)$ of group $(G, \cdot)$ (denoted by $H \leq G$) is a non-empty subset of elements in G, which itself is a group. Furthermore, $|H|$ is a divisor of $|G|$ and the *unit* $e_G = e_H$ stays the same.

Definition 2.1.1 (*Discrete Logarithm*)**.** Let G be a finite group, then the *discrete logarithm* of $h \in G$ to the base $g \in G$ denoted by $log_g(h)$ is a solution x to $g^x = h$, where x is *unique* modulo the group order $|G|$.

Here is a simple example of a discrete logarithm and the order of an element.

Example 2.1.2 (*Discrete Logarithm, Order of Element*)**.** We use the multiplicative, *Abelian* group, given by the integers mod 5, as $\mathbb{Z}_5^* = \mathbb{Z}_5 \setminus \{0\}, = \{1, 2, 3, 4\}$ with generator $g = 2$:

$2^x \equiv 3 \bmod 5,$ a discrete logarithm solution is $x = 3$,

$2^4 = 16 \equiv 1 \bmod 5$ is the smallest positive exponent, therefore 4 is the order of $2 \in \mathbb{Z}_5$.

We will elaborate further on the discrete logarithm and the so-called *discrete logarithm problem* in Chapter 3.

2.1.2 Fields

A *field* $(F, +, \cdot)$ is a triplet where F is a set and $+ : F \times F \to F$ as well as $\cdot : F \times F \to F$ are two mappings, such that:

- $(F, +)$ is an *additive Abelian* group, with *additive unit* 0.
- $(F^*, \cdot) = (F \setminus \{0\}, \cdot)$ is a *multiplicative Abelian* group, with *multiplicative unit* 1.

Both mappings $+$ and $\cdot$ satisfy and are connected through the *distributive law*, meaning that:

$$\forall a\ \forall b\ \forall c \in F : \ (a + b) \cdot c = a \cdot c + b \cdot c, \ \textit{(right-distributivity)}, \text{ and}$$
$$\forall a\ \forall b\ \forall c \in F : \ a \cdot (b + c) = a \cdot b + a \cdot c, \ \textit{(left-distributivity)}.$$

If the set F is *finite*, the field $(F, +, \cdot)$ is called *finite field.*

2.1.3 Homomorphisms

A *group homomorphism* $h : (G, \cdot) \to (G', *)$ is a mapping between two groups $(G, \cdot)$ and $(G', *)$. Group homomorphisms preserve the group structure with respect to the group operations. More precisely, this means that for all $a, b \in G : h(a \cdot b) = h(a) * h(b)$. Also, the inverse $h(a^{-1}) = h(a)^{-1}$ of all elements $a \neq 0$ and the identity element are preserved, i.e., $h(e_G) = e_{G'}$.

A *field homomorphism* $f : (F, +, \cdot) \to (F', \oplus, \odot)$ is a function which represents a group homomorphism for both $(F, +) \to (F', \oplus)$ and $(F, \cdot) \to (F', \odot)$.

An *injective homomorphism* is called *monomorphism*, similarly a *surjective homomorphism* is called *epimorphism* and a *bijective homomorphism* is called *isomorphism.* If two groups G, G' or two fields F, F' are *isomorphic*, meaning that an *isomorphism* between them exists, this is denoted by $G \simeq G'$ respectively $F \simeq F'$.

2.1.4 Finite Fields

A *finite* field, denoted by $\mathbb{F}_q$, has q elements, where $q = p^n$ for some prime p with $n \in \mathbb{N}$. More precisely, for every such prime power p^n, there is, up to isomorphism, exactly one *finite* field $\mathbb{F}_{p^n}$, which is summarized in Theorem 2.1.1.

Theorem 2.1.1 (*Number of Finite Fields*)**.** For every *prime power* $q = p^n$ with $p \in \mathbb{P}, n \in \mathbb{N}$ there exists, up to isomorphisms, exactly one finite field of prime power order q (denoted by $\mathbb{F}_q$).

Proposition 2.1.1 (*Cyclic Multiplicative Group*)**.** The *multiplicative* group $(\mathbb{F}_q^*, \cdot)$ of a *finite* field $(\mathbb{F}_q, +, \cdot)$ is cyclic.

Theorem 2.1.2 (*Fermat's Little Theorem*)**.** States the fact that $a^p \equiv a \pmod{p}$ for all $a \in \mathbb{Z}_p$ where $\mathbb{Z}_p = \{a\ mod\ p : a \in \mathbb{Z}\}$ and $p \in \mathbb{P}$.

Definition 2.1.2 (*Euler's Totient Function*)**.** Let φ denote *Euler's totient function.* Then, given a positive integer n, $\varphi(n)$ gives the number of all integers $1 \leq k \leq n$, where k is *coprime* to n.

Theorem 2.1.3 (*Euler's Theorem*)**.** States the fact that, given two positive, *coprime* integers a and n, $a^{\varphi(n)} \equiv 1 \pmod{n}$. This translates directly to *finite* fields where $a^{\varphi(q)} \equiv 1 \pmod{q}$ for all $a \in \mathbb{F}_q^*$.

The *characteristic* of a field F (denoted by $char(F)$) is defined as the smallest $n \in \mathbb{N}$ such that:

$$\forall a \in F : n \cdot a = 0.$$

If such an integer does not exist, the *characteristic* is 0 by definition. The *characteristic* of a *finite field* is always prime. Given an arbitrary finite field F, the relation between the order $|F|$ and the *characteristic* of the field is given as:

$$|F| = char(F)^n,$$

for some $n \in \mathbb{N}$. If $n = 1$ the field is called *prime field*, otherwise it is called *extension field.*

Theorem 2.1.4 (*Freshman's Dream*)**.** In a field $\mathbb{F}_p$ of characteristic p, we have $(x+y)^p = x^p + y^p$.

Proof. This can easily be seen, as:

$$(x+y)^p = \sum_{n=0}^{p} \begin{pmatrix} p \\ n \end{pmatrix} x^n y^{p-n} = \sum_{n=0}^{p} \frac{p!}{n!(p-n)!} x^n y^{p-n} = x^0 y^p + x^p y^0 = x^p + y^p.$$

Taking a closer look at $\begin{pmatrix} p \\ n \end{pmatrix} = \frac{p!}{n!(p-n)!}$ which is $0 \pmod{p}$ if $n \neq \{0, p\}$ (as $\begin{pmatrix} p \\ 0 \end{pmatrix} = \begin{pmatrix} p \\ p \end{pmatrix} = 1$), leaving only the case $x^0 y^p + x^p y^0 = x^p + y^p$. □

All fields $\mathbb{F}_p$ of prime characteristic are equal to $\mathbb{Z}_p$. Addition and multiplication are calculated modulo the prime p. Subsequently, only *binary fields* respectively the optimized arithmetic over *binary fields* is mentioned in more detail.

Binary Fields

The information in this section is based on information in [BSS99]. A *finite* field $\mathbb{F}_q$ with $q = 2^m$ is called a *binary field.* All *elements* of a *binary field* can be represented with *polynomials.* A binary polynomial $f \in \mathbb{F}_2[x]$ of degree m is given as

$$f(x) = \sum_{i=0}^{m} a_i x^i$$

with *coefficients* $a_i \in \mathbb{F}_2$. The *degree* of a polynomial, denoted $deg(f)$, is defined as the greatest index i for which a_i is not zero, if there is no such i, i.e., $f = 0$, then we define $deg(f) = -\infty$.

Definition 2.1.3 (*Irreducible Polynomial*)**.** A polynomial $f \in \mathbb{F}_2[x]$ is called *irreducible* if f is not invertible in $\mathbb{F}_2[x]$ and its only factors are f and 1.

Definition 2.1.4 (*Monic Polynomial*)**.** A polynomial is called *monic* if the non-zero coefficient of highest degree is 1.

If $f(x)$ is *monic* and *irreducible* in $\mathbb{F}_2[x]$ and of degree m, then $(\mathbb{F}_2[x]/(f(x)), +, \cdot)$ is a *finite* field of order 2^m. A field $\mathbb{F}_{2^m}$ is a vector space of dimension m over $\mathbb{F}_2$, therefore the elements of a binary field are represented as *binary vectors of coefficients* of degree m. The vector is given relative to a *basis.* There are two bases with which binary fields are commonly represented.

- *Polynomial* Basis: The *polynomial* basis is given as $(1, \alpha, \alpha^2, \ldots, \alpha^{m-1})$, where $\alpha \in \mathbb{F}_{2^m}$ is a root of $f(x)$.
- *Normal* Basis: The *normal* basis is given as $(\alpha, \alpha^2, \alpha^{2^2} \ldots, \alpha^{2^{m-1}})$ with $\alpha \in \mathbb{F}_{2^m}$ being a root of $f(x)$.

Example 2.1.3 (*Binary Field*)**.** Given the field $\mathbb{F}_{2^3} = \mathbb{F}_2[x]/(f)$ with irreducible polynomial $f(x) = x^3 + x + 1$, $\alpha \in \mathbb{F}_{2^3}$.

The elements of $\mathbb{F}_{2^3} = \mathbb{F}_2[x]/(f)$ with irreducible polynomial $f(x) = x^3 + x + 1$ are: $0,\ 1,\ \alpha,\ \alpha^2,\ \alpha^3 = \alpha + 1,\ \alpha^4 = \alpha^2 + \alpha,\ \alpha^5 = \alpha^2 + \alpha + 1,\ \alpha^6 = \alpha^2 + 1,\ \alpha^7 = 1$, where $\alpha \in \mathbb{F}_{2^3}$ is a root of $f(x)$.

Arithmetic

In $\mathbb{F}_{2^m}$, calculations are done modulo an *irreducible, binary* polynomial $f(x)$ of degree m. Therefore, an element in $\mathbb{F}_{2^m}$ is always represented via a *binary polynomial* of *degree* smaller then m. Computing squares in *binary* fields is greatly simplified due to Theorem 2.1.4. There are no mixed terms, as $(a + b)^2 = a^2 + b^2$. After a multiplication or a squaring the result may have a *degree* $\geq m$; if so it is reduced modulo the reduction polynomial $f(x)$. Consequently $f(x)$ should be of *low weight* to give a better performance. Polynomial addition modulo 2 is commonly implemented as a series of *exclusive or* operations, as an addition of two bits modulo 2 is equivalent to the logic *exclusive or* operation.

With *polynomial basis* representation, multiplications are more efficient than in *normal basis*, as the polynomial multiplication is a carry-free version of a m-bit integer multiplication. A squaring operation can be realized by inserting 0-bits between the consecutive bits of the internal representation of the polynomial, followed by a reduction modulo the *reduction polynomial.* The 0-bit insertion can be precomputed for all 256 byte values, making it a linear time table lookup. Therefore, the reduction is the most time-consuming part of the squaring operation. With *normal basis* representation, so-called *bit-serialized multipliers*, which are of interest in hardware implementations, can be realized. Also field squarings are very efficient, as they can be implemented as cyclic shifts.

Example 2.1.4 (*Binary Field Multiplication*)**.** Multiplication on the field $\mathbb{F}_{2^3}$ with irreducible polynomial $f(\alpha) = \alpha^3 + \alpha + 1,\ \alpha \in \mathbb{F}_{2^3}$ with $f(\alpha) = 0$:

$$\begin{gathered}(\alpha^2 + \alpha + 1) \cdot (\alpha^2 + \alpha + 1) \bmod (\alpha^3 + \alpha + 1) = \\ (\alpha^4 + \alpha^3 + \alpha^2) + (\alpha^3 + \alpha^2 + \alpha) + (\alpha^2 + \alpha + 1) \bmod (\alpha^3 + \alpha + 1) = \\ (\text{recall } 2 \equiv 0 \bmod 2) \text{ therefore } (\alpha^4 + \alpha^2 + 1) \bmod (\alpha^3 + \alpha + 1) = -\alpha + 1 = \alpha + 1.\end{gathered}$$

2.2 Elliptic Curves

An *elliptic curve* E is a *smooth, algebraic curve* over some field F. The curve is given by the set of points (x, y) satisfying the equation:

$$E : y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6, \tag{2.1}$$

with $a_1, a_2, a_3, a_4, a_6 \in F$, plus the point at infinity $\mathcal{O}$, are called the *F-rational* points and denoted by $E(F)$. $\mathcal{O}$ acts as *unit* to form an *Abelian* group. *Smooth* means there are no singular points, i.e., the *tangent* through every point on E is uniquely determined, which can be ensured by checking that the *discriminant* $\Delta(E)$ of E is $\Delta(E) \neq 0$. For a curve, given by Equation (2.1), $\Delta(E)$ is defined in the following equation, given in [BSS05]:

$$\left.\begin{aligned} b_2 &= a_1^2 + 4a_2, \\ b_4 &= 2a_4 + a_1a_3, \\ b_6 &= a_3^2 + 4a_6, \\ b_8 &= a_1^2a_6 + 4a_2a_6 - a_1a_3a_4 + a_2a_3^2 - a_4^2, \text{ and} \\ \Delta(E) &= -b_2^2b_8 - 8b_4^3 - 27b_6^2 + 9b_2b_4b_6. \end{aligned}\right\}$$

For a field of characteristic 2, Equation (2.1) can be simplified through an *isomorphic* change of variables. There are two cases concerning the change of variables:

$$\left.\begin{aligned} x &= x' + a_2, \\ y &= y' \end{aligned}\right\} \text{for } a_1 = 0, \text{ and} \qquad \left.\begin{aligned} x &= a_1^2x' + \frac{a_3}{a_1}, \\ y &= a_1^3y' + \frac{a_1^2a_4 + a_3^2}{a_1^3} \end{aligned}\right\} \text{for } a_1 \neq 0.$$

This yields the following Equation (2.2), for *non-supersingular* curves in *short Weierstrass form*:

$$\begin{aligned} E : y^2 + xy = x^3 + a_2x^2 + a_6 \ \text{ for } a_1 \neq 0, \text{ with } a_2, a_6 \in F, \text{ and} \\ \Delta(E) = a_6, \end{aligned} \tag{2.2}$$

and Equation (2.3), for so-called *supersingular* curves.

$$\begin{aligned} E : y^2 + a_3y = x^3 + a_4x + a_6. \ \text{ for } a_1 = 0, \text{ with } a_3, a_4, a_6 \in F, \text{ and} \\ \Delta(E) = a_3^4. \end{aligned} \tag{2.3}$$

For a field F of *prime* characteristic, i.e., *char*$(F) \neq \{2,3\}$, the *isomorphic* change of variables:

$$x = x' - \frac{a_1^2 + 4a_2}{12},$$
$$y = y' - \frac{a_1}{2}(x' - \frac{a_1^2 + 4a_2}{12}) - \frac{a_3}{2},$$

yields the following equation in in *short Weierstrass form*:

$$\begin{aligned} &E : y^2 = x^3 + a_4x + a_6, \quad \text{with } a_4, a_6 \in F, \text{ and} \\ &\Delta(E) = -16(4a_4^3 + 27a_6^2). \end{aligned} \tag{2.4}$$

Group Law and Addition Formulas

On elliptic curves there is an addition law, called the *chord-and-tangent* method.

Proposition 2.2.1. If a line *(chord)* is drawn through two distinct points $P_1, P_2 \in E(F)$, $P_1 \neq P_2$ the line intersects E in a third point $P_3 \in E(F)$. If $P_1 = P_2$ the point P_3 is the intersection of E and the *tangent* through P_1 (P_1 is counted twice).

Definition 2.2.1 (*Point Reflection*)**.** A point $P(x, y) \in E(F) \setminus \{\mathcal{O}\}$ is *reflected* across the x-axis by calculating the point $-P(x, -y - a_1x - a_3)$.

In order to add two points $P_1, P_2 \in E(F)$, $P_1 \neq P_2$ one draws a line (*chord*) through P_1 and P_2. The line is going to intersect E at a third point $P_3 \in E(F)$ (taking multiplicities into account). This third point is then reflected across the x-axis, giving the result $-P_3 = P_1 + P_2$. The concept is illustrated in Figure 2.1.

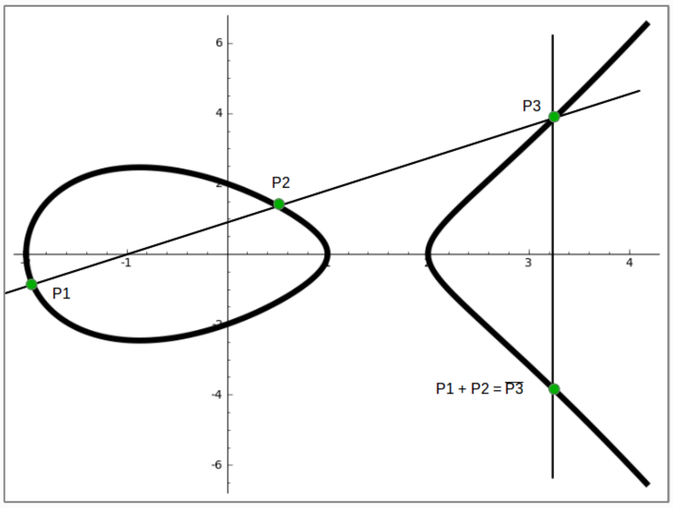

Figure 2.1: Point addition on $E : y^2 = x^3 - x^2 - 4x + 4$ over $\mathbb{R}$

If $P_1 = P_2$, P_1 must be doubled, for which the *tangent* rule applies. The tangent intersects E in another point P_3. This point, is again reflected across the x-axis, giving the result $-P_3 = 2P_1$. This procedure is illustrated in Figure 2.2.

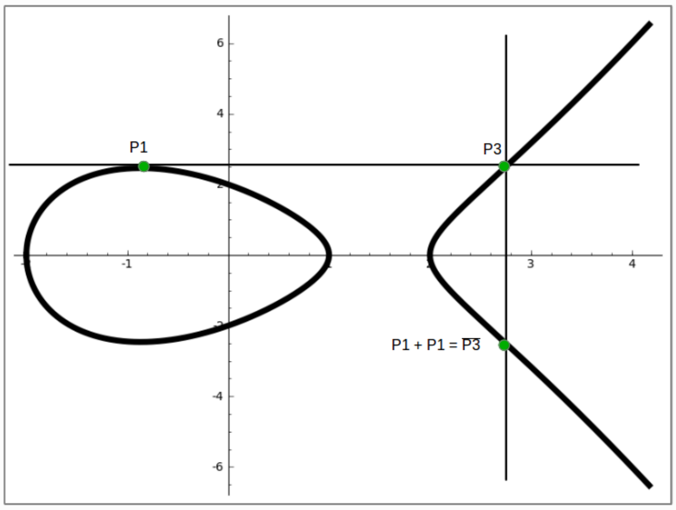

Figure 2.2: Point doubling on $E : y^2 = x^3 - x^2 - 4x + 4$ over $\mathbb{R}$

Group Law: The outlined addition law, in combination with the corresponding set of points, given by $E(F)$, forms an *Abelian* group. Every one of these groups $(E(F), +)$ has a mapping $+ : E(F) \times E(F) \to E(F)$ such that:

1. $E(F)$ is *closed* with respect to $+$, this follows directly from Proposition 2.2.1,
2. $+$ is *associative*: $P_1 + (P_2 + P_3) = (P_1 + P_2) + P_3$ holds for all $P_1, P_2, P_3 \in E(F)$,
3. the *unit element* $e \in E(F)$ is $\mathcal{O}$ such that $P + \mathcal{O} = \mathcal{O} + P = P$ for all $P \in E(F)$,
4. every element $P \in E(F)$ has an *inverse* $-P$ (as stated in Proposition 2.2.1),
5. $+$ is *commutative*, meaning that $P_1 + P_2 = P_2 + P_1$ for all $P_1, P_2 \in E(F)$.

Addition Formulas: The formula for point *addition* with *affine* points, on *non-supersingular, binary curves*, given by Equation (2.2) (taken from [HMV04, p.81]), is:

$$\text{Addition: } x_3 = \lambda^2 + \lambda + x_1 + x_2 + a_2 \text{ and } y_3 = \lambda(x_1 + x_3) + x_3 + y_1 \text{ with } \lambda = \frac{y_1 + y_2}{x_1 + x_2}.$$

$$\text{Doubling: } x_3 = \lambda^2 + \lambda + a_2 = x_1^2 + \frac{b}{x_1^2} \text{ and } y_3 = x_1^2 + \lambda x_3 + x_3 \text{ with } \lambda = \frac{x_1 + y_1}{x_1}.$$

Point *addition* with *affine* points, on *prime curves* given by Equation (2.4), is achieved with these formulas (taken from [HMV04, p.80]):

$$\text{Addition: } x_3 = \left(\frac{y_2 - y_1}{x_2 - x_1}\right)^2 - x_1 - x_2 \text{ and } y_3 = \left(\frac{y_2 - y_1}{x_2 - x_1}\right)(x_1 - x_3) - y_1.$$

$$\text{Doubling: } x_3 = \left(\frac{3x_1^2 + a_4}{2y_1}\right)^2 - 2x_1 \text{ and } y_3 = \left(\frac{3x_1^2 + a_4}{2y_1}\right)(x_1 - x_3) - y_1.$$

The point at infinity $\mathcal{O}$ serves as point of intersection, if for example a point of order 2 is doubled and the tangent does not intersect the elliptic curve. This is illustrated in Figure 2.3.

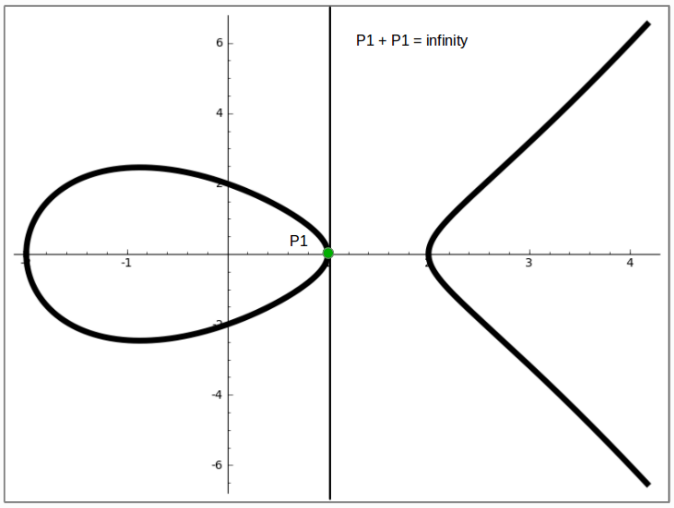

Figure 2.3: Necessity for the point at infinity $\mathcal{O}$, on $E : y^2 = x^3 - x^2 - 4x + 4$ over $\mathbb{R}$

Birational Equivalences

A *birational equivalence* works similar to an *isomorphism*, although it is slightly less strong because it is undefined over some, so-called *exceptional points*.

A pair of elliptic curves (E, E') is *birationally equivalent* if there is a *birational, bijective* map $\Phi : E \to E'$. A *birational* map is a combination of two rational functions such that each works on elements in one of the groups given by the elliptic curve $E(F)$ and $E'(F)$, respectively.
The function $\phi = (\phi_x(x), \phi_y(y))$ with $\phi_x(x), \phi_y(y) \in E(F)$ is mapping

$$\phi : E \to E'$$

and the function $\psi = (\psi_{x'}(x'), \psi_{y'}(y'))$ with $\psi_{x'}(x'), \psi_{y'}(y') \in E'(F)$ ψ is mapping

$$\psi : E' \to E.$$

ϕ and ψ are the inverse of each other and the *identity elements* are $id_E = \psi \circ \phi$ and $id_{E'} = \phi \circ \psi$.

2.2.1 Projective Coordinate Systems

On elliptic curves, the formulas for adding and doubling points with *affine* coordinates need at least one inversion, which is the most expensive operation in *finite* fields. A way to mitigate these costs are *projective* coordinate systems, as these coordinate systems lead to addition and doubling formulas that avoid costly inversions. The following is a brief outline on the theoretical background of *projective* coordinates.

First the equivalence relation $\sim$, over a field F, is defined as follows:

$$(X_1, Y_1, Z_1) \sim (X_2, Y_2, Z_2) \text{ if } X_1 = \lambda X_2, Y_1 = \lambda Y_2, Z_1 = \lambda Z_2 \text{ for some } \lambda \in F^*.$$

$\sim$ is working on the set $F^3 \backslash \{(0, 0, 0)\}$ of non-zero triples over F. A *projective* point represents an equivalence class for *representations* and is given by the following equation:

$$(X : Y : Z) = \{(\lambda X, \lambda Y, \lambda Z) : \lambda \in F^*\}.$$

A *projective* point can be represented by every element in its equivalence class. If $Z \neq 0$, the $1-1$ relation of *affine* and *projective representations* of points can easily be shown by scaling the point. This is done by calculating $(\frac{X}{Z} : \frac{Y}{Z} : 1)$, which is the only *projective* representation where $Z = 1$. The *line at infinity* are those projective points that do not correspond to any affine points, namely those points where $Z = 0$. This set of points is given as $P(F)^0 = \{(X : Y : Z),\ X, Y, Z \in F,\ Z = 0\}$.

For *standard projective coordinates* the homogenized, projective, long Weierstrass equation is given as:

$$E : Y^2Z + a_1XYZ + a_3YZ^2 = X^3 + a_2X^2Z + a_4XZ^2 + a_6Z^3.$$

This equation is derived from the *affine* Weierstrass equation, given in Equation (2.1), by substituting (x, y) with $(\frac{X}{Z}, \frac{Y}{Z})$. For *standard projective coordinates* the *point at infinity* $\mathcal{O}$ is $(0 : 1 : 0)$, and the *negation* of a point P is given as $-P = (X : -Y : Z)$.

Using *projective* coordinates does have its cost, the projective formulas need more multiplications as the additional Z-part of the coordinate needs to be calculated as well. Therefore, *projective* coordinate systems offer speed advantages only if calculating field multiplications is considerably cheaper than calculating the inverse of a field element. For other projective coordinate systems with practical relevance, see Chapter 3.

2.3 Summary

In this chapter, we introduced the reader to basic mathematical concepts necessary for working with elliptic curves. Groups, fields and finite fields were defined and several of their important mathematical properties were explained. We provided a more detailed look at binary fields and their optimized arithmetic. All these concepts are the foundation for elliptic curves, which were introduced, and their group and addition laws were stated. Additionally we introduced the theoretical background of *projective* coordinate systems, which are essential for high performance implementations of elliptic curves point operations.

Chapter 3

Elliptic Curves in Cryptography

In this chapter, we discuss the standard elliptic curve types used in cryptography. We also explain interesting and fast coordinate systems for each of these curve types. Later, a more detailed look at the security of elliptic curves is given. We discuss some known mathematical attacks and give a brief introduction to implementation attacks. This chapter is based on information published in the following publications: [HMV04], [CF05], [Sma02], [BSS05], and [Kop09].

We start with prime Weierstrass curves which are covered by Section 3.1, followed by binary Weierstrass curves in Section 3.2. Additionally the concept of *Huff curves* is covered in Section 3.3. The security of elliptic curves is discussed in Section 3.4, where we state the important *elliptic curve discrete logarithm problem* (ECDLP) and several types of generic and state-of-the-art attacks on elliptic curve cryptography. This section is followed by Section 3.5, which covers implementation attacks on elliptic curve cryptography with a focus on timing and fault attacks. The chapter is summed up in Section 3.6.

3.1 Prime Weierstrass Curves

A Weierstrass curve over a *prime, finite* field $\mathbb{F}_p$ with $p \neq 2, 3$ is given by all points which fulfill the following equation:

$$E : Y^2 = X^3 + a_4X + a_6, \text{ with } a_4, a_6 \in \mathbb{F}_p \tag{3.1}$$

plus the *point at infinity* $\mathcal{O}$. Additionally, the curve has to be *non-singular.* This can be ensured via its *discriminant*:

$$\Delta E(\mathbb{F}_p) : -(4a_4^3 + 27a_6^2) \neq 0.$$

In the following sections, we will discuss several *projective* coordinate systems, as well as their point addition and point doubling formulas.

Projective Coordinates

Given *projective coordinates*, the *projective* point $P(X : Y : Z)$ corresponds to its *affine* representation, given by (x, y) with $(x, y) = (\frac{X}{Z}, \frac{Y}{Z})$ for all $Z \neq 0$ and otherwise to the *point at infinity* $\mathcal{O}$. The *point at infinity* $\mathcal{O}$ is given as $(0 : 1 : 0)$ and the negation of P is given as $-P = (X : -Y : Z)$. Furthermore, the *projective* equation of the elliptic curve E, as defined in Equation (3.1), is given as:

$$E : Y^2Z = X^3 + a_4XZ^2 + a_6Z^3, \text{ with } a_4, a_6 \in \mathbb{F}_p.$$

Affine coordinates (x, y) correspond to *projective* coordinates $(X : Y : Z)$ simply with $X = x, Y = y$ and $Z = 1$. Currently, the fastest formulas for point doubling, point addition and scaling a point, according to [BL14b], are stated as:

$$\text{Addition: } 11M + 6S, \qquad \text{Doubling: } 5M + 6S, \qquad \text{Scaling: } 1I + 2M.$$

Here, M denotes a *prime* field multiplication, S denotes a *prime* field squaring and I denotes a *prime* field inversion. In the following, we state these addition and doubling formulas:

Addition: For two points $P(X_1 : Y_1 : Z_1),\ Q(X_2 : Y_2 : Z_2) \in E(\mathbb{F}_p)$ the decomposed addition formula for $P_3(X_3 : Y_3 : Z_3) = P + Q$, is given as (taken from [BL14b]):

$$\begin{aligned}
U_1 &= X_1 \cdot Z_2, & U_2 &= X_2 \cdot Z_1, & S_1 &= Y_1 \cdot Z_2,\\
S_2 &= Y_2 \cdot Z_1, & ZZ &= Z_1 \cdot Z_2, & T &= U_1 + U_2,\\
TT &= T^2, & M &= S_1 + S_2, & R &= TT - U_1 \cdot U_2 + a_4 \cdot ZZ^2,\\
F &= ZZ \cdot M, & L &= M \cdot F, & LL &= L^2,\\
G &= (T + L)^2 - TT - LL, & W &= 2 \cdot R^2 - G, & &\\
Y_3 &= R \cdot (G - 2 \cdot W) - 2 \cdot LL, & X_3 &= 2 \cdot F \cdot W, & Z_3 &= 4 \cdot F \cdot F^2.
\end{aligned}$$

Doubling: For a point $P(X_1 : Y_1 : Z_1) \in (E(\mathbb{F}_p))$, the decomposed doubling formula for calculating $2P(X_3 : Y_3 : Z_3)$ is given as (taken from [BL14b]):

$$\begin{aligned}
XX &= X_1^2, & ZZ &= Z_1^2, & w &= a_4 \cdot ZZ + 3 \cdot XX,\\
s &= 2 \cdot Y_1 \cdot Z_1, & ss &= s^2, & sss &= s \cdot ss,\\
R &= Y_1 \cdot s, & RR &= R^2, & B &= (X_1 + R)^2 - XX - RR,\\
h &= w^2 - 2 \cdot B, & & & &\\
Y_3 &= w \cdot (B - h) - 2 \cdot RR, & X_3 &= h \cdot s, & Z_3 &= sss.
\end{aligned}$$

Jacobian Coordinates

Jacobian coordinates were introduced in [CC86]. Jacobian coordinates are so-called *weighted projective* coordinates. The *affine* point $(x : y)$ is represented by the Jacobian point $P(X : Y : Z)$, where $X = \lambda^c x, Y = \lambda^d y, Z = \lambda$ with $\lambda \neq 0$, $c = 2$ and $d = 3$. The *point at infinity* $\mathcal{O}$ is given as $(1 : 1 : 0)$ and the negation of P is given as $-P = (X : -Y : Z)$. The *projective* equation of the elliptic curve E, as defined in Equation (3.1), is given as:

$$E : Y^2 = X^3 + a_4 X Z^4 + a_6 Z^6, \text{ with } a_4, a_6 \in \mathbb{F}_q. \tag{3.2}$$

To convert *Jacobian* coordinates $(X : Y : Z)$ with $Z \neq 0$ to *affine* coordinates (x, y), one has to compute $x = \frac{X}{Z^2}$ and $y = \frac{Y}{Z^3}$, if $Z = 0$ the corresponding point is $\mathcal{O}$. *Affine* coordinates correspond to *Jacobian* coordinates simply by $X = x, Y = y$ and $Z = 1$. Currently the fastest formulas for point doubling, point addition and scaling a point, according to [BL14b], are stated as:

$$\text{Addition: } 11M + 5S, \qquad \text{Doubling: } 1M + 8S, \qquad \text{Scaling: } 1I + 3M + 1S.$$

Again, M denotes a *prime* field multiplication, S denotes a *prime* field squaring and I denotes a *prime* field inversion. In the following, we state these addition and doubling formulas:

Addition: For two points $P(X_1 : Y_1 : Z_1)$, $Q(X_2 : Y_2 : Z_2) \in E(\mathbb{F}_p)$ the decomposed addition formula for $P_3(X_3 : Y_3 : Z_3) = P + Q$ is given as (taken from [BL14b]):

$$\begin{aligned}
Z_1Z_1 &= Z_1^2, & Z_2Z_2 &= Z_2^2, & U_1 &= X_1 \cdot Z_2Z_2,\\
U_2 &= X_2 \cdot Z_1Z_1, & S_1 &= Y_1 \cdot Z_2 \cdot Z_2Z_2, & S_2 &= Y_2 \cdot Z_1 \cdot Z_1Z_1,\\
H &= U_2 - U_1, & I &= (2 \cdot H)^2, & J &= H \cdot I,\\
r &= 2 \cdot (S_2 - S_1), & V &= U_1 \cdot I, & &\\
X_3 &= r^2 - J - 2 \cdot V, & Y_3 &= r \cdot (V - X_3) - 2 \cdot S_1 \cdot J, & Z_3 &= ((Z_1 + Z_2)^2 - Z_1Z_1 - Z_2Z_2) \cdot H.
\end{aligned}$$

Doubling: For a point $P(X_1 : Y_1 : Z_1) \in E(\mathbb{F}_p)$, the decomposed doubling formula for calculating $2P(X_3 : Y_3 : Z_3)$ is given as (taken from [BL14b]):

$$\begin{aligned}
XX &= X_1^2, & YY &= Y_1^2, & A &= YY^2,\\
ZZ &= Z_1^2, & S &= 2 \cdot ((X_1 + YY)^2 - XX - A), & M &= 3 \cdot XX + a_4 \cdot ZZ^2,\\
T &= M^2 - 2 \cdot S, & & & &\\
X_3 &= T, & Y_3 &= M \cdot (S - T) - 8 \cdot A, & Z_3 &= (Y_1 + Z_1)^2 - YY - ZZ.
\end{aligned}$$

Jacobian Chudnovsky Coordinates

In the literature, there are several so called *mixed representations*; among them are the so-called *Jacobian Chudnovsky* coordinates, where a point $P(X : Y : Z : Z^2 : Z^3)$ represents a *Jacobian* point. The redundant values give this coordinate system a slight speed advantage for additions at the expense of slower doublings and additional memory consumption.

Given *Jacobian Chudnovsky coordinates*, the *projective* point $P(X : Y : Z : Z^2 : Z^3)$, corresponds to its *affine* point, given by (x, y), with $(x, y) = (\frac{X}{Z^2}, \frac{Y}{Z^3})$ for all $Z \neq 0$ and to the *point at infinity* $\mathcal{O}$ otherwise. The *point at infinity* $\mathcal{O}$ is given as $(0 : 1 : 0)$ and the negation of P is given as $-P = (X : -Y : Z : Z^2 : Z^3)$.

The *projective* equation of the elliptic curve E is the same as in Equation (3.2). Currently, the fastest formulas for point doubling and point addition, according to [Joy08, Table 1], and the fastest formula for scaling a point, according to [BL14b], are stated as:

Addition: $10M + 4S$, Doubling: $4M + 5S$, Scaling: $1I + 3M + 1S$.

Here, M denotes a *prime* field multiplication, S denotes a *prime* field squaring and I denotes a *prime* field inversion. Next, we state these addition and doubling formulas.

Addition: For two points $P(X_1 : Y_1 : Z_1 : E_1 : F_1)$, $Q(X_2 : Y_2 : Z_2 : E_2 : F_2) \in E(\mathbb{F}_p)$ the decomposed addition formula for $P_3(X_3 : Y_3 : Z_3 : E_3 : F_3) = P + Q$, is given as (taken from [Joy08, Equation 4]):

$$\begin{aligned}
R &= S_1 - S_2, & G &= 4H^3, & V &= 4U_1H^2,\\
S_1 &= 2Y_1F_2, & S_2 &= 2Y_2F_1, & H &= U_1 - U_2,\\
U_1 &= X_1E_2, & U_2 &= X_2E_1, & &\\
X_3 &= R^2 + G - 2V, & Y_3 &= R(V - X_3) - S_1G, & Z_3 &= ((Z_1 + Z_2)^2 - E_1 - E_2)H,\\
E_3 &= Z_3^2, & F_3 &= E_3Z_3. & &
\end{aligned}$$

Doubling: For a point $P(X_1 : Y_1 : Z_1 : Z_1^2 : Z_1^3) \in E(\mathbb{F}_p)$, the decomposed doubling formula for calculating $2P(X_3 : Y_3 : Z_3 : Z_3^2 : Z_3^3)$ is given as (taken from [CF05, p.282]):

$$\begin{aligned}
A &= 4X_1Y_1^2, & B &= 3X_1^2 + a_4 4Z_1^4,\\
X_3 &= -2A + B^2, & Y_3 &= -8Y_1^4 + B(A - X_3), & Z_3 &= 2Y_1Z_1.
\end{aligned}$$

3.2 Binary Weierstrass Curves

A Weierstrass curve over a *binary, finite* field $\mathbb{F}_{2^m}$ is given by all points which fulfill the following *affine* equation:

$$E : Y^2 + a_1XY + a_3Y = X^3 + a_2X^2 + a_4X + a_6, \text{ with } a_1, a_2, a_3, a_4, a_6 \in \mathbb{F}_{2^m},$$

plus the *point at infinity* ($\mathcal{O}$). Additionally, the curve has to be *non-singular*. This can be ensured via its *discriminant*:

$$\Delta E(\mathbb{F}_{2^m}) : a_6 \neq 0.$$

There are two possible *binary* curve types, so-called *non-supersingular* curves, given by Equation (3.3):

$$E : Y^2 + XY = X^3 + a_2X^2 + a_6, \text{ with } a_2, a_6 \in \mathbb{F}_{2^m}, \tag{3.3}$$

and, so-called *supersingular* curves, given by Equation (3.4).

$$E : Y^2 + a_3Y = x^3 + a_4X + a_6 \text{ with } a_2, a_6 \in \mathbb{F}_{2^m}. \tag{3.4}$$

Supersingular curves are susceptible to the MOV attack (see Section 3.4.3) and therefore not adequate for use in cryptography. One can check whether an elliptic curve is *supersingluar* via its *j-invariant*. The *j-invariant* of $E(\mathbb{F}_{2^m})$, denoted by $j(E(\mathbb{F}_{2^m}))$, is given as:

$$j(E(\mathbb{F}_{2^m})) = \frac{a_1^{12}}{\Delta E(\mathbb{F}_{2^m})}.$$

To ensure that a *binary* curve is *non-supersingular*, the *j-invariant* has to be $j(E(\mathbb{F}_{2^m})) \neq 0$. In the following sections, we will discuss several *projective* coordinate systems, as well as their point addition and point doubling formulas.

Projective Coordinates

Given *projective coordinates*, the *projective* point $P(X : Y : Z)$ corresponds to its *affine* point, given by (x, y), with $(x, y) = (\frac{X}{Z}, \frac{Y}{Z})$ for all $Z \neq 0$ and otherwise to the *point at infinity* $\mathcal{O}$. The *point at infinity* $\mathcal{O}$ is given as $(0 : 1 : 0)$ and the negation of P is given as $-P = (X : X + Y : Z)$. Furthermore, the *projective* equation of the elliptic curve $E(\mathbb{F}_{2^m})$, as defined in Equation (3.3), is given as:

$$E : Y^2Z + XYZ = X^3 + a_2X^2Z^+a_6Z^3, \text{ with } a_2, a_6 \in \mathbb{F}_{2^m}.$$

Affine coordinates (x, y) correspond to *projective* coordinates $(X : Y : Z)$ simply by $X = x, Y = y$ and $Z = 1$. Currently the fastest formulas for point doubling, point addition, and scaling a point, according to [BL14b], are stated as

$$\text{Addition: } 14M + 1S, \qquad \text{Doubling: } 7M + 3S, \qquad \text{Scaling: } 1I + 2M.$$

Here M denotes a *binary* field multiplication, S denotes a *binary* field squaring and I denotes a *binary* field inversion. In the following, we state these addition and doubling formulas.

Addition: For two points $P(X_1 : Y_1 : Z_1)$, $Q(X_2 : Y_2 : Z_2) \in E(\mathbb{F}_{2^m})$ the decomposed addition formula for $P_3(X_3 : Y_3 : Z_3) = P + Q$, is given as (taken from [BL14b]):

$$\begin{aligned}
&Y_1Z_2 = Y_1 \cdot Z_2, && X_1Z_2 = X_1 \cdot Z_2, && A = Y_1Z_2 + Z_1 \cdot Y_2,\\
&B = X_1Z_2 + Z_1 \cdot X_2, && AB = A + B, && C = B^2,\\
&D = Z_1 \cdot Z_2, && E = B \cdot C, && F = (A \cdot AB + a_2 \cdot C) \cdot D + E,\\
&Y_3 = C \cdot (A \cdot X_1Z_2 + B \cdot Y_1Z_2) + AB \cdot F, && X_3 = B \cdot F, && Z_3 = E \cdot D.
\end{aligned}$$

Doubling: For a point $P(X_1 : Y_1 : Z_1) \in E(\mathbb{F}_{2^m})$, the decomposed doubling formula for calculating $2P(X_3 : Y_3 : Z_3)$ is given as (taken from [BL14b]):

$$\begin{aligned}
&A = X_1^2, && B = A + Y_1 \cdot Z_1, && C = X_1 \cdot Z_1,\\
&BC = B + C, && D = C^2, && E = B \cdot BC + a_2 \cdot D,\\
&X_3 = C \cdot E, && Y_3 = BC \cdot E + A^2 \cdot C, && Z_3 = C \cdot D.
\end{aligned}$$

Jacobian Coordinates

Jacobian coordinates are so-called *weighted projective* coordinates and the affine point $(x : y)$ is represented by the Jacobian point $P(X : Y : Z)$, where $X = \lambda^c x, Y = \lambda^d y, Z = \lambda$ with $\lambda \neq 0$, $c = 2$ and $d = 3$. The *point at infinity* $\mathcal{O}$ is $(1 : 1 : 0)$ and the negation of P is given as $-P = (X : XZ + Y : Z)$. The *projective* equation of the elliptic curve E, as defined in Equation (3.3), is given as:

$$E : Y^2 + XYZ = X^3 + a_2X^2Z^2 + a_6Z^6, \text{ with } a_2, a_6 \in \mathbb{F}_{2^m}.$$

To convert *Jacobian* coordinates $(X : Y : Z)$ with $Z \neq 0$ to *affine* coordinates (x, y), one has to compute $x = \frac{X}{Z^2}$ and $y = \frac{Y}{Z^3}$. If $Z = 0$ the corresponding point is $\mathcal{O}$. *Affine* coordinates correspond to *Jacobian* coordinates simply by $X = x, Y = y$ and $Z = 1$. Currently the fastest formulas for point doubling, point addition and scaling a point, according to [BL14b], are stated as:

$$\text{Addition: } 14M + 5S, \qquad \text{Doubling: } 4M + 5S, \qquad \text{Scaling: } 1I + 3M + 1S.$$

Again M denotes a *binary* field multiplication, S denotes a *binary* field squaring and I denotes a *binary* field inversion. In the following, we state these addition and doubling formulas:

Addition: For two points $P(X_1 : Y_1 : Z_1)$, $Q(X_2 : Y_2 : Z_2) \in E(\mathbb{F}_{2^m})$ the decomposed addition formula for $P_3(X_3 : Y_3 : Z_3) = P + Q$, is given as (taken from [BL14b]):

$$\begin{aligned}
&O_1 = Z_1^2, && O_2 = Z_2^2, && A = X_1 \cdot O_2,\\
&B = X_2 \cdot O_1, && C = Y_1 \cdot O_2 \cdot Z_2, && D = Y_2 \cdot O_1 \cdot Z_1,\\
&E = A + B, && F = C + D, && G = E \cdot Z_1,\\
&H = F \cdot X_2 + G \cdot Y_2, && Z_3 = G \cdot Z_2, && I = F + Z_3,\\
&X_3 = a_2 \cdot Z_3^2 + F \cdot I + E \cdot E^2, && Y_3 = I \cdot X_3 + G^2 \cdot H. &&
\end{aligned}$$

Doubling: For a point $P(X_1 : Y_1 : Z_1) \in E(\mathbb{F}_{2^m})$, the decomposed doubling formula for calculating $2P(X_3 : Y_3 : Z_3)$ is given as (taken from [BL14b]):

$$\begin{aligned}
&A = X_1^2, && B = A^2, && C = Z_1^2,\\
&D = C^2, && && \\
&X_3 = B + a_6 \cdot D^2, && Z_3 = X_1 \cdot C, && Y_3 = B \cdot Z_3 + (A + Y_1 \cdot Z_1 + Z_3) \cdot X_3.
\end{aligned}$$

López-Dahab Coordinates

López-Dahab coordinates are a coordinate system used with elliptic curves over binary fields $E(\mathbb{F}_{2^m})$. They were introduced by López and Dahab in [LD98]. This subsection sums up some ideas in their paper.

They derive their formulas by using the definition of a *projective plane* P^2 as a set of equivalence classes of triples (X, Y, Z), $\exists(X \vee Y \vee Z) \neq 0$, where a pair of triples is considered equivalent if:

$$\exists\lambda \in \mathbb{F}_{2^m},\ \lambda \neq 0 \text{ for } X_1 = \lambda X_2,\ Y_1 = \lambda^2 Y_2,\ Z_1 = \lambda Z_2.$$

Each of these equivalence classes is a *projective point*. The *point at infinity* $\mathcal{O}$ is the point $(1 : 0 : 0)$. The projective elliptic curve equation, as given in their paper, is:

$$E : Y^2 + XYZ = X^3Z + a_2X^2Z^2 + a_6Z^4, \text{ with } a_2, a_6 \in \mathbb{F}_{2^m}.$$

The conversion of *affine* coordinates (x, y) to *López-Dahab* coordinates $(X : Y : Z)$ is done by setting $X = x, Y = y$, and $Z = 1$. To convert *López-Dahab* coordinates to *affine* coordinates it is necessary to compute $(x.y) = \left(\frac{X}{Z}, \frac{Y}{Z^2}\right)$. It is easy to see that the *projective* and the *affine* plane correspond. The negation of P is given as $-P(X : XZ + Y : Z)$. Currently the fastest formulas for point doubling, point addition and scaling a point, according to [BL14b], are stated as:

$$\text{Addition: } 13M + 4S, \qquad \text{Doubling: } 3M + 5S, \qquad \text{Scaling: } 1I + 2M + 1S.$$

Again M denotes a *binary* field multiplication, S denotes a *binary* field squaring and I denotes a *binary* field inversion. Below we state these addition and doubling formulas.

Addition: For two points $P(X_1 : Y_1 : Z_1),\ Q(X_2 : Y_2 : Z_2) \in E(\mathbb{F}_{2^m})$ the decomposed addition formula for $P_3(X_3 : Y_3 : Z_3) = P + Q$, is given as (taken from [BL14b]):

$$\begin{aligned}
&A = X_1 \cdot Z_2, && B = X_2 \cdot Z_1, && C = A^2,\\
&D = B^2, && E = A + B, && F = C + D,\\
&G = Y_1 \cdot Z_2^2, && H = Y_2 \cdot Z_1^2, && I = G + H,\\
&J = I \cdot E,\\
&Z_3 = F \cdot Z_1 \cdot Z_2, && X_3 = A \cdot (H + D) + B \cdot (C + G), && Y_3 = (A \cdot J + F \cdot G) \cdot F + (J + Z_3) \cdot X_3.
\end{aligned}$$

Doubling: For a point $P(X_1 : Y_1 : Z_1) \in E(\mathbb{F}_{2^m})$, the decomposed doubling formula for calculating $2P(X_3 : Y_3 : Z_3)$ is given as (taken from [BL14b]):

$$\begin{aligned}
&A = Z_1^2, && B = a_6 \cdot A^2, && C = X_1^2,\\
&Z_3 = A \cdot C, && X_3 = C^2 + B, && Y_3 = (Y_1^2 + a_2 \cdot Z_3 + B) \cdot X_3 + Z_3 \cdot B.
\end{aligned}$$

3.3 Binary Huff Curves

The so-called Huff model for elliptic curves was introduced in [Huf48] by Gerald Huff, to study a diophatine problem. The work in [JTV10] generalized the idea to fields of odd characteristic and made it available to cryptography. This section will loosely follow the outline given in [DJ11], as it was the first paper that introduced *binary* Huff curves. All equations in this section were introduced in [DJ11]. A *generalized, affine, binary*

Huff curve H over $\mathbb{F}_{2^m}$ is given by the set $H(\mathbb{F}_{2^m})$, consisting of all *affine* points (x, y) that satisfy Equation (3.5) and the *points at infinity* $(a, b), (1, 0)$ and $(0, 1)$. The *identity element* is given as $\mathbf{o} = (0, 0)$.

$$H : ax(y^2 + fy + 1) = by(x^2 + fx + 1)), \text{ with } a, b, f \in \mathbb{F}^*_{2^m}, a \neq b, f \neq 0. \tag{3.5}$$

H is *birationally equivalent*, over $\mathbb{F}_{2^m}$ for every $m > 3$, to a Weierstrass curve W' given by:

$$W' : v(v + (a + b)fu) = u(u + a^2)(u + b^2),$$

with the *inverse* mappings:

$$\Psi : H \to W' \text{ with } (x, y) \longmapsto \left(\frac{ab}{xy}, \frac{ab(axy + b)}{x^2y}\right), \text{ and}$$
$$\Phi : W' \to H \text{ with } (u, v) \longmapsto \left(\frac{b(u + a^2)}{v}, \frac{a(u + b^2)}{v + (a + b)fu}\right).$$

The Weierstrass curve W' is isomorphic to the Weierstrass curve W, given by:

$$W : v^2 + uv = u^3 + a_2u^2 + a_6, \text{ with } a_2, a_6 \in \mathbb{F}_{2^m}, a_6 \neq 0,$$

with the admissible change of variables:

$$\Theta : (u, v) \leftarrow (\mu^2u, \mu^3(v + su + \sqrt{a_6})), \text{ with } \mu = (a + b)f.$$

Its inverse is given by:

$$\Phi : (u', v') \leftarrow (v^2u', v^3v' + sv^2u' + \sqrt{a_6}), \text{ with } v = \mu^{-1}.$$

For the change of variables to be admissible, certain conditions have to be met, namely:

$$s^2 + s + a_2 + f^{-2} = 0 \text{ and } (a + b)^4f^4\sqrt{a_6} = a^2b^2 \text{ with } s \in \mathbb{F}_{2^m}.$$

The necessary parameter $f \in \mathbb{F}_{2^m}$ is chosen such that $Tr(f^{-1}) = Tr(a_2)$ and $Tr(f\sqrt[8]{a_6}) = 0$. The second parameter $s \in \mathbb{F}_{2^m}$ is calculated by solving the equation $s^2+s+a_2+f^{-2} = 0$. The third parameter $t \in \mathbb{F}_{2^m}$ is a solution to $t^2+(f^4\sqrt{a_6})^{-1}t+1 = 0$. Lastly, the parameters $a \in \mathbb{F}_{2^m}$ and $b \in \mathbb{F}_{2^m}$ can be calculated as $\sqrt{t} = ab^{-1}$.

Standard Coordinates

Similar to *Weierstrass curves*, the *projective* point $P(X : Y : Z)$ on a *generalized Huff curve* corresponds to its *affine* point, given by (x, y), with $(x, y) = (\frac{X}{Z}, \frac{Y}{Z})$ for all $Z \neq 0$ and otherwise to one of the *points at infinity* $\mathcal{O}$ otherwise. The *projective points at infinity* $\mathcal{O}$ are given as:

$$\mathcal{O} = (1 : 0 : 0),\ (0 : 1 : 0),\ \text{and } (a : b : 0).$$

Affine coordinates (x, y) correspond to *projective* coordinates $(X : Y : Z)$ simply by $X = x, Y = y$ and $Z = 1$. On *generalized Huff curves*, the negation of a point cannot be calculated via its *identity* element as this element is not an *inflection point* of the curve. However, the authors of [DJ11] show that the inverse can be calculated as it is the third point of intersection when drawing a tangent at the *identity element.* For the inverse of an *affine* point $P(x, y)$ they give the following formula:

$$-P = P * \left(\frac{b \cdot f}{a + b}, \frac{a \cdot f}{a + b}\right), \text{ with } a, b, f \in \mathbb{F}_{2^m}.$$

The points P and $\left(\frac{b\cdot f}{a+b}, \frac{a\cdot f}{a+b}\right)$ are joined with a line, where $*$ denotes the third point of intersection (counting multiplicities) with the curve. They also show an alternative way where the *birational equivalence* Ψ to a Weierstrass curve is exploited, which leads to an alternative definition for $-P$, namely:

$$-P\left(\frac{y_1(\alpha x_1+1)}{(\beta y_1+1)}, \frac{x_1(\beta x_1+1)}{(\alpha x_1+1)}\right),$$

with $\alpha = \frac{a+b}{b\cdot f}$ and $\beta = \frac{a+b}{a\cdot f}$. In contrast to *binary Weierstrass curves*, where the inverse of a point can be calculated at the costs of one field addition, this operation is, in terms of computing power, very expensive as it involves several *binary* field multiplications and *binary* field inversions.

Currently the computational costs of formulas for point doubling and point addition, according to [GH13], are stated as:

$$\text{Addition: } 15M + 3S, \qquad \text{Doubling: } 6M + 2D + 6S.$$

Again M denotes a *binary* field multiplication, D denotes a *binary* field multiplication by a constant and S denotes a *binary* field squaring. Here we state these addition and doubling formulas.

Addition: For two points $P(X_1 : Y_1 : Z_1)$, $Q(X_2 : Y_2 : Z_2) \in H(\mathbb{F}_{2^m})$ the decomposed addition formula for $P_3(X_3 : Y_3 : Z_3) = P + Q$, is given as (taken from [DJ11]):

$$\begin{aligned}
&m_1 = X_1X_2, && m_2 = Y_1Y_2, && m_3 = Z_1Z_2,\\
&m_4 = (X_1+Z_1)(X_2+Z_2)+m_1+m_3, && m_5 = (Y_1+Z_1)(Y_2+Z_2)+m_2+m_3, &&\\
&m_6 = m_4(m_2+m_3), && m_7 = m_5(m_1+m_3), && m_8 = m_1m_2+m_3^2,\\
&m_9 = m_8 + (X_1Y_1+Z_1^2)(X_2Y_2+Z_2^2), && &&\\
&X_3 = m_6m_9, && Y_3 = m_7m_9, && Z_3 = m_4m_5m_8.
\end{aligned}$$

Doubling: For a point $P(X_1 : Y_1 : Z_1) \in H(\mathbb{F}_{2^m})$, the decomposed doubling formula for calculating $2P(X_3 : Y_3 : Z_3)$ is given as (taken from [DJ11]):

$$\begin{aligned}
&m_1 = X_1Y_1 + Z_1^2, && m_2 = X_1Z_1, && m_3 = Y_1Z_1,\\
&X_3 = \alpha\cdot[m_2(Y_1+Z_1)^2]^2, && Y_3 = \beta\cdot[m_3(X_1+Z_1)^2]^2, && Z_3 = [m_1(m_1+m_2+m_3)]^2.
\end{aligned}$$

Here $\alpha = \frac{a+b}{b}$ and $\beta = \frac{a+b}{b}$, where a and b are parameters in Equation (3.5).

WZ Coordinates

Devigne and Joye introduced, a new *projective* coordinate system for Huff curves in [DJ11]: so-called *WZ coordinates*. *WZ coordinates* make it possible to use *differential* addition and doubling formulas without being forced to use an *affine* coordinate system. *Affine* coordinate systems come with a severe performance penalty as they require costly inversions. An *affine* point $P = (\frac{X_1}{Z_1}, \frac{Y_1}{Z_1})$ is represented in *WZ coordinates*, with $\theta \in \mathbb{F}_2^*$, as follows:

$$(W : Z) = \begin{cases} (\theta w(P) : \theta) = (\theta X_1Y_1 : \theta Z_1^2) & \text{if } P \neq (a : b : 0), \text{ and} \\ (1 : 0) & \text{otherwise.} \end{cases}$$

ω is a coordinate function that fulfills $\omega(P) = \omega(-P)$, for example:

$$\omega : (x, y) \mapsto (x \cdot y). \tag{3.6}$$

3.4 Security Properties of Elliptic Curves

In this section, we will discuss the security properties of elliptic curves, starting with the *elliptic curve discrete logarithm problem* (ECDLP) in Section 3.4.1. This is followed by mathematical attacks, wherein we first explain so-called *generic attacks* against the ECDLP in Section 3.4.2. These are attacks that do not have special requirements concerning the curve they are attacking, and secondly we explain *state-of-the-art* attacks in Section 3.4.3. These are attacks with more specific requirements. In Section 3.5, we will give a short introduction to *implementation attacks*, with a special focus on *timing attacks* and *fault attacks.*

3.4.1 The Elliptic Curve Discrete Logarithm Problem

The *elliptic curve discrete logarithm problem* (ECDLP) is the mathematically hard problem on which the security assumptions of elliptic curve cryptography are based.

Definition 3.4.1 (*ECDLP*)**.** Let G be an elliptic curve group generated by P. Given $Q \in G$, the ECDLP is defined as finding an integer $0 \leq k \leq |P| - 1$ such that $kP = Q$.

3.4.2 Generic Attacks against the ECDLP

An algorithm is considered *generic* if all its computations are any one of the following (as stated by the authors of [CF05]):

- the *composition* of two group elements or,
- calculating the *inverse* of a group element or,
- *comparing* two group elements.

Therefore, the following *generic* algorithms for attacking the ECDLP work on any group without relying on any special group properties.

Pollard's ρ-Method

This section is a summary of the details and information given in [CF05] and [BSS99], as well as in [HMV04]. Pollard's method can be used to solve the ECDLP ($kP = Q$) as given in Definition 3.4.1. The initial idea is to find a so-called *collision* by taking advantage of the *birthday paradox* given in Theorem 3.4.1.

Theorem 3.4.1 (*Birthday paradox*)**.** While selecting randomly with replacement from an urn with n labeled balls, one can expect to select a ball with the same label a second time after $\sqrt{\frac{\pi n}{2}} \approx 1.25\sqrt{n}$ draws from the urn.

To avoid the costly storage and computational requirements of a naive attack using the *birthday paradox*, where it is necessary to store each previously selected element and compare each element with all previously chosen elements, Pollard's method uses a *deterministic, pseudorandom iterating function* ($f : G \to G$) to conduct a walk over the group G generated by P and calculate $log_P(Q)$ with $Q \in G$. This iterative walk consists of a sequence s of steps, where each step s_i starts by selecting two random integers $c, d \in [0, n-1]$, with $n = |G|$, followed by calculating $s_i = cP + dQ$. At a certain point the sequence will start to loop, meaning that a step $s_j = c'P + d'Q = cP + dQ$ is obtained a second time. If two steps s_i, s_j with $i \neq j$ are equal, such that $cP + dQ = c'P + d'Q$, a so-called *collision* has happened. In this case, the *discrete logarithm* is given as follows

$$log_P(Q) = \frac{c - c'}{d' - d} \bmod n.$$

All steps in s are in a finite set of size n, therefore collisions must occur. As the *iterating function* behaves randomly, the approximate likelihood l of a collision is $l = \sqrt{\frac{\pi n}{2}}$ due to the *birthday paradox* (given in Theorem 3.4.1). A visualisation of the steps of s forms the Greek letter ρ, as shown in Figure 3.1, where one can also see that after a collision s starts to cycle forever. In order to be able to efficiently detect that a collision happened, and as a consequence the sequence s is looping, a so-called *cycle detection algorithm* is necessary.

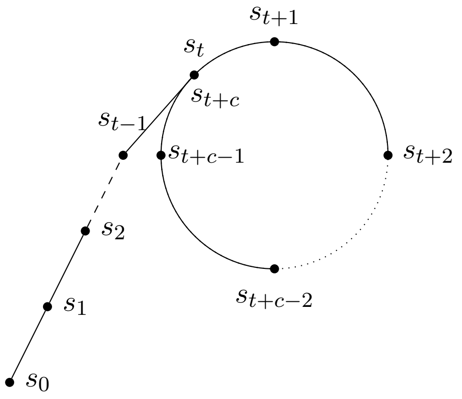

Figure 3.1: The ρ shaped figure formed by all steps in the sequence s

Cycle detection is a problem that occurs in many areas of computer science, therefore multiple algorithms which solve this problem already exist. One of them is *Floyd's algorithm*, which goes back to an idea for finding cycles in directed graphs, introduced in [Flo67]. A cycle detection algorithm introduces two important parameters for the sequence s, namely t the *tail length* and c the *cycle length*. The *tail length* gives the number of elements until the cycle is reached; the *tail* is given by the elements $\{s_0, s_1, \dots s_{t-1}\}$ in Figure 3.1. Similarly, the *cycle length* gives the number of elements and therefore the "size" of the cycle. In Figure 3.1 the *cycle* consists of the elements $\{s_t, \dots, s_{t+c}\}$. These two parameters, in the case of *Floyd's algorithm*, have the following expectancies: $t \approx \sqrt{\frac{\pi n}{8}}$ and $c \approx \sqrt{\frac{\pi n}{8}}$. Both expectancies are under the assumption that the *iteration function* behaves randomly. *Floyd's algorithm* detects a loop if $s_i = s_{2i}$ which is the case with $i = c(1 + \lfloor \frac{t}{c} \rfloor)$.

Floyd's algorithm drastically reduces the storage and computational requirements, compared to a naive attack, as only $s_i = s_{2i}$ for each $s_i \in s$ need to be compared to reliably detect a loop, and therefore the storage requirements are only one pair (s_i, s_{2i}) of iteration steps.

The expected number of group operations leading to the collision depends heavily on the *cycle finding algorithm* as well as the close to random behavior of the *iterating function*. The typical approach for getting an almost random iterating function is to divide the group into several sets from which to randomly chose elements. For the currently best iterating functions [Tes98] gives $\approx 1.453\sqrt{|G|}$ iterations to solve the ECDLP. It is also worth mentioning that Pollard's-ρ algorithm can be parallelized very efficiently, where for M additional processors a linear speedup of M can be achieved. The authors of [HMV04] give Algorithm 1 as a single processor algorithm.

Algorithm 1 Pollard's rho algorithm for the ECDPL (single processor) [HMV04, Algorithm 4.3].

```
Input: $P \in E(\mathbb{F}_q)$ of prime order $n$, $Q \in \langle P \rangle$.
Output: The discrete logarithm $l = log_P Q$.
 1: Select the number $L$ of branches (e.g., $L = 16$ or $L = 32$)
 2: Select a partition function $H : \langle P \rangle \to \{1, 2, \ldots, L\}$
 3: for $j$ from 1 to $L$ do
 4:     Select $a_j, b_j \in_R [0, n-1]$
 5:     Compute $R_j = a_j P + b_j Q$
 6: end for
 7: Select $c', d' \in_R [0, n-1]$ and compute $X' = c'P + d'Q$
 8: Set $X'' \leftarrow X'$, $c'' \leftarrow c', d'' \leftarrow d'$
 9: repeat
10:     Compute $j = H(X')$
11:     Set $X' \leftarrow X' + R_j, c' \leftarrow c' + a_j \bmod n, d' \leftarrow d' + b_j \bmod n$
12:     for $i$ from 1 to 2 do
13:         Compute $j = H(X'')$
14:         Set $X'' \leftarrow X'' + R_j, c'' \leftarrow c'' + a_j \bmod n, d'' \leftarrow d'' + b_j \bmod n$
15:     end for
16: until $X' = X''$
17: if $d' = d''$ then
18:     return failure
19: else
20:     compute $l = (c' - c'')(d'' - d')^{-1} \bmod n$
21:     return $l$
22: end if
```

The Pohlig-Hellman Algorithm

This section is based on information given in [BSS99] and [CF05], and follows the explanations given in [WT02]. The *Pohlig-Hellman* algorithm can solve the *discrete logarithm* (DL) (as defined in Section 2.1.1) in a given group G if a factorization of $|G|$ is known. The *Pohlig-Hellman* algorithm reduces the DL problem in the large group G to several DL problems in probably much smaller subgroups, given by the prime factors of $|G|$. It gives speed advantages compared to other methods only if the prime factors are significantly smaller than $|G|$. The *Pohlig-Hellman* algorithm relies heavily on the *Chinese Remainder Theorem* (CRT) which is introduced next.

The Chinese Remainder Theorem: The *Chinese Remainder Theorem* states that given a set of congruences, there exists a *unique* solution to this set of congruences modulo the product of all moduli, given that they are pairwise *coprime*. This is stated more formally in Theorem 3.4.2.

Theorem 3.4.2 (Chinese Remainder Theorem)**.** Given $m_1, \ldots, m_n \in \mathbb{Z}$, where (m_i, m_j) are pairwise *coprime* for all $i \neq j$ and given $a_1, \ldots, a_n \in \mathbb{Z}$, there exists an $x \in \mathbb{Z}$ for all:

$$x \equiv a_i \bmod m_i, \quad \text{for all } 1 \leq i \leq n.$$

Furthermore, this x is *unique* module $m_1 \ldots m_n$.

The *Chinese Remainder Theorem* leads to a generic algorithm, given in Algorithm 2.

Algorithm 2 Chinese Remainder Algorithm

```
Input: A set of equations of form $x \equiv a_i \bmod m_i$.
Output: A solution $x$ to the set of equations.
1: calculate $m = \prod_{i=1}^{r} m_i = m_i M_i$ where $M_i = \frac{m}{m_i}$
2: calculate $n_i$ such that $n_i M_i \equiv 1 \bmod m_i$
3: return $x = \sum_{i=1}^{r} a_i M_i n_i \pmod{m}$
```

Example 3.4.1 (*Chinese Remainder Example*)**.** Given the following system of congruences: $x = 7 \bmod 26, x = 1 \bmod 11$ and $x = 3 \bmod 17$. One starts by calculating:

$$\begin{aligned} m &= 26 \cdot 11 \cdot 17 = 4862, & m_1 &= 26, M_1 = 11 \cdot 17 = 187, \\ m_2 &= 11, M_2 = 26 \cdot 17 = 442, & m_3 &= 17, M_3 = 26 \cdot 11 = 286. \end{aligned}$$

Next the so-called n_i are calculated, such that $n_i M_i \equiv 1 \bmod m_i$:

$$\begin{aligned} 187 \cdot n_1 &\equiv 1 \bmod 26, & 442 \cdot n_2 &\equiv 1 \bmod 11, & 286 \cdot n_3 &\equiv 1 \bmod 17, \\ n_1 &= -5, & n_2 &= -5, & n_3 &= -6. \end{aligned}$$

Now a solution can be calculated as follows:

$$\begin{aligned} x &= \sum_{i=1}^{r} a_i M_i n_i \pmod{m}, \\ &= (7 \cdot 187 \cdot -5) + (1 \cdot 442 \cdot -5) + (3 \cdot 286 \cdot -6) \pmod{4862} = -13903 \pmod{4862}, \\ &= 683. \end{aligned}$$

Now we have a look at the *Pohlig-Hellman* algorithm in the context of groups given by an elliptic curve. We follow the outline given in [HMV04, Section 4.1.1]. The *elliptic curve discrete logarithm problem* ECDLP (as introduced in Definition 3.4.1) finding an integer $0 \leq k \leq |P| - 1$ such that $k = log_P(Q)$, given that a group G is an elliptic curve group generated by P and $Q \in G$.

The prime factorisation of $|G|$ is then given, as $|P| = n = \prod_{i=1}^{r} p_i^{e_i}$. As every $p_i^{e_i}$ is *coprime* to all other factors, the strategy is to transfer the ECDLP to all i subgroups, each of order $|p_i^{e_i}|$. By solving the ECDLP in all r subgroups, all k_{p_i} such that:

$$k_{p_i} \equiv k \pmod{p_i^{e_i}} \text{ with } 1 \leq i \leq r, \tag{3.7}$$

are determined. The CRT (as given in Section 3.4.2) is applied to these r equations and gives a *unique* solution $k \pmod{n}$ to the ECDLP for the group G.

For every $p_i^{e_i}$ with $e_i > 1$, some intermediate steps are necessary as the solution mod $p_i^{e_i}$ is calculated using the CRT. As $p_i^{e_i}|n$, one calculates $k_{p_i} = z \bmod p_i^t$ for $t = \{1, \ldots, e_i\}$, which is referred to as *lift* of the value. The sought-after z is written in *p_i-ary* representation, with $0 \leq z_i < p_i$, as:

$$z = z_0 + z_1 p_i + z_2 p_i^2 + \cdots + z_{e_{i-1}} p_i^{e_{i-1}}.$$

Then, every z_i is calculated sequentially, starting with z_0. Following along with the explanations given in [HMV04], one can construct the following formulas using the *p_i-ary* representation. One starts by transferring the ECDLP to the subgroup of order p_i by calculating $P_0 = \left(\frac{n}{p_i}\right) P$ and $Q_0 = \left(\frac{n}{p_i}\right) Q$. The order of P_0 is p_i, as $p_i P_0 = \frac{p_i n}{p_i} P = nP$, the same can be checked for Q_0. This leads to the following equation (taken from [HMV04]):

$$Q_0 = \frac{n}{p_i} Q = k \left(\frac{n}{p_i} P\right) = kP_0 = z_0 P_0.$$

Therefore, z_0 can be calculated by solving $z_0 = log_{P_0}(Q_0)$. We use this intermediate result to calculate z_1 as follows (taken from [HMV04]):

$$\begin{aligned} Q_1 &= \frac{n}{p_i^2}(Q - z_0 P) = \frac{n}{p_i^2}(k - z_0)P = (k - z_0)P\left(\frac{n}{p_i^2} P\right) \\ &= (z_0 + z_1 p_i - z_0)\left(\frac{n}{p_i^2} P\right) = z_1 \left(\frac{n}{p_i} P\right) = z_1 P_0. \end{aligned}$$

Again, the ECDLP has to be solved in $\langle P_0 \rangle$ and z_1 can be calculated as $z_1 = log_{P_0}(Q_1)$. The authors of [HMV04] give the following generic formula for calculating $z_t = log_{P_0}(Q_t)$:

$$Q_t = \frac{n}{p_i^{t+1}} \left(Q - z_0 P - z_1 p_i P - z_2 p_i^2 P - \cdots - z_{t-1} p_i^{t-1} P\right),$$

given that $z_0, \ldots, z_{t-1}$ are already calculated. This leads to a system of equations similar to Equation (3.7), which can be solved by the CRT.

3.4.3 State-of-the-Art Attacks against the ECDLP

The main difference in the attacks described in the following section compared to the attacks shown in Section 3.4.2 is that they involve certain assumptions concerning the elliptic curves they are attacking.

The MOV Attack

The first sub-exponential algorithm which solves the ECDLP on some closely defined groups was given 1991 by Menezes, Okamoto and Vanstone (which explains the name MOV attack) in [MVO91] and uses the *Weil pairing*. Later on, this attack was generalized and enhanced by others, most notably Frey and Rück in [FR94] and [FMR99]. The idea behind this type of attack is to use a so-called *pairing* on an elliptic curve to transfer the ECDLP in an elliptic curve $E(\mathbb{F}_q)$ to a DLP in an extension field $\mathbb{F}_{q^k}$ of $\mathbb{F}_q$, for which subexponential attack algorithms, most notably the *index-calculus method* (described in Section 3.4.3), are known. This is done by using the *Weil pairing* to establish an *isomorphism* between the subgroup of E, generated by the prime order point P and the subgroup in $\mathbb{F}_{q^k}$ given by its n^{th} *root of unity*, where n is the order of P and assumed to be $n \geq 3$. To follow the attack it is necessary to introduce the *Weil pairing*.

The Weil Pairing: A *Weil pairing* on an elliptic curve $E(\mathbb{F}_q)$ is given as a mapping:

$$e_n : E[n] \times E[n] \rightarrow \mu_n(\mathbb{F}_{q^k}),$$

where $n \in \mathbb{N}$ is relatively prime to q and μ_n is the n^{th} *root of unity* (meaning that $e_n(P, Q)^n = 1$ for all $P, Q \in E[n]$).

The so-called *embedding degree* k, with respect to n, is the smallest positive value that fulfills $n|(q^k - 1)$. This property is based on the observation that all points of order n in E mapped by e_n are *isomorphic* to points in the subgroup, consisting of all n^{th} *roots of unity*, of the minimal field $\mathbb{F}^*_{q^k}$ that contains all n^{th} *roots of unity*. This means that the order of $|(\mathbb{F}_{q^k})| = (q^k - 1)$ must be divided by the order n of the subgroup created by the mapped points from $E[n]$. For *supersingular curves*, k is always small. An interesting paper discussing the *minimal embedding field* and its impact on the security of pairing based cryptography is [Hit07].

Furthermore, the pairing e_n has the following basic properties for $P, Q, P', Q' \in E[n]$:

- *identity* such that $e_n(P, P) = 1$ for all $P \in E[n]$,
- e_n is *bilinear* meaning that $e_n(P + P', Q) = e_n(P, Q) \cdot e_n(P', Q)$ as well as $e_n(P, Q + Q') = e_n(P, Q) \cdot e_n(P, Q')$,
- e_n is *non-degenerate* meaning that there is no $P \neq \mathcal{O}$ such that $e_n(P, Q) = 1$ for all Q and no $Q \neq \mathcal{O}$ such that $e_n(P, Q) = 1$ for all P,
- e_n is *alternating* as $e_n(P, P) = 1$ for all P and therefore $e_n(P + Q, P + Q) = e_n(P, P) \cdot e_n(P, Q) \cdot e_n(Q, P) \cdot e_n(Q, Q) = e_n(P, Q) \cdot e_n(Q, P) = 1$ from which follows that $e_n(P, Q) = e_n(Q, P)^{-1}$.

The given *Weil pairing* is used in the attack as an important step in Algorithm 3 given by the authors of [MVO91].

Algorithm 3 [MVO91, Algorithm 2]. Algorithm that reduces the ECDLP to a DLP in a finite field

```
Input: An element $P \in E(F_q)$ of order $n$, and $R \in \langle P \rangle$.
Output: An integer $l$ such that $R = lP$.
1: Determine the smallest integer $k$ such that $E[n] \subseteq E(F_{q^k})$
2: Find $Q \in E[n]$ such that $\alpha = e_n(P, Q)$ has order $n$
3: Compute $\beta = e_n(R, Q)$
4: Compute $l$, the discrete logarithm of $\beta$ to the base $\alpha$ in $F_{q^k}$
```

The order n of point P has to be *coprime* to q and $n|q^k - 1$. For step 4 in Algorithm 3 the *index-calculus method*, as given in the next section, is suitable to calculate $\alpha^l = \beta$.

Index-Calculus Method: This section follows the excellent explanation of the *Index-Calculus method* given in [HMV04, Section 4.1.3] and uses additional information given in [Die12]. The *Index-Calculus method* is the state-of-the-art attack for the DLP (as stated in Definition 2.1.1) in the multiplicative group $\mathbb{F}^*_q$ of a *finite* field $\mathbb{F}_q$. According to McCurley in [DM89], the main idea for the *Index-Calculus method* was first stated in the early 20th century and then reinvented in the late 1970s. As outlined in [HMV04], the *index-calculus method* can be sketched in four steps, as given in Algorithm 4.

Algorithm 4 The Index-Calculus Method

```
Input: A generator $g$, a modulus $n$ and an argument $h$.
Output: A solution $x$ to $g^x \equiv h\ (mod\ n)$.
1: find a so-called factor base $S$
2: find linear relations
3: solve the linear system of equations given by those relations
4: extract the solution
```

Now follows a more detailed insight into each of the steps given in Algorithm 4, which calculates $log_g(h)$ for the given group $G = \langle g \rangle$. G is of order $n = |G|$ and $h \in G$.

Definition 3.4.2 (*Smoothness Bound*)**.** Given a *positive* integer B as so-called *smoothness bound*, then a *positive* integer a is called *B-smooth* if all its *prime* factors are smaller or equal to B.

Step 1: The *factor base* $S \subset G$ should consist of a subset of elements in G such that a "significant" number of all elements in G can be expressed as a linear combination of few elements in S and with small coefficients. In practice if G is the multiplicative group of a *prime* field $\mathbb{F}_q$, all elements in G can be represented by the integers of size less than q. A *smoothness bound* B is chosen. This ensures that the biggest prime factor of each of the chosen positive integers is smaller than or equal to B. The *factor base* S is then given by all primes in the integer representation of G, which are smaller than or equal to B, giving S as $S = \{p_i\}$ with $2 \leq p_i \leq B$ with all *primes* $p \in G$, let x denote the *cardinality* of S.

If G is the multiplicative group of a *binary* field $\mathbb{F}_{2^m}$, all elements of G can be represented by polynomials of degree smaller than m. The *factor base* S is given by the *irreducible* polynomial of degree lower than or equal to the *smoothness bound* B. This ensures that every element in G that can be factored over S has irreducible factors that are of smaller degree than B, again x denotes the *cardinality* of S.

Step 2: One chooses a random number $0 \leq k < n$ such that all the factors of g^k lie in S, or more formally (taken from [Equation 4.6][HMV04]):

$$g^k = \prod_{i=1}^{x} p_i^{c_i} \text{ with } c_i \in \mathbb{N}. \tag{3.8}$$

A relation of indices can be obtained by taking the logarithms to the base g in the Equation (3.8). Such a relations is given as follows (taken from [Equation 4.7][HMV04]):

$$k \equiv \sum_{i=1}^{x} c_i log_g(p_i) \text{ (mod } n).$$

Here the estimated number of relations is *slightly larger* than x [HMV04]. Step 2 also shows a trade-off as the bigger the *factor base* S, the easier it is to find a g^k which factors over S, but it also increases the complexity of the system of equations that need to be solved in the next step.

Step 3: The system of equations is obtained by calculating $log_g(p_i)$ for all $1 \leq i \leq x$.

Step 4: After all logarithms of elements in S are calculated, a solution is extracted. To compute $log_g(h)$ numbers $0 \leq k < n$ are selected until $g^k h$ is a product of elements in s, more formally (taken from [Equation 4.8][HMV04]):

$$g^k h = \prod_{i=1}^{x} p_i^{d_i} \text{ with } d_i \in \mathbb{N}.$$

Similar to Step 1, one can apply the logarithm to both sides of the equation and retrieves the solution as (taken from [Equation 4.9][HMV04]):

$$log_g(h) = -k + \sum_{i=1}^{x} d_i log_g(p_i) \bmod n.$$

It should be mentioned that the *Index-Calculus method* can not directly solve the ECDLP in *elliptic curve groups*. This is due to the lack of *prime elements*, which makes generic methods for finding an efficient *factor base* infeasible. In [Mil86, page 423], Miller argues in more detail, as to why it is highly unlikely that an *Index-Calculus method* attack will ever work directly on elliptic curves.

The SSSA Attack

The *SSSA attack* works on so-called *prime field anomalous curves*, which are elliptic curves where the *trace of Frobenius* is one. This is the case if the *group order* of a curve $E(\mathbb{F}_q)$ is equal to the order of the underlying finite field. These curves have the property that they are *cyclic* and *isomorphic* to $(\mathbb{Z}_p, +)$, therefore the ECDLP of the curve can be reduced to the DLP in $(\mathbb{Z}_q, +)$ and solved in subexponential time if a suitable, meaning efficiently computable, *isomorphism* can be found. The attack was stated in three variations by Semaev in [Sem98], Satoh and Araki in [SA98] and also by Smart in [Sma99]. The attack can be roughly outlined as in Algorithm 5.

Algorithm 5 Outline of the SSSA attack

```
Input: Points $P, Q \in E(F_q)$, an isomorphism $\phi : E(\mathbb{F}_q) \to \mathbb{F}_q$ such that $\phi(P) \neq 0$.
Output: $P = l \cdot P$.
1: Calculate $\alpha = \phi(Q), \beta = \phi(P)$
2: Find $0 \leq l \leq p - 1$ such that $l = \beta\alpha^{-1} \bmod p$ using Algorithm 18
3: $l\alpha \equiv \beta \bmod p$ therefore $P = l \cdot Q$
4: return $l$
```

This attack renders all elliptic curves whose group order is equal to the order of the underlying finite field unsuitable for cryptographic purposes.

3.5 Implementation Attacks against ECC

Implementation attacks, as the name suggests, do not try to cryptographically break a theoretically secure cryptographic scheme, but instead focus on the concrete *implementation* of said scheme. The goal is to extract, via so-called *side channels*, enough information about *secret* data, for example a secret key, such that it can be reconstructed or partially reconstructed afterwards. There exists a wide variety of different side channels which can leak different amounts of information. Some examples are: the power consumption, magnetic emanation, acoustic emanation (sound), or execution time of operations on hardware architectures or the side channel of response time for interactive cryptographic protocols. In the following sections, we will focus on implementation attacks that are most relevant in the context of applications running on servers. Section 3.5.1 gives an introduction to *timing attacks*, followed by *fault attacks* in Section 3.5.2.

3.5.1 Timing Attacks

In 1996, Kocher was the first who published the concept of a timing attack in [Koc96]. He described the attack as a signal detection problem. The signal is expressed as timing variations of operations that depend on secret data, which is camouflaged in a sum of noise introduced by inaccurate timing measurements and/or noise generated by other properties of the cryptographic system. A simple example illustrating a timing side channel, is the elliptic curve point multiplication method called *right-to-left binary method for point multiplication* which works as given in Algorithm 6. By studying the algorithm and assuming that performing a *double* operation does not take the same amount of time as performing an *addition*, an attacker can guess if a bit of k is 0 or 1. With this knowledge, recovering the scalar k is an iterative process where one bit of k can be recovered by executing one iteration of the algorithm's *for* loop. One starts with k_0 and measures if an addition or a doubling is performed, which is equivalent to knowing if k_0 is 1 or 0. The knowledge of k_0 is then used during the attack on k_1 and so forth and so on, until all bits of k are determined. Depending on the attacked implementation, it may not be necessary to recover all secret bits, e.g., the authors of [NS03] showed that if a few bits of the random nonce are leaked, the *private-key* used in the *elliptic curve digital signature algorithm* (ECDSA) can be recovered. In order for a timing side channel attack to be feasible, precise timing information is of essence. This was one of the reasons why timing side channel attacks where considered to be local attacks until, in 2003, Brumeley and Boneh (in [BB05]) successfully extracted an RSA private key from a server over their local network. In 2011, Brumley and Tuveri published in [BT11] an example for a remote attack on OpenSSL's binary elliptic curve implementation of the Montgomery ladder (detailed in Section 4.1). They attacked the Elliptic Curve Digital Signature Algorithm (ECDSA) and exploited a side channel which leaked parts of the random nonce used in the ECDSA. As this nonce is used for *blinding* the private key, a *Lattice Attack* could be mounted to recover the secret key.

Defenses against Timing Attacks

An implementation which is not vulnerable to a timing attack doesn't have a correlation between secret values and the implemented algorithm's execution time. This means that the algorithm has to work in *constant* time, regardless of its input. Sadly there is no perfect countermeasure against all side-channel attacks; one merely makes known attacks infeasible. The scientific community came up with different concepts to ensure the timing attack resistance of implementations. Some of them will be explained in more detail in this section.

The authors of [CF05] state that making the implementation *regular* thwarts so-called *simple side-channel attacks* as every link between secret data and observable outputs is broken. The assumptions are that the attacker is able to observe *one* execution of the scalar multiplication algorithm. The authors recommend a variety of countermeasures, namely:

- dummy arithmetic operations,
- the use of *unified* addition/doubling formulas,
- the *Montgomery ladder* (as described in Section 4.1) for scalar multiplication.

There are several advantages and drawbacks for every one of the approaches. Some examples of drawbacks are: dummy operations can be attacked with *safe-error attacks* (described in Section 3.5.2), *unified* formulas are slower than their speed-optimized counterparts and the *Montgomery ladder* does not support speedups through precomputations.

3.5.2 Fault Attacks

Fault attacks comprise a wide variety of mostly hardware based attacks where a wide range of physical environment conditions can be altered to induce a fault in a chip. Some examples are exposing the chip to high temperatures, supply voltage outside of the specifications, or exposure to radiation or magnetic fields with the goal of inducing an error in the computation. This error should help a cryptographer to deduce some otherwise secret information from the chip.

In context of elliptic curve cryptography, faults can also be induced if the validity of elliptic curve public keys is not checked properly. The authors of [ABM+03], provide practical, so-called *invalid curve* attacks on several protocols that make use of elliptic curve cryptography.

Safe-Error Attacks

The rough outline of a *safe-error attack* is as follows. A fault is injected into the computation of an implementation that uses dummy operations. Later on, the output of the computation is checked as to whether the result is valid. If so, this means that the error was introduced at a time when a dummy operation was calculated. For this attack, the adversary needs physical control over the device. Therefore, it mainly poses a threat to smart cards. This type of attack is not possible if all operations contained in an algorithm are *effective* and *regular*, as every error influences the result of the calculation. This is, for example, the case for the *Montgomery ladder* (described in Section 4.1) as well as *Joye's Double-and-Add* algorithm (described in Section 4.5).

Defenses against Fault Attacks

The authors of [CF05] suggest checking the result of the cryptographic operation as a good countermeasure to mitigate fault attacks. The result must be a valid point which lies on the used elliptic curve. They also bring to the reader's attention that this kind of checking, where the result is held back in case of errors, may have a downside. It can be used in another *safe-error attack*. Therefore, they additionally recommend randomizing the scalar.

The authors of [ABM+03] recommend checking the following four properties (taken from [Definition 1][ABM+03]) of a received *public* key to ensure that a point $W = (x_W, y_W) \in E$ is valid and therefore thwart their attacks:

1. $W \neq \mathcal{O}$,
2. x_W and y_W are properly represented elements of $\mathbb{F}_q$,
3. W satisfies the *defining* curve equation of $E(\mathbb{F}_q)$,
4. $n \cdot W = \mathcal{O}$ with $n = |P|$, where P is a *prime order base point* of $E(\mathbb{F}_q)$.

Checking beforehand is only effective if one can be certain that during the following cryptographic operations no additional fault can be induced.

3.6 Summary

In this chapter, we introduced the standard elliptic curve types used in cryptography, accompanied by a subset of the available coordinate systems. We showed the fastest available doubling and addition formulas as well as their computational costs for easy comparison. Additionally, we took a deeper look into the security of elliptic curves and explained several well known mathematical attacks starting with older generic attacks that work without special requirements, followed by newer, more specific attacks. Those attacks were the reason why trust in elliptic curves was shaken in the beginning of the 1990s, and as a result certain classes of elliptic curves are no longer used in ECC. Later on, we gave a brief introduction to implementation attacks on elliptic curve cryptography, especially to *timing* and *fault* attacks. Implementation attacks are a constant threat to the security of any cryptographic system. Theoretical concepts and assumptions tend to be hard to implement in real world computers, and implementation errors can compromise the security of systems that are theoretically secure. Implementation attacks can therefore not be overlooked, and the defenses we mentioned have to be considered when implementing cryptographic systems.

Chapter 4

Scalar Multiplication on Elliptic Curves

In this chapter, we describe several ways in which a scalar multiplication on an elliptic curve can be executed. Scalar multiplication is an essential building block for public-key elliptic curve cryptography and has a significant influence on the execution time of ECC algorithms. Therefore, optimized scalar multiplication methods are vital for good ECC performance. Optimizations can be realized in several ways, e.g., by using more efficient algorithms, by exploiting special elliptic curve properties like isomorphisms or by using more efficient addition and doubling formulas based on special coordinate systems.

We start with basic scalar multiplication methods in Section 4.1. One of the most crucial design decisions for a scalar multiplication algorithm is time-memory trade-off. In this context, time-memory trade-off means computing and storing some intermediate values that solely depend on a previously fixed point P beforehand. The algorithm gains a reasonable speedup while executing scalar multiplications with P afterwards. In Section 4.2, we give several scalar multiplication methods which utilize precomputations. A scalar multiplication method that uses efficiently computable endomorphisms in combination with precomputations is discussed in Section 4.3. If there is no memory for precomputations available, various alternative algorithms with small memory footprints and acceptable performance do exist. We introduce two different kinds. First, we introduce the *Montgomery ladder* multiplication method in Section 4.4. Second, we also describe *Joye's Double-and-Add* multiplication method in Section 4.5. It should be noted that the *Montgomery ladder* and *Joye's Double-and-Add* method have the additional advantage of being resistant to many implementation attacks. This chapter concludes with a short summary in Section 4.6.

4.1 Basic Scalar Multiplication Methods

In this section, we first introduce some basic scalar multiplication methods. This is followed by selected ideas for scalar encodings that will improve the performance of several scalar multiplication methods given in later parts of the chapter.

Double-and-Add Multiplication

One of the most basic scalar multiplication algorithms is the *Double-and-Add* algorithm. It exists in different varieties; one is illustrated in Algorithm 6.

Algorithm 6 Right-to-left binary method for point multiplication [HMV04, Algorithm 3.26]

```
Input: $k = (k_{\ell-1}, \ldots, k_0)_2$, $P \in E(\mathbb{F}_q)$.
Output: $k \cdot P$.
  $Q = \mathcal{O}$
  for $i = 0$ to $\ell - 1$ do
      if $k_i = 1$ then
          $Q = Q + P$
      end if
      $P = 2P$
  end for
  return $Q$
```

One can see that the scalar is represented in binary form and processed from right-to-left. As there are no assumptions on the form of the scalar k, it is expected that on average half of the ℓ bits are ones. Therefore, the expected running time of the algorithm can be estimated as:

$$\approx \left(\frac{\ell}{2} \; additions + \ell \; doublings \right).$$

As mentioned in Section 3.5, this algorithm is vulnerable to *side channel* attacks.

Double-and-Always-Add Multiplication

The *double-and-always-add* algorithm tries to counter *simple side channel attacks* via so-called *dummy* instructions. Those are instructions that are executed but have no effect on the outcome of the calculation. The author of [Cor99] introduced Algorithm 7, which does exactly this. One point addition and one point doubling is executed per bit k_i of the scalar k and the result is chosen depending on k_i. One of the executed operations is not *effective*.

Algorithm 7 Double-and-Always-Add method for point multiplication [Cor99, Algorithm 1']

```
Input: $k = (k_{\ell-1}, \ldots, k_0)_2$, $P \in E(\mathbb{F}_q)$.
Output: $Q = k \cdot P$.
  $R_0 = P$
  for $i = \ell - 2$ to $0$ do
      $R_0 = 2R_0$
      $R_1 = R_0 + P$
      $R_0 = R_{k_i}$
  end for
  return $R_0$
```

One can see that the execution time of this algorithm does not depend on the format of the scalar k so the algorithm is therefore called *regular*. Nonetheless this comes with a performance penalty, as the estimated running time is:

$$\approx ((\ell - 2) \; additions + (\ell - 2) \; doublings).$$

Montgomery Ladder Multiplication

The so-called *Montgomery ladder* is an algorithm for scalar multiplication. It is based on an idea introduced by Peter Montgomery in [Mon87]. The *Montgomery ladder* scalar multiplication algorithm has two main advantages. First it has very low storage requirements and secondly it executes the same *effective* operations for every bit of the scalar. This has the effect that the execution time of the *Montgomery ladder* does not depend on the format of the scalar k, which means the *Montgomery ladder* is *regular*. Those properties give a good starting point if one aims to create an implementation which is resistant to various implementation attacks (see Section 3.5 for an introduction to implementation attacks). In its simplest form, the *Montgomery ladder* works as shown in Algorithm 8.

Algorithm 8 Montgomery Ladder on Elliptic Curves

```
Input: A point $P$ on $E$ and a scalar $k = (k_{\ell-1}, \dots, k_0)_2 \in \mathbb{Z}_p$.
Output: $k \cdot P$.
  $R_0 = P$ and $R_1 = 2P$
  for $i = \ell - 2$ downto 0 do
      if $k_i = 0$ then
          $R_1 = R_0 + R_1$, and
          $R_0 = 2R_0$
      else
          $R_0 = R_0 + R_1$, and
          $R_1 = 2R_1$
      end if
  end for
  return $R_0$
```

The scalar k of length ℓ is given in *binary* form. For every bit of the scalar, one addition and one doubling are performed; this is called a *Montgomery ladder* step (MLS). It is of interest that the relation between R_1 and R_0 is invariant throughout all ladder steps. It should also be noted that it is possible to parallelize the *Montgomery ladder* algorithm. An obvious way would be to use two processing units, one for additions and the other for doublings, assuming that those two operations take roughly the same time to compute. The expected runtime of Algorithm 8 for a scalar k of length ℓ is given as:

$$\approx ((\ell - 2) \ \textit{additions} + (\ell - 1) \ \textit{doublings}) .$$

The number of additions and doubling stays the same throughout all optimizations of the *Montgomery ladder*. Therefore, the addition and doubling formulas have to be optimized to be more efficient in terms of *field* operations. We refere the reader to Section 4.4 where we discuss several optimizations in detail.

Joye's Double-and-Add Multiplication

Joye's Double-and-Add algorithm has several properties similar to the *Montgomery ladder*, e.g., all its operations are *effective*. This means that no dummy operations are necessary for the algorithm to be *regular*. It should be noted that in *Joye's Double-and-Add multiplication method* the scalar is processed right-to-left, giving advantages as it thwarts for example the idea of the *doubling attack* given in [FV03]. This attack works only if the scalar is processed right-to-left. The authors of [FV03] introduce their concept for a downward Double-and-Add scalar multiplication method, as given in Algorithm 7. They concentrate on the doubling steps, and observe that while computing the scalar multiplication for two values P and $2P$, similar intermediate steps emerge. Namely the steps $k+1$ for P and k for $2P$. If those two steps are similar, one can deduce that a certain bit of the scalar is 0. The attack can retrieve the zero bits in addition-subtraction chains such as the NAF (as given in Section 4.1).

Joyes's Double-and-Add algorithm was introduced for an *additive Abelian group G* by the authors of [Joy07]. Given an arbitrary point $P \in G$ and a scalar k in *binary* representation of length ℓ, the multiplication $Q = kP \in G$ can be calculated as $Q = \sum_{j=0}^{\ell-1}(k_j 2^j)P$. For $0 \leq j < \ell$ two states are defined in [Joy07]. Namely $S_j = \sum_{i=0}^{j} k_i B_i$ (where $B_j = 2^j P$) and $T_j = B_{j+1} - S_j$. These states represent the necessary intermediate results while calculating the scalar multiplication. The authors of [Joy07] give the following formulas for calculating both states:

$$\begin{aligned} S_j &= \sum_{i=0}^{j} k_i B_i = k_j B_j + S_{j-1} = k_j(S_{j-1} + T_{j-1}) + S_{j-1}, \\ &= (1 + k_j)S_{j-1} + k_j T_{j-1}, \end{aligned}$$

and

$$\begin{aligned} T_j &= B_{j+1} - S_j = 2B_j - (k_j B_j + S_{j-1}) = (2 - k_j)B_j - S_{j-1}, \\ &= (2 - k_j)T_{j-1} + (1 - k_j)S_{j-1}. \end{aligned}$$

This is summarized by the authors in [Joy07, Proposition 1], which is stated as:

$$S_j = \begin{cases} S_{j-1} \text{ if } k_j = 0 \\ 2S_{j-1} + T_{j-1} \text{ if } k_j = 1 \end{cases} \quad \text{and} \quad T_j = \begin{cases} S_{j-1} + 2T_{j-1} \text{ if } k_j = 0 \\ T_{j-1} \text{ if } k_j = 1 \end{cases}$$

for all $j \geq 0$ as $Q = kP = S_{\ell-1}$. This leads directly to the so-called *Double-and-Add scalar multiplication algorithm* depicted by Algorithm 9. The algorithm iterates over j bits of the *binary* representation of a scalar k. For every bit k_j the two states R_0 and R_1 are given as: $R_0 = S_j$ and $R_1 = T_j$. Additionally, after the j^{th} iteration the relation of R_0 and R_1 satisfies $R_0 + R_1 = 2^j P$.

Algorithm 9 Joye's Double-and-Add Multiplication Method [Joy07, Algorithm 1]

```
Input: A point $P \in G$ and $k = (k_{\ell-1}, \dots, k_0)_2 \in \mathbb{N}$.
Output: $Q = k \cdot P$.
  $R_0 = \mathcal{O}$ and $R_1 = P$
  for $j = 0$ to $\ell - 1$ do
      $b = 1 - k_j$
      $R_b = 2R_b + R_{k_j}$
  end for
  return $R_0$
```

One can see that Algorithm 9 has an estimated running time of:

$$\approx ((\ell - 1) \text{ } additions + (\ell - 1) \text{ } doublings).$$

Non-Adjacent Form (NAF)

The NAF representation is a so-called signed digit representation. On elliptic curves subtracting a point is (almost) equally expensive to adding a point. One can use this observation for finding a minimal *signed digit representation* $k = \sum_{i=0}^{\ell-1} k_i 2^i$ with $k_i \in \{0, \pm 1\}$ for the scalar k, where no two consecutive digits are non-zero. Representing a scalar in NAF form guarantees for every positive integer k the following properties (as stated in [HMV04, Theorem 3.29]):

- every k gives a unique NAF, denoted $NAF(k)$ of length ℓ,

- $NAF(k)$ is the signed digit representation with the least non-zero digits, compared to all signed digit representations,
- the maximum length ℓ of $NAF(k)$ is given by the length of the binary representation of k plus one,
- the length of ℓ is given as $\frac{2^\ell}{3} < k < \frac{2^{\ell+1}}{3}$,
- the average density of non-zero digits of all NAFs of length ℓ is given with approximately $\frac{1}{3}$.

Encoding a scalar in NAF form can yield performance advantages, for example in the case of Algorithm 6, as it reduces the number of necessary additions significantly. Algorithm 10 shows how the NAF of a scalar k can be computed.

Algorithm 10 Computing the NAF of a positive integer [HMV04, Algorithm 3.30]

```
Input: positive integer $k$.
Output: $NAF(k)$.
1: $i = 0$
2: while $k \geq 1$ do
3:     if $k$ is odd then
4:         $k_i = 2 - (k \ mod\ 4)$
5:         $k = k - k_i$
6:     else
7:         $k_i = 0$
8:     end if
9:     $k = \frac{k}{2}$
10:    $i = i + 1$
11: end while
12: return $(k_{i-1}, k_{i-2}, \ldots, k_1, k_0)$
```

Windowing Methods: The performance of scalar multiplication methods which use NAF representation for the scalar can be further enhanced by so-called *windowing methods*. Solinas introduced the *width-w windowing method* in [Sol00]. Again the scalar is represented in a *signed digit representation*, with $w \geq 2$ and ℓ denoting the length of the *width-w NAF*. The scalar can be written as $k = \sum_{i=0}^{\ell-1} k_i 2^i$ where k_i is odd, $|k_i| < 2^{w-1}$ and $k_{\ell-1} \neq 0$. Among w consecutive digits, at most one is non-zero.

Width-w Non-adjacent Form (NAF_w): Algorithm 11 shows how to compute the *width-w* NAF of a positive integer. As stated in [HMV04, Theorem 3.33], the *width-w* NAF representation guarantees, for every positive integer k, the following properties:

- every k gives a **unique** width-w NAF, denoted $NAF_w(k)$,
- $NAF_2(k) = NAF(k)$,
- the maximum length ℓ of $NAF_w(k)$ is given by the length of the binary representation of k plus one,
- the average density of non-zero digits of all width-w NAF of length ℓ is given with approximately $\frac{1}{w+1}$.

With the *width-w* windowing method, the additions for w bits of the scalar k are processed at the same time. For every zero in the NAF representation, a doubling has to be calculated. Algorithm 11 can be used to efficiently calculate the *width-w NAF* of a scalar.

Algorithm 11 Computing the width-w NAF of a positive integer [HMV04, Algorithm 3.35]

```
Input: Window width $w$, positive integer $k$.
Output: $NAF_w(k)$.
 1: $i = 0$
 2: while $k \geq 1$ do
 3:     if $k$ is odd then
 4:         $k_i = k \; mods \; 2^w$ (Note: $mods$ gives an integer $u$ with $u \equiv k \; (mod \; 2^w)$ and $-2^{w-1} \leq u < 2^{w-1}$)
 5:         $k = k - k_i$
 6:     else
 7:         $k_i = 0$
 8:     end if
 9:     $k = \frac{k}{2}$
10:     $i = i + 1$
11: end while
12: return $(k_{i-1}, k_{i-2}, \dots, k_1, k_0)$
```

The authors of [HMV04] give Algorithm 12 as the *width-w NAF* version of the right-to-left binary method for point multiplication (introduced in Algorithm 6).

Algorithm 12 Window NAF method for point multiplication [HMV04, Algorithm 3.36]

```
Input: Window width $w$, positive integer $k, P \in E(\mathbb{F}_q)$.
Output: $kP$.
 1: Use Algorithm 11 to compute $NAF_w(k) = \sum_{i=0}^{\ell-q} k_i 2^i$
 2: Compute $P_i = iP$ for $i \in \{1, 3, 5, \dots, 2^{w-1} - 1\}$
 3: $Q = \mathcal{O}$
 4: for $i = \ell - 1$ to $0$ do
 5:     $Q = 2Q$
 6:     if $k \neq 0$ then
 7:         if $k_i > 0$ then
 8:             $Q = Q + P_{k_i}$
 9:         else
10:             $Q = Q - P_{-k_i}$
11:         end if
12:     end if
13: end for
14: return $Q$
```

An approximation of the expected running time of Algorithm 12, with $m = \lceil log_2(q) \rceil$ is given as:

$$\approx \underbrace{\left[1 \; doubling \; + (2^{w-2} - 1) \; additions)\right]}_{\text{precomputation costs (Step 2 of Algorithm 12)}} + \left[\frac{m}{w+1} \; additions + m \; doublings\right].$$

4.2 Comb Multiplication

In this section, we present several scalar multiplication methods which use precomputations for a fixed point P, meaning that they precalculate and store values that solely depend on P to speed up a scalar multiplication performed later on. We start with the *fixed-base comb multiplication method* in Section 4.2.1. This concept is extended for the *improved fixed-base comb multiplication method* in Section 4.2.2. This method is also available in a *simultaneous scalar multiplier* version in Section 4.2.4.

4.2.1 Fixed-Base Comb Multiplication Method

The *fixed-base comb multiplication* method represents the scalar k of length ℓ as a *binary matrix* with c columns and r rows. This concept was introduced by Lim and Lee in [LL94], where they give a binary representation for k by splitting k into r blocks K_i, with $0 \leq i \leq r-1$, of equal length $d = \lceil \frac{\ell}{r} \rceil$. The scalar is padded with leading zeros if necessary. Each block K_i is then written as a row in the matrix. The matrix columns c are the base for all further computations. This scalar representation is illustrated in [HMV04] as follows:

$$k = \begin{bmatrix} K_0 \\ \vdots \\ K_i \\ \vdots \\ K_{r-1} \end{bmatrix} = \begin{bmatrix} K_{0,d-1} & \cdots & K_{0,0} \\ \vdots & & \vdots \\ K_{i,d-1} & \cdots & K_{i,0} \\ \vdots & & \vdots \\ K_{r-1,d-1} & \cdots & K_{r-1,0} \end{bmatrix} = \begin{bmatrix} k_{d-1} & \cdots & k_0 \\ \vdots & & \vdots \\ k_{(i+1)d-1} & \cdots & k_{ir} \\ \vdots & & \vdots \\ k_{rd-1} & \cdots & k_{(r-1)d} \end{bmatrix}.$$

To gain a speedup later on, it is necessary to precompute all possible bit permutations for a bitstring $s = (b_{r-1}, \ldots, b_1, b_0)$ of length r. This gives us a lookup table for every possible window value, as:

$$[b_{r-1}, \ldots, b_2, b_1, b_0]P = b_{r-1}2^{(r-1)d}P + \cdots + b_2 2^{2d} + b_1 2^d P + b_0 P.$$

The actual scalar multiplication is computed with Algorithm 13. It is easy to see the table lookup in Step 4:

Algorithm 13 Fixed-base comb method for point multiplication [HMV04, Algorithm 3.44]

```
Input: Window width $r$, $d = \lceil \frac{\ell}{r} \rceil$, $k = (k_{\ell-1}, \ldots, k_1, k_0)_2$, $P \in E(\mathbb{F}_q)$.
Output: $k \cdot P$.
1: Precomputation: compute $[b_{r-1}, \ldots, b_2, b_1, b_0]P$ for all permutations for a bitstring $b$ of length $r$
2: for $i = d-1$ to 0 do
3:     $Q = 2Q$.
4:     $Q = Q + [K_{r-1,i}, \ldots, K_{1,i}, K_{0,i}]P$.
5: end for
6: return $(Q)$
```

As stated in [HMV04], Algorithm 13 has an expected running time of:

$$\approx \left[\left(\frac{2^r - 1}{2^r} d - 1 \right) \textit{additions} + (d-1) \textit{ doublings} \right].$$

It is of interest that the number of precomputations is given as $2^r - 1$. Those precomputation costs amortize only for points fixed a priori, which is indicated by the name of the multiplication method.

4.2.2 Improved Fixed-Base Comb Method for Fast Scalar Multiplication

To fully understand the improvements introduced by Mohammed et al. in [MHH12], it is necessary to discuss the so-called Sakai-Sakurai method for direct doubling beforehand. Mohammed et al. use it to speedup the necessary doublings in their algorithm, which is discussed later on.

Sakai-Sakurai Method for Direct Doubling

As the name suggests, the Sakai-Sakurai method (given in Algorithm 14) gives a way to directly calculate kP (with $P \in E(\mathbb{F}_p)$) from P. It avoids all the intermediate steps of calculating 2^iP for $0 \leq i \leq k-1$. The Sakai-Sakurai method computes 2^iP with costs of $1I$, $4i+1M$, and $4i+1S$ compared to the costs of separate i doublings, given as iI, $2iM$, and $2iS$. Here I denotes a *prime field inversion*, M denotes a *prime field multiplication* and S denotes a *prime field squaring*. The method only works under the assumption that the scalar k is a power of two, meaning that $k = 2^r$ with $r \geq 1$.

Algorithm 14 Sakai-Sakurai method for direct doubling [MHH12, Algorithm 3]

```
Input: A positive integer $r$ such that $k = 2^r$ and $P \in E(\mathbb{F}_q)$.
Output: $k = 2^rP$.
 1: $A_1 = x_1, B_1 = 3x_1^2 + a$ and $C_1 = -y_1$
 2: for $i = 2$ to r do
 3:     $A_i = B_{i-1}^2 - 8A_{i-1}C_{i-1}^2$
 4:     $B_i = 3A_i^2 + 16^{i-1}a(\prod_{j=1}^{i-1} C_j)^4$
 5:     $C_i = -8C_{i-1}^4 - B_{i-1}(A_i - 4A_{i-1}C_{i-1}^2)$
 6: end for
 7: Compute $D_r = 12A_rC_r^2 - B_r^2$
 8: Compute $x_{2^r} = \frac{B_r^2 - 8A_rC_r^2}{(2^r \prod_{i=1}^{r} C_i)^2}$
 9: Compute $y_{2^r} = \frac{8C_r^4 - B_rD_r}{(2^r \prod_{i=1}^{r} C_i)^3}$
10: return $(x_{2^r}, y_{2^r})$
```

In the following section, we will discuss the *improved fixed-base comb method for scalar multiplication* which utilizes the *Sakai-Sakurai method* discussed.

Improved Fixed-Base Comb Method for Fast Scalar Multiplication

In 2012, Mohammed et al. proposed the *improved fixed-base comb method for fast scalar multiplication* [MHH12]. Their work builds on ideas of Lim and Lee [LL94] as well as the improvements introduced 2005 by Tsaur and Chou [TC05]. In their paper, Tsaur and Chou represented the scalar in non-adjacent form (NAF) representation and, to speed up additions, they used the so called Sakai-Sakurai method (as explained in the previous section) for direct doubling. Mohamed et al. represent the scalar k in a so-called width-w non-adjacent form (NAF_w) (as described in Section 4.1) of length ℓ. To create the matrix representation for multiplying, k is split into $a = \lceil \frac{\ell}{w} \rceil$ blocks of size w and if necessary k is padded with zeros. This transforms k into the following form, $k = K_{a-1}||\cdots||K_1||K_0$. To build a $a \times w$ binary matrix every K becomes a column in the matrix, as follows:

$$k = \begin{bmatrix} K_{a-1} \;||\; \cdots \;||\; K_0 \end{bmatrix} = \begin{bmatrix} k_{a-1,0} & \cdots & k_{0,0} \\ \vdots & & \vdots \\ k_{a-1,w-1} & \cdots & k_{0,w-1} \end{bmatrix},$$

then from right-to-left the $a \times w$ matrix is split into $w \times v$ blocks of size $b = \lceil \frac{a}{v} \rceil$ as follows:

$$k = \begin{bmatrix} [k_{a-1,0} & \cdots & k_{a-b,0}] & \cdots & [k_{b-1,0} & \cdots & k_{0,0}] \\ & \vdots & & & & \vdots & \\ [k_{a-1,w-1} & \cdots & k_{a-b,w-1}] & \cdots & [k_{b-1,w-1} & \cdots & k_{0,w-1}] \end{bmatrix}.$$

To increase speed precomputations are necessary, and [MHH12] give the following formulas for all values that need to be precomputed:

$$\begin{aligned} G[0][sd] &= e_{w-1}2^{w-1}P + e_{w-2}2^{w-2}P + \cdots + e_0P, \\ &= sdP, \\ G[j][sd] &= 2^{wb}(G[j-1][sd]), \\ &= 2^{jwb}G[0][sd] = 2^{jwb}sdP, \end{aligned}$$

for all $0 < j \leq v-1$ with $s \in \{1, 2^1, 2^2, 2^3, \ldots, 2^{w-1}\}$ and $d \in \{1, 3, 7, \ldots, 2^{w-1}-1\}$. Furthermore, the lookup table index sd is given by the binary string $e_{w-1} \ldots e_1 e_0$. After precomputing all necessary values, Algorithm 15 can be used to calculate kP. The computational costs for this algorithm are stated in [MHH12] as:

$$\begin{cases} \left[\left(1 - \left(\frac{w}{w+1}\right)^w\right) a\right] \; additions + (b-2)X \text{ for the average case, and} \\ \left[\left(1 - \left(\frac{w-1}{w}\right)^w\right) a\right] \; additions + (b-2)X \text{ for the worst case.} \end{cases}$$

Here X is given as:

$$X = \begin{cases} doublings, \text{ if } w = 1, \text{ and} \\ \text{applications of Algorithm 14 otherwise.} \end{cases}$$

Algorithm 15 Proposed width-w NAF method for scalar multiplication [MHH12, Algorithm 4]

```
Input: Positive integers $w, v, (k = k_{\ell-1}, \ldots, k_1, k_0)_{NAF_w}$ and $P \in E(\mathbb{F}_q)$.
Output: $Q = kP$.
 1: $a = \lceil \frac{\ell}{w} \rceil$ and $b = \lceil \frac{a}{v} \rceil$
 2: Compute $G[0][sd]$ and $G[j][sd]$ for all $s \in \{1, 2, 2^2, 2^3, \ldots, 2^{w-1}\}, 0 < j \leq v-1$ and $d \in$
    $\{1, 3, 7, \ldots, 2^{w-1}-1\}$
 3: $Q = \mathcal{O}$
 4: for $t = b-1$ downto 0 do
 5:     if $w = 1$ then
 6:         $Q = 2Q$
 7:     else
 8:         Use Algorithm 14 to compute $Q = 2^w Q$
 9:     end if
10:     for $j = v-1$ downto 0 do
11:         $I_{j,t} = (k_{jb+t,w-1} \ldots k_{jb+t,0})_{NAF_w}$
12:         if $I_{j,t} > 0$ then
13:             $Q = Q + G[j][I_{j,t}]$
14:         else if $I_{j,t} < 0$ then
15:             $Q = Q - G[j][-I_{j,t}]$
16:         end if
17:     end for
18: end for
19: return $(Q)$
```

4.2.3 Multiple Point Scalar Multiplication

A multiple point scalar multiplication method is designed to calculate scalar multiplications of form $kP + mQ$ with elliptic curve points $P, Q \in E(\mathbb{F}_q)$ and integers $0 \leq k < ord_E(P)$ and $0 \leq m < ord_E(Q)$. Multiplications of this form are used for example in the *Elliptic Curve Digital Signature Algorithm (ECDSA)*. Using for example *Shamir's trick* (see [Sol01] for more information) this can be done faster than two separate multiplications, followed by an addition. One can accomplish this as outlined in [HMV04, Section 3.3.3]. The *binary* representation ℓ of both scalars k and m is written in a $2 \times \ell$ matrix. This is followed by precomputations for a windows of size w it is necessary to compute $iP + jQ$ for all $0 \leq i, j < 2^w$. The result is calculated by adding the precomputed values $iP + jQ$ chosen using the $2 \times w$ bits of the scalar matrix. In total there are $\lceil \frac{\ell}{w} \rceil$ intermediate steps. The discussed method is given in Algorithm 16.

Algorithm 16 Simultaneous multiple point multiplication [HMV04, Algorithm 3.48]

```
Input: Window width $w, k = (k_{\ell-1}, \dots, k_0)_2, m = (m_{\ell-1}, \dots, m_0)_2, P, Q \in E(\mathbb{F}_q)$.
Output: $kP + mQ$.
1: Write $k = (K^{d-1}, \dots, K^1, K^0)$ and $m = (M^{d-1}, \dots, M^1, M^0)$ where each $K^i, M^i$ is a bitstring of
   length $w$, and $d = \lceil \frac{\ell}{w} \rfloor$
2: $R = \mathcal{O}$
3: for $i = d - 1$ downto 0 do
4:     $R = 2^w R$
5:     $R = R + (K^i P + M^i Q)$
6: end for
7: return $(R)$
```

An approximation of the expected running time of Algorithm 16 is given in equation [HMV04, Equation 3.30] as:

$$\approx \underbrace{\left[(3 \cdot 2^{2(w-1)} - 2^{w-1} - 1)\ additions + (2^{2(w-1)} - 2^{w-1})\ doublings\right]}_{\text{precomputation costs}}$$
$$+ \left[\left(\frac{2^{2(w-1)}}{2^{2w}} d - 1\right)\ additions + (d-1)w\ doublings\right].$$

4.2.4 Multiple Point Improved Fixed-Base Comb Method for Fast Scalar Multiplication

The multiple point scalar multiplication method proposed in [MHH12] works very similarly to the method described in Section 4.2.2 and assumes that both scalars are in NAF_w form and of length ℓ. Analogous to the steps in Section 4.2.2, each of the two scalars is split into $a = \lceil \frac{\ell}{w} \rceil$ blocks of size w. This gives a representation as follows (all formulas in this section are taken from [MHH12]):

$$k = K_{a-1} || \dots || K_0 = \sum_{d=0}^{a-1} K_d 2^{dw} \text{ and } r = R_{a-1} || \dots || R_0 = \sum_{d=0}^{a-1} R_d 2^{dw}.$$

Then from right-to-left the $a \times w$ matrices are split into $w \times v$ blocks of size $b = \lceil \frac{a}{v} \rceil$ as given by the following formulas:

$$kP = \sum_{t=0}^{b-1} 2^{tw} \sum_{j=0}^{v-1} K_{jb+t} 2^{jbw} P \text{ and } rQ = \sum_{t=0}^{b-1} 2^{tw} \sum_{j=0}^{v-1} R_{jb+t} 2^{jbw} Q. \tag{4.1}$$

Similarly to Section 4.2.2, the blocks K_{jb+t} and R_{jb+t} of the scalars are in NAF_w representation, namely $[k_{jb+t,w-1} \dots k_{jb+t,0}]$ and $[r_{jb+t,w-1} \dots r_{jb+t,0}]$. Therefore, the formulas in Equation (4.1) can be combined to the following formula:

$$kP + rQ = \sum_{t=0}^{b-1} 2^{tw} \sum_{j=0}^{v-1} (K_{jb+t} 2^{jbw} P + R_{jb+t} 2^{jbw} Q). \tag{4.2}$$

To gain a speedup, in [MHH12] the following formulas for values that need to be precomputed, are given:

$$\begin{aligned}
G_p[0][sd] &= e_{w-1} 2^{w-1} P + e_{w-2} 2^{w-2} P + \dots + e_0 P, \\
&= sdP, \\
G_p[j][sd] &= 2^{wb}(G_p[j-1][sd]), \\
&= 2^{jwb} G_p[0][sd] = 2^{jwb} sdP, \\
G_q[0][sd] &= e_{w-1} 2^{w-1} Q + e_{w-2} 2^{w-2} Q + \dots + e_0 Q, \\
&= sdQ, \\
G_q[j][sd] &= 2^{wb}(G_q[j-1][sd]), \\
&= 2^{jwb} G_q[0][sd] = 2^{jwb} sdQ,
\end{aligned}$$

for all $0 < j \leq v - 1$ with $s \in \{1, 2^1, 2^2, 2^3, \dots, 2^{w-1}\}$ and $d \in \{1, 3, 7, \dots, 2^{w-1} - 1\}$. Furthermore, the index sd is given by the binary string $e_{w-1} \dots e_1 e_0$. This allows us to rewrite Equation (4.2) as

$$kP + rQ = \sum_{t=0}^{b-1} 2^{tw} \sum_{j=0}^{v-1} (G_p[j][M_{j,t}] + G_q[j][N_{j,t}]),$$

where $0 \leq t < b$, $M_{j,t} = [k_{jb+t,w-1} \dots k_{jb+t,0}]$ and $N_{j,t} = [r_{jb+t,w-1} \dots r_{jb+t,0}]$. After precomputing all necessary values, Algorithm 17 can be used to calculate $a \cdot P + b \cdot Q$. The expected computational costs for this algorithm are stated (in [MHH12]) as:

$$2 \left[\left(1 - \left(\frac{w}{w+1} \right)^w \right) a \right] \textit{ additions } + (b-2)X \text{ for the average case.}$$

Here X is given as:

$$X = \begin{cases} \textit{doublings}, \text{ if } w = 1, \text{ and} \\ \text{applications of Algorithm 14 otherwise.} \end{cases}$$

Algorithm 17 Proposed width-w NAF method for multiple scalar multiplication [MHH12, Algorithm 5]

```
Input: Positive integers $w, v, P, Q \in E(\mathbb{F}_q), k = (k_{\ell-1}, \dots, k_1, k_0)_{NAF_w}, r = (r_{\ell-1}, \dots, r_1, r_0)_{NAF_w}$.
Output: $kP + rQ$.
 1: $a = \lceil \frac{\ell}{w} \rceil$ and $b = \lceil \frac{a}{v} \rceil$
 2: Compute $G_p[0][sd]$ and $G_p[j][sd]$ for all $s \in \{1, 2, 2^2, 2^3, \dots, 2^{w-1}\}, 0 < j \leq v-1$ and $d \in$
    $\{1, 3, 7, \dots, 2^{w-1} - 1\}$
 3: Compute $G_q[0][sd]$ and $G_q[j][sd]$ for all $s \in \{1, 2, 2^2, 2^3, \dots, 2^{w-1}\}, 0 < j \leq v-1$ and $d \in$
    $\{1, 3, 7, \dots, 2^{w-1} - 1\}$
 4: $R = \mathcal{O}$
 5: for $t = b-1$ downto $0$ do
 6:     if $w = 1$ then
 7:         $R = 2R$
 8:     else
 9:         Use Algorithm 14 to compute $R = 2^w R$
10:     end if
11:     for $j = v-1$ downto $0$ do
12:         $M_{j,t} = (k_{jb+t,w-1} \dots k_{jb+t,0})_{NAF_w}$
13:         if $M_{j,t} > 0$ then
14:             $R = R + G_p[j][M_{j,t}]$
15:         else if $M_{j,t} < 0$ then
16:             $R = R - G_p[j][-M_{j,t}]$
17:         end if
18:     end for
19:     for $j = v-1$ downto $0$ do
20:         $N_{j,t} = (k_{jb+t,w-1} \dots k_{jb+t,0})_{NAF_w}$
21:         if $N_{j,t} > 0$ then
22:             $R = R + G_q[j][N_{j,t}]$
23:         else if $N_{j,t} < 0$ then
24:             $R = R - G_q[j][-N_{j,t}]$
25:         end if
26:     end for
27: end for
28: return $(R)$
```

4.3 Scalar Multiplication using Efficiently Computable Endomorphisms

The scalar multiplication operation on elliptic curves can be accelerated, given that there is an *efficiently computable endomorphism* available on the respective curve. The concepts given in this section are related to the special arithmetic on *Koblitz curves* (see [Kob91] and [Sol00]). They are not as powerful, but work on a larger class of elliptic curves. This section is a summarization of the ideas given in [GLV01].

Endomorphisms

Given a *finite* field $\mathbb{F}_q$ and an elliptic curve $E(\mathbb{F}_q)$, an *endomorphism* ϕ is a mapping $\phi : E \to E$ given by a pair of *rational functions* g and h such that for all $P \in E : \phi(P) = (g(P), h(P))$ and $\phi(\mathcal{O}) = \mathcal{O}$. Furthermore, all coefficients of g and h have to lie in $\mathbb{F}_q$ and ϕ is a *group homomorphism* (as explained in Section 2.1.3) for the *Abelian group* defined by $E(\mathbb{F}_q)$.

Using Efficient Endomorphisms

The authors of [GLV01] state the idea that decomposing the scalar for the scalar multiplication can yield a significant increase in speed. Given a field $\mathbb{F}_q$, an elliptic curve $E(\mathbb{F}_q)$, a point $P \in E(\mathbb{F}_q)$ of *prime* order n, and an endomorphism ϕ on $E(\mathbb{F}_q)$. For the decomposition to work, the *characteristic polynomial* of ϕ has to have a root λ modulo n. The endomorphism ϕ works on $\langle P \rangle$ as a *multiplication map* $[\lambda] : P \mapsto \lambda P$, meaning that $\phi(P) = \lambda P$. The method given in [GLV01] yields a performance improvement if the costs of calculating ϕ are smaller than computing approximately $\frac{log_2\ (n)}{3}$ point doublings. The main idea in [GLV01] is to represent the scalar $1 \leq k \leq n-1$ as $k = k_1 + k_2\lambda$ with $k_1, k_2 \in \{0, \ldots, \lceil\sqrt{n}\rceil\}$. A scalar point multiplication can then be stated with the following equation (taken from [GLV01, Equation 6]):

$$kP = (k_1 + k_2\lambda)P \ = \ k_1P + k_2(\lambda P) \ = \ k_1P + k_2\phi(P).$$

Given that k_1 and k_2 roughly have half the bit length of k, this decomposition makes it possible to use a variety of *simultaneous* or *interleaving scalar multiplication* algorithms to obtain speedups. The scalar decomposition builds on the following train of thought, outlined in [GLV01]. Given $G = \mathbb{Z}\times\mathbb{Z}$ and a *homomorphism* $f : G \to \mathbb{Z}_n$ where $f : (i, j) \mapsto (i + j\lambda)\ mod\ n$, then the problem of finding two integers k_1, k_2 which are both small can also be expressed as finding a vector $(k_1, k_2) \in \mathbb{Z} \times \mathbb{Z}$ with a small *Euclidean norm.* The authors of [GLV01] show how to find two *linearly independent, short vectors*, $v_1, v_2 \in G$ where $f(v_1) = f(v_2) = 0$ by applying the *extended Euclidean algorithm* (Algorithm 18) to λ and n.

Algorithm 18 Extended Euclidean algorithm for integers [HMV04, Algorithm 2.19]

```
Input: Positive Integers $a$ and $b$ with $a \leq b$.
Output: $d = gcd(a, b)$ and integers $x, y$ satisfying $ax + by = d$.
1: $u = a, v = b$
2: $x_1 = 1, y_1 = 0, x_2 = 0, y_2 = 1$
3: while $u \neq 0$ do
4:     $q = \lfloor \frac{v}{u} \rfloor, r = v - qu, x = x_2 - qx_1, y = y_2 - qy_1$
5:     $v = u, u = r, x_2 = x_1, x_1 = x, y_2 = y_1, y_1 = y$
6: end while
7: $d = v, x = x_2, y = y_2$
8: return $(d, x, y)$
```

The resulting vectors v_1, v_2 generate an integer lattice that contains a vector v that is close to $(k, 0)$. By rewriting v_1, v_2 and $(k, 0)$ as vectors in $\mathbb{Q} \times \mathbb{Q}$ the authors of [GLV01] give $(k, 0) = \beta_1 v_1 + \beta_2 v_2$ with $\beta_1, \beta_2 \in \mathbb{Q}$. By rounding $b_1 = \lceil \beta_1 \rfloor$ and $b_2 = \lceil \beta_2 \rfloor$, v can be stated as $v = b_1 v_1 + b_2 v_2$. To accomplish the described scalar decomposition, the authors of [HMV04] give Algorithm 19, which utilizes Algorithm 18 in a precomputation step.

Algorithm 19 Balanced length-two representation of a multiplier [HMV04, Algorithm 3.74]

```
**Input:** Integers $n, \lambda, k \in [0, n-1]$.
**Output:** Integers $k_1, k_2$ such that $k \equiv k_1 + k_2\lambda$ $(mod\ n)$ and $|k_1|, |k_2| \approx \sqrt{n}$.
1: Run the extended Euclidean algorithm (Algorithm 18) with inputs $n$ and $\lambda$ The algorithm produces
   a sequence of equations $s_i n + t_i \lambda = r_i$ where $s_0 = 1, t_0 = 0, r_0 = n, s_1 = 0, t_1 = 1, r_1 = \lambda$, and the
   remainders $r_i$ are non-negative and strictly decreasing. Let $\ell$ be the greatest index for which $r_\ell \geq \sqrt{n}$
2: Set $(a_1, b_1) = (r_{\ell+1}, -t_{\ell+1})$
3: **if** $(r_\ell^2 + t_\ell^2) \leq (r_{\ell+2}^2, -t_{\ell+2}^2)$ **then**
4:     $(a_2, b_2) = (r_\ell, -t_\ell)$.
5: **else**
6:     $(a_2, b_2) = (r_{\ell+2}, -t_{\ell+2})$
7: **end if**
8: Compute $c_1 = \lfloor \frac{b_2 k}{n} \rceil$ and $c_2 = \lfloor \frac{-b_1 k}{n} \rceil$
9: Compute $k_1 = k - c_1 a_1 - c_2 a_2$ and $k_2 = -c_1 b_1 - c_2 b_2$
10: **return** $(k_1, k_2)$
```

Given a decomposed scalar $k = k_1 + k_2\lambda\ mod\ n$ and a suitable endomorphism ϕ, Algorithm 20 calculates kP for a point $P \in E(\mathbb{F}_q)$ by interleaving $k_1 P + k_2\phi(P)$. The expected runtime is given in [HMV04, Equation 3.38] as follows:

$$\approx \left[|\{j : w_j > 2\}|\ doublings + \sum_{j=1}^{2} (2^{w_j - 2} - 1)\ additions + C_k + C_\phi \right] + \left[doublings + \sum_{j=1}^{2} \frac{1}{w_j + 1}\ additions \right] \frac{t}{2},$$

where C_k denotes the costs of decomposing the scalar k, t is the bitlength of n, k_j is given in width-w_j NAF and C_ϕ are the costs of finding a suitable homomorphism. The storage requirements are stated in [HMV04] as $2^{w_1-2} + 2^{w_2-2}$ points. With proper precalculations, it is possible to avoid most of the costs C_k, as v_1 and v_2 do not depend on k, and therefore the estimates $v_1 = \frac{b_1}{n}$ and $v_2 = \frac{-b_2}{n}$ in Algorithm 19 can be used.

Algorithm 20 Point multiplication with efficiently computable endomorphisms [HMV04, Algorithm 3.77]

```
Input: Integers $k \in [0, n-1], P \in E(\mathbb{F}_q)$, window width $w_1$ and $w_2$, and $\lambda$.
Output: $kP$.
 1: Use Algorithm 10 to find $k_1$ and $k_2$ such that $k = k_1 + k_2\lambda \bmod n$
 2: Calculate $P_1 = \phi(P)$, and let $P_1 = P$
 3: Use Algorithm 18 to compute $NAF_{w_j}(|k_j|) = \sum_{i=0}^{\ell_j-1} k_{j,i}2^i$ for $j = 1, 2$
 4: Let $\ell = max\{\ell_1, \ell_2\}$ and define $k_{j,i} = 0$ for $\ell_j \leq i < \ell$, $1 \leq j \leq 2$
 5: If $k_j < 0$, the set $k_{j,i} = -k_{j,i}$ for $0 \leq i < \ell_j, 1 \leq j \leq 2$
 6: Compute $iP_j$ for $i \in \{1, 3, \ldots, 2^{w_j-1} - 1\}, 1 \leq j \leq 2$
 7: $Q = \mathcal{O}$
 8: for $i = \ell - 1$ downto 0 do
 9:     if $k_{j,i} \neq 0$ then
10:         if $k_{j,i} > 0$ then
11:             $Q = Q + k_{j,i}P_j$
12:         else
13:             $Q = Q - |k_{j,i}|P_j$
14:         end if
15:     end if
16: end for
17: return $Q$
```

4.4 Montgomery Ladder Multiplication Methods

In this section, we explain several improvements to the *Montgomery ladder* which were introduced over the last few years. All discussed improvements preserve the desirable properties of the *Montgomery ladder*, i.e., low storage requirements and regularity. As already mentioned in Section 4.1, the number of additions and doublings executed by the *Montgomery ladder* is fixed. Therefore all improvements in the *Montgomery ladder* target the addition and doubling formulas which are optimized in terms of *field* operations.

The standard *Montgomery ladder* was already discussed in Section 4.1. We start by introducing an improved *Montgomery ladder* in Section 4.4.1 and an improved *Co-Z coordinate* version of it in Section 4.4.2. A further improved version with *Co-Z coordinates* and *differential XZ formulas* is given in Section 4.4.4. In Section 4.4.5, a fast *Montgomery ladder* on *Huff curves* is introduced.

4.4.1 Differential Montgomery Ladder Multiplication

López and Dahab further extended the concept of the *Montgomery ladder* (as discussed in Section 4.1) in [LD99]. One of their optimizations was the introduction of the *differential Montgomery ladder*. López and Dahab state in [LD99, Lemma 2] that on a *binary* elliptic curve, the x-coordinate of the sum of two affine points $P_1 = (x_1, y_1), P_2 = (x_2, y_2)$, denoted by x_3, can be computed given the x-coordinates of P_1 and P_2 as well as the x and y-coordinate of their difference $\Delta P = (P_2 - P_1) = (x, y)$. This concept is called *differential addition* or *differential doubling*, and [LD99] gives the explicit formula as follows:

$$x_3 = \begin{cases} x + \left(\frac{x_1}{x_1+x_2}\right)^2 + \frac{x_1}{x_1+x_2} & \text{if } P_1 \neq P_2, \textit{ (differential addition)} \text{ and} \\ x_1^2 + \frac{a_6}{x_1^2} & \text{if } P_1 = P_2 \textit{ (differential doubling).} \end{cases}$$

To speed up the *Montgomery ladder* these two observations, namely the invariant relation between the ladder steps as well as the *differential addition/doubling*, are combined. As only the x-coordinate is calculated in the intermediate ladder steps, the y-coordinate needs to be recovered in an efficient manner at the end of the algorithm. Again López and Dahab state in [LD99] an explicit formula, namely: $y_1 = \frac{(x_1+x)\{(x_1+x)(x_2+x)+x^2+y\}}{x} + y$. Using these *differential* formulas, several field multiplications are saved per bit of scalar k which lessens the computational costs considerably. Furthermore, the y-coordinate recovery costs are one-time-only costs. *Differential Montgomery ladder* implementations use so-called *projective* XZ coordinates as only the x- and z-coordinate are necessary to calculate all intermediate steps. Currently the costs for the fastest formulas (according to [BL14b]) for calculating a *binary projective XZ coordinate Montgomery ladder step (MLS)* are:

$$5M + 4S + 1M_{\sqrt{a_6}}.$$

Here M denotes a *binary field multiplication* and S a *binary field squaring*. $1M_{\sqrt{a_6}}$ denotes a *binary field multiplication* with the curve parameter a_6. The *Montgomery ladder step* is executed once per bit of the scalar k, and consists of one point addition plus one point doubling. The additional one-time costs for the y-coordinate recovery (assuming P is scaled) are given as (taken from [LD98]):

$$1I + 10M + 1S.$$

Here I denotes a *binary field inversion*, M denotes a *binary field multiplication* and S denotes a *binary field squaring*.

4.4.2 Montgomery Ladder Multiplication with Co-Z Coordinates

The concept of Co-Z coordinates was introduced by Meloni in [Mel07]. It works on *projective* coordinates and is founded on the observation that points can be added more efficiently if they share a common Z-coordinate. Meloni introduced his formulas for *Jacobian coordinates* (explained in more detail in Section 3.1) under the following assumptions: E is an elliptic curve over a field $\mathbb{F}$, with $char(\mathbb{F}) \geq 3$, and $P_1 = (X_1 : Y_1 : Z)$ and $P_2 = (X_2 : Y_2 : Z)$ share the same Z-coordinate. Given these conditions [Mel07] states the formula for a point addition $P_1 + P_2 = P_3 = (X_3 : Y_3 : Z_3)$ as follows:

$$\begin{aligned} X_3 &= (Y_2 - Y_1)^2 - X_2(X_2 - X_1)^2 - X_1(X_2 - X_1)^2, \\ Y_3 &= (Y_2 - Y_1)[X_1(X_2 - X_1)^2 - X_3] - Y_1(X_1 - X_1)^3, \\ Z_3 &= Z(X_2 - X_1). \end{aligned} \tag{4.3}$$

This formula has the advantage that an alternative representation P_1' for the point P_1, which has the same Z-coordinate as P_3, is calculated. This is done without any additional computation costs. A combination of intermediate values of Equation (4.3) is used to express P_1'. P_1' is given as: $P_1' = (X_1(X_1 - X_2)^2 : Y_1(X_1 - X_2)^3 : Z_3) \sim P_1$. This makes it possible to use the newly calculated point P_3 and the point P_1 as new input to the addition formula. In [GJM10] Goundar et al. took this concept further and developed a so-called *conjugate Co-Z addition*, applicable to prime fields, which enables the use of *Co-Z formulas* for binary scalar multiplication methods. The authors of [GJM10] give the costs of one *Montgomery ladder* step (MLS), with their *Co-Z* formulas as:

$$9M + 7S.$$

Here, M denotes a *prime field multiplication* and S denotes a *prime field squaring*.

4.4.3 Montgomery Ladder Multiplication with XY-only Co-Z Coordinates

The authors of [VD10] discovered that when using *Co-Z* formulas it is possible to calculate several consecutive additions without the necessity to calculate the z-coordinate of intermediate results. To recover the *affine* coordinates of the final result, it is necessary to recover the z-coordinate of the last step of the *XY-only Co-Z coordinate Montgomery ladder*. We implemented the *XY-only Co-Z coordinate Montgomery ladder* as stated in Algorithm 21.

Algorithm 21 *Montgomery ladder* with *XY-only Co-Z doubling addition* [Riv11, Algorithm 10]

```
Input: $P \in E(\mathbb{F}_q)$, $k = (k_{\ell-1}, \dots, k_, k_0)_2 \in \mathbb{N}$ with $k_{\ell-1} = 1$.
Output: $Q = k \cdot P$.
  $(R_1, R_0) = \text{XYCZ-IDBL}(P)$
  $b = k_{\ell-2}$
  $(R_{1-b}, R_b) = \text{XYCZ-ADDC}(R_b, R_{1-b})$
  for $i = \ell - 2$ to $1$ do
      $b = k_i$
      $d = k_{i-1}$
      $s = d \text{ xor } b$
      $(R_{1-d}, R_d) = \text{XYCZ-DA}(R_{1-b}, R_b)$
      $R_d = (-1)^s R_d$
  end for
  $b = k_0$
  $\lambda = \text{FinalInvZ}(R_0, R_1, P, b)$
  $(R_b, R_{1-b}) = \text{XYCZ-ADD}(R_{1-b}, R_b)$
  return $(X_0\lambda^2, Y_0\lambda^3)$
```

The authors of [Riv11] give several highly specialized operations used in 21.

1. Initial *doubling with Co-Z update* operation, denoted XYCZ-IDBL (given in [Riv11, Algorithm 23]).
2. The *XY-only Co-Z conjugate addtion*, denoted XYCZ-ADDC (given in [Riv11, Algorithm 20]).
3. The *XY-only Co-Z doubling-addition with update*, denoted XYCZ-DA (given in [Riv11, Algorithm 21]).
4. The *coordinate recovery*, denoted FinalInvZ (given in [Riv11, Algorithm 22]).
5. The *XY-only Co-Z addition with update*, denoted XYCZ-ADD (given in [Riv11, Algorithm 18]).

For a detailed explanation of the implementation and inner workings of these operations, please consult [Riv11]. The authors of [Riv11] give the costs of one *Montgomery ladder* step (MLS) with *XY-only Co-Z* formulas as:

$$8M + 6S.$$

The additional one-time costs for calculating the *affine* coordinates of the results are given as (taken from [Riv11]):

$$1I + 18M + 10S.$$

Here, I denotes a *prime field inversion*, M denotes a *prime field multiplication* and S denotes a *prime field squaring.*

4.4.4 Montgomery Ladder Multiplication with XZ-only Co-Z Coordinates

Finally, Hutter et al. proposed an *x-coordinate only (differential) Co-Z Montgomery ladder* in [HJS11]. It can be applied to elliptic curves over a field $\mathbb{F}$, with $char(\mathbb{F}) \neq 2, 3$. All their formulas use *homogeneous projective coordinates* and the following setting is assumed: E is an elliptic curve over a field $\mathbb{F}$, with $char(\mathbb{F}) \neq 2, 3$, $P_1 = (X_1 : Y_1 : Z)$ and $P_2 = (X_2 : Y_2 : Z)$ share the same Z-coordinate and P_D denotes the *affine* difference $\Delta P_D = P_2 - P_1 = (x, y)$. Their *differential addition formula*, which calculates $P_3 = P_1 + P_2$ is given as:

$$\begin{aligned} X_3 &= 2(X_1 + X_2)(X_1X_2 + a_2Z^2) + 4a_6Z^3 - xZ(X_1 - X_2)^2, \\ Z_3 &= Z(X_1 - X_2)^2. \end{aligned}$$

Additionally, they state their *differential doubling formula* as follows:

$$\begin{aligned} X_4 &= (X_2^2 - a_2Z^2)^2 - 8a_6Z^3X_2, \\ Z_4 &= Z[4X_2(X_2^2 + a_2Z^2) + 4a_6Z^3]. \end{aligned}$$

In order to be usable in a *Montgomery ladder*, the x-coordinate of the results $R_0 = (X_3 : Z_3)$ and $R_1 = (X_4 : Z_4)$ need to have a shared Z-coordinate. This is achieved by using two equivalent representations $R_0 \cong (X_3Z_4 : Z_3Z_4)$ and $R_1 \cong (X_4Z_3 : Z_3Z_4)$. The authors of [HJS11] further optimized their formulas for the *Montgomery ladder* by combining the *differential addition* with the *differential doubling* to the following equation:

$$\begin{aligned} X_1' &= V[(X_1 + X_2)(X_1^2 + X_2^2 - U + 2a_4Z^2) + 4a_6Z^3 - xZU], \\ X_2' &= U[(X_2^2 - a_4Z^2)^2 - 8a_6Z^3X_2], \\ Z' &= UVZ, \end{aligned}$$

where $U = (X_1 - X_2)^2$ and $V = X_2(X_2^2 + a_4Z^2) + 4a_6Z^3$. Furthermore, $R_0 = (X_1' : Z')$ and $R_1 = (X_2' : Z')$. This formulas for a *XZ-only Co-Z Montgomery ladder* step (MLS) can be evaluated with costs given as:

$$9M + 5S + 1M_{a_4} + 1M_{4a_6}.$$

Here, M denotes a *prime field multiplication* and S denotes a *prime field squaring*, $1M_{a_4}$ denotes the costs of multiplying with the curve parameter a_4, similarly $4a_6$ denotes the costs of multiplying $4a_6$. After all *Montgomery ladder steps*, the y-coordinate of the result needs to be recovered as only the x-coordinate is calculated. To achieve this, the authors of [HJS11] give the following equation:

$$\begin{aligned} X_1' &= DX_1A, \\ Y_1' &= 2[(CX_1 + a_4A)(C + X_1) - X_2(C - X_1)^2] + 4a_6B, \\ Z' &= DB, \end{aligned}$$

with $A = Z^2, B = ZA, C = xZ$ and $D = 4y$. The given y-coordinate recovery formula has costs of:

$$\begin{aligned} &8M + 2S + 1M_{a_4} + 1M_{4a_6}, \text{ plus} \\ &1I + 2M \text{ if } \textit{affine coordinates} \text{ are needed.} \end{aligned}$$

Here, I denotes a *prime field inversion*, M denotes a *prime field multiplication* and S denotes a *prime field squaring*, $1M_{a_4}$ denotes the costs of multiplying with the curve parameter a_4, similarly $4a_6$ denotes the costs of multiplying $4a_6$.

4.4.5 Differential Montgomery Ladder Multiplication on Huff Curves

To be able to give a *differential Montgomery ladder* on *Huff curves*, it is necessary to first introduce the *differential addition and doubling* operation on *Huff curves.*

Differential Addition and Doubling: The idea of *differential* addition and doubling was already introduced in Section 4.4.1. It works similarly for *Huff curves.* López and Dahab showed in [LD99] how to efficiently recover the y-coordinate of the result on binary curves. Devigne and Joye used these concepts in [DJ11] as follows. Given a coordinate function ω, as defined in Equation (3.6), and two points $\omega(P), \omega(Q) \in H(E)$, *differential addition* describes the concept of using a known difference $\omega(Q-P)$ to speed up the calculation of $\omega(P+Q)$. They give their *differential* formulas, with $P \neq Q, \omega_1 = \omega(P), \omega_2 = \omega(Q)$ and $\bar{\omega} = \omega(Q-P)$, for *differential* doubling as:

$$\omega(2P) = \begin{cases} \frac{\gamma \cdot w_1^2}{(1+w_1)^4} & \text{if } \omega_1 \neq 1, \text{ with } \gamma = \frac{(a+b)^2}{ab},\ \omega_1 = \omega(P), \\ (1:0) & \text{if } \omega_1 = 1. \end{cases}$$

and their *differential* addition works as:

$$\omega(P+Q) = \begin{cases} \frac{(\omega_1+\omega_2)^2}{\bar{\omega}\cdot(1+\omega_1\omega_2)^2} & \text{if } \omega_1\omega_2 \neq 1, \text{ with } \gamma = \frac{(a+b)^2}{ab} \\ (1:0) & \text{if } \omega_1\omega_2 = 1. \end{cases}$$

The idea of *differential* addition and *differential* doubling is heavily utilized when calculating a so-called *differential Montgomery ladder*, as shown in the next section.

Differential Montgomery Ladder: The *projective Montgomery ladder* on Huff curves uses *WZ coordinates* as described in Section 3.3 and the *projective* versions of the *differential addition* and *differential doubling* formulas stated in the previous section. The *projective differential doubling* formula, as given by [DJ11], is stated as follows:

$$\begin{aligned} W(2P) =& \gamma(W_1 Z_1)^2, \\ Z(2P) =& (W_1 + Z_1)^4, \end{aligned} \tag{4.4}$$

with $\gamma = \frac{(a+b)^2}{ab}$ and $P = (W_1 : Z_1)$. Additionally [DJ11] states the *projective differential addition* formula as follows:

$$\begin{aligned} W(P+Q) =& \bar{Z}(W_1 Z_2 + W_2 Z_1)^2, \\ Z(P+Q) =& \bar{W}(W_1 W_2 + Z_1 Z_2)^2, \end{aligned}$$

where $P \neq Q$, $P = (W_1 : Z_1)$ and $Q = (W_2 : Z_2)$, and $W(P-Q) = (\bar{W}, \bar{Z})$. As there are no standardized Huff curves, the practical use of Huff curves involves a mapping of points from a given Weierstrass curve to the corresponding Huff curve, and later on mapping the result on a Huff curve back to the corresponding Weierstrass curve. Fast and efficient formulas to accomplish the mapping, as well as a simultaneous y-coordinate recovery of the

Montgomery ladder's result, are given in our paper in [GH13]. The formula for mapping an *affine* point on a Weierstrass curve, denoted as $P_W = (u, v)$, to the corresponding point in *WZ coordinates* on a Huff curve, denoted as $P_H = (W : Z)$, is stated as:

$$(W : Z) = \left(ab\left(\mu^2 u + a^2\right)\left(\mu^2 u + b^2\right) : \mu^6\left(v + su + \sqrt{a_6}\right)\left(\left(v + su + \sqrt{a_6}\right) + u\right)\right).$$

Where $\mu = (a + b)f$, and a, b, f and s are so-called Huff curve parameters. We refere the reader to Section 3.3 for a more detailed explanation of those. The mapping of a point $P_H = (W : Z)$, in *WZ coordinates*, on a Huff curve to the corresponding *affine* point $P_W = (u, v)$ on a Weierstrass curve (including the y-coordinate recovery), takes multiple steps and is stated as:

$$\begin{aligned} U_1 &= \delta Z_1 u W_1 W_2, \\ V_1 &= \beta\left(\beta\left(\delta Z_2 + uW_2\right) + \left(u^2 + v\right) W_1 W_2\right) + vuW_1^2 W_2, \end{aligned}$$

with $\delta = \frac{ab}{\mu^2}$ and $\beta = \delta Z_1 + uW_1$. The *affine* point $P_W = (u, v)$ is finally calculated as:

$$(u_1, v_1) = \left(\frac{U_1}{uW_1^2 W_2}, \frac{V_1}{uW_1^2 W_2}\right).$$

The authors of [DJ11] state the costs of a *projective differential addition and doubling* with WZ-coordinates as:

$$\textit{differential}\text{ doubling: } 1M + 1D,$$
$$\textit{differential}\text{ addition: } 5M.$$

Where M denotes the cost of a *binary field multiplication* and D denotes the cost of a *binary field multiplication* with γ (given in Equation (4.4)). If used in a *Montgomery ladder*, the constant difference of $W(P - Q)$ can be scaled meaning that $\bar{Z} = 1$. This reduces the costs for a *differential Montgomery ladder step* (MLS) to:

$$5M + 1D,$$

It is important to note that additional one-time costs not only for the y-coordinate recovery but also for mapping between the Weierstrass and Huff curve arise. Those costs are given as (taken from [GH13]):

$$\text{Weierstrass to Huff curve: } 2M + 4M_8,$$
$$\text{Huff to Weierstrass curve + } y\text{-coordinate recovery: } 1I + 6M + 5M_4 + 2M_8 + 1S.$$

Here, I denotes a *binary field inversion*, M denotes a *binary field multiplication*, S denotes a *binary field squaring*, M_4 and M_8 denote a *binary field multiplication* with already existing *multiplication tables* with window size 4 and 8 respectively. Those are used by a precomputational scalar multiplication method for binary field elements, which is attributed to Lim and Lee; for more details see Section 6.2.

4.5 Improved Joye's Double-and-Add Multiplication Method

In this section, we show that *Joye's Double-and-Add method* also benefited from some already mentioned improvements. The authors of [GJM10] improved their implementation of Algorithm 9 by introducing a so-called *Co-Z point double-add with update (ZDAU)* operation which takes advantage of Co-Z formulas. The *ZDAU* formula is comprised of the *Co-Z* coordinate idea, as well as merging the addition and doubling steps into one formula. This enables the implementation of Algorithm 9 as shown in Algorithm 22 (taken from [GJM10]).

Algorithm 22 Joye's Double-and-Add Multiplication Method with Co-Z Addition Formulas

```
Input: A point $P \in E(\mathbb{F}_q)$ and $k = (k_{\ell-1}, \ldots, k_0)_2 \in \mathbb{N}$ with $k_0 = 1$.
Output: $Q = k \cdot P$.
  $b = k_1, R_b = P$ and $(R_{1-b}, R_b) = TPLU(R_b)$
  for $i = 2$ to $\ell - 1$ do
      $b = k_i$
      $(R_{1-b}, R_b) = ZDAU(R_{1-b}, R_b)$
  end for
  return $R_0$
```

Here, the initial *tripling* operation, denoted as $TPLU$, is given by the evaluation of $P = P + 2P$ in *Co-Z arithmetic.* The point tripling is achieved via a *Co-Z* doubling (DBL) that gives $(2P, \tilde{P})$ from $DBL(P)$ followed by a *Co-Z* addition ($ZADDU$) with parameters $(\tilde{P}, 2P)$. For a detailed explanation of the implementation and inner workings of the *ZDAU, TPLU, ZADDU* as well as the *DBL* operation, please consult [GJM10]. The authors of [GJM10] give the costs per bit of the scalar with their *Co-Z* formulas for each *Co-Z Double-and-Add step* as:

$$9M + 7S.$$

Where M denotes a *prime field multiplication* and S denotes a *prime field squaring.*

4.6 Summary

In this chapter we introduced several scalar multiplication methods; these can be split into three parts. We started with basic scalar multiplication concepts, and provided popular scalar recoding algorithms. Later, we gave the first class of scalar multiplication methods which utilize precomputations. We explained them in detail and showed how different improvements where achieved. The influence of those improvements can be seen in the explicitly stated execution time estimations. We stated how a speedup for scalar multiplication methods can be gained by using an *efficiently computable endomorphism* in combination with a scalar conversion. This concept is not universally applicable, as an efficiently computable endomorphism is necessary. We think it illustrates a different venue for speeding up scalar multiplication methods and is therefore of general interest. The second class of multiplication methods we explained was the class of *regular* multiplication methods. We gave several flavors of the *Montgomery ladder.* We showed some ideas on how the basic *Montgomery ladder* was improved using different addition and doubling formulas and coordinate systems. The computational costs of every version of the *Montgomery ladder* were explicitly stated to ease the comparison. We also showed how *Joye's Double-and-Add* multiplication method was improved, and stated the computational costs. We tried to give an interesting overview of the developments in recent years, and illustrate and explain them with carefully selected examples.

Chapter 5

Cryptography in Java

In this chapter, we give an introduction to the software architecture in Java related to cryptography. All the scalar multiplication methods given in this thesis were implemented in Java as part of the *ECCelerate*™ add-on to the *IAIK Java Cryptography Extension* (IAIK-JCE) software library. The IAIK-JCE is developed at the *Institute for Applied Information Processing and Communications* (IAIK) at *Graz University of Technology*. This chapter is based on the extensive documentation given in [Ora14] and information published at [IT14].

This chapter is structured as follows. We start by giving an introduction to the *Java Cryptography Architecture* (JCA) in Section 5.1, to give all readers a high-level overview of the programming context for cryptographic implementations in Java. In Section 5.2, we show how a *Java Cryptography Extension* (JCE) is designed and integrated into the Java infrastructure. This is followed by Section 5.3, which gives a detailed look at the *ECCelerate*™ add-on that provides all the ECC support for the IAIK-JCE library. Of special interest are the implemented performance improvements and optimizations to the *finite field level* of all ECC operations. The chapter concludes with a short summary in Section 5.4.

5.1 Java Cryptography Architecture

The *Java Cryptography Architecture* is a framework within Java targeted at providing cryptographic services to applications. The goal is to give an interface to cryptographic functionalities that is *implementation independent* and ensures *implementation interoperability.* Those two design goals are reached in different ways. One of the core principles is defining a standard application programming interface (API) for all so-called *cryptographic providers.* An application doesn't have to implement its own cryptographic functions. It requests the desired cryptographic functionality from one of the available providers. *Cryptographic providers* implement all security related functionality, e.g., signature algorithms, encryption and decryption, or key conversion services. Each of them is able to offer several cryptographic services simultaneously. The concept of a *cryptographic provider* is detailed in Section 5.2.

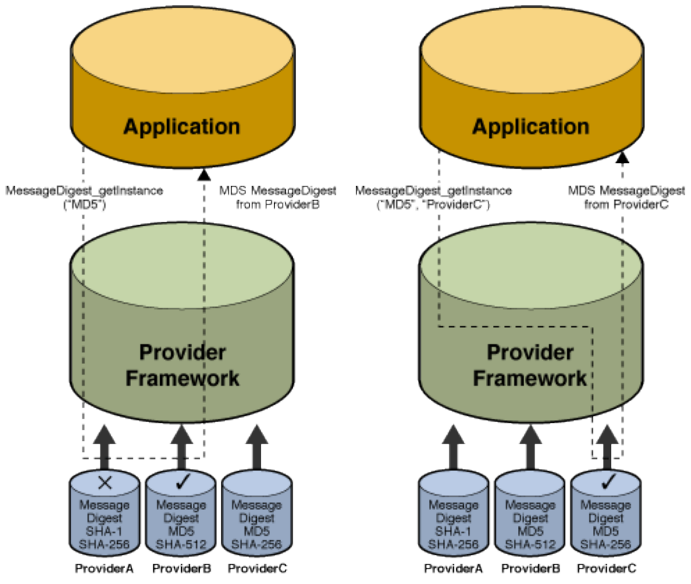
Application
Application
MessageDigest_getInstance ("MD5")
MDS MessageDigest from ProviderB
MessageDigest_getInstance ("MD5", "ProviderC")
MDS MessageDigest from ProviderC
Provider Framework
Provider Framework
×
✓
Message Digest SHA-1 SHA-256
Message Digest MD5 SHA-512
Message Digest MD5 SHA-256
ProviderA
ProviderB
ProviderC
Message Digest SHA-1 SHA-256
Message Digest MD5 SHA-512
✓
Message Digest MD5 SHA-256
ProviderA
ProviderB
ProviderC

Figure 5.1: Overview of the JCA (taken from [Ora14])

The strict enforcement of a standardized API for *cryptographic providers* guarantees *implementation independence*. Numerous providers can be registered at the JCA, and applications request the functionality via the standardized interface. This is shown in Figure 5.1. They are thus able to use several providers simultaneously. The standardized interface also enables *implementation interoperability*. This means that, for example, in a cryptographic protocol an application can generate its cryptographic keys with provider `A`, and then use those keys for cryptographic operations which are performed by a different provider `B`. Analogously, the providers can be used by several applications simultaneously. Other important design principles of the JCA are *algorithm independence* and *algorithm extensibility*. *Algorithm independence* is realized through so-called engine classes, which are designed to represent the functionality and algorithmic flows in cryptographic building blocks in a very abstract, high-level way. Examples for engine classes are `MessageDigest`, `SecureRandom` or `KeyGenerator`. This helps to encapsulate the concrete implementations of algorithms. Furthermore, it enables *algorithm extensibility* as those engine classes simply call actual implementation classes which have the same method signatures (more on that in Section 5.2). This leads to easily extensible and easily updatable cryptographic functionality.

5.2 Java Cryptography Extension

A *Java Cryptography Extension* (JCE) has to implement a *cryptographic provider* and consists of one or several packages. The entirety of JCE's functionality, meaning the implemented *cryptographic algorithms* and *cryptographic schemes*, are registered with the JCA. This can happen *statically*, where the *cryptographic provider* is entered in the security properties configuration file, or *dynamically* via method calls to the `Security` class in the JCA. The *dynamic* registration has some restrictions. First the provider has to have the necessary privileges, and secondly it is only added to the currently running Java virtual machine. A typical program flow looks like this: an application uses the software library to request an algorithm's implementation via so-called *factories* provided by the JCA. For a comprehensive list of all available *factories*, see [Ora14]. There are two possible scenarios after the request is issued. Both are illustrated in Figure 5.1 for a so-called `MessageDigest` engine class combined with the `MD5` hash function.

1. The user specified no provider, the JCA checks all providers in descending preference order and takes the first available implementation of the algorithm (depicted on the left side of Figure 5.1).
2. The user specified a provider, the JCA checks if the algorithm is indeed available, and if so, returns an instance of the algorithm from the chosen provider (depicted on the right side of Figure 5.1).

The JCA returns an instance of a `MessageDigest` engine class. This engine class is intended to represent the abstract concept of a cryptographically secure message digest. It encapsulates the following call structure. The call to specific functionality of the `MessageDigest` class is forwarded to an *abstract* so-called `MessageDigestSpi` class, which implements the Service Provider Interface (SPI). *Abstract* means that these classes cannot be *instantiated*. The *cryptographic provider* therefore subclasses the `MessageDigestSpi` class with a class that implements the actual functionality of a cryptographically secure message digest. This is the class that executes the requested operations, and the result is then communicated back to the application in the reversed call order. There are three general types of engine classes that a *cryptographic provider* can implement, namely:

1. *Cryptographic operations*: classical cryptographic tasks and building blocks e.g., en- and decryption, hash functions (message digests), (pseudo)random number generators, or digital signatures.
2. *Generators or converters of cryptographic material*: this typically means keys, key pairs and standard parameters for algorithms.
3. *Objects (keystores or certificates)*: those are standardized data structures which hold cryptographic data, for example certificate stores and key stores.

For a complete list of all available engine classes see [Ora14].

5.3 IAIK-JCE and ECCelerate™

The *IAIK Java Cryptography Extension* (IAIK-JCE) is a software library which implements a JCE as described in Section 5.2. It offers an implementation of the *Java Development Kit's* default functionality, enhanced by a whole ecosystem of supporting software, e.g., advanced X.509 certificate support and ASN.1 structures. Please see [IT14] for a comprehensive list.

The *ECCelerate*™ library is one component of the IAIK-JCE; it provides all ECC support. In the following section, we give an overview of some implemented *finite field level* optimizations. This is of interest as in Chapter 6 timings and benchmarks are stated which all benefited from the here mentioned optimizations. The entire finite field arithmetic is implemented using various mathematical speedups. For binary fields, the following optimizations are implemented:

1. Where possible, all binary field arithmetic works in-place.

2. For NIST standardized *binary* curves fast reduction polynomials according to [Nat13] are used.
3. All elements are represented in a custom, optimized, little-endian `long[]` format.
4. Squaring of elements is performed in linear time, using lookup tables, as stated by the authors of [SOOS95].
5. Multiplication of elements is performed using a left-to-right comb multiplier ([HMV04, Algorithm 2.34]) with precomputation window size w as follows:
   (a) $w = 8$ for curve parameters,
   (b) $w = 4$ for all other multiplications
6. If possible, several field elements are inverted simultaneously using [HMV04, Algorithm 2.26].

It is important to note that a performance gain can be achieved if a *binary finite field element* is multiplied more than once. In this case, it is possible to reuse the precomputed data and enjoy a considerable speedup without precomputation costs. In our performance evaluation in Chapter 6, this is denoted as M_4 for a *binary finite field multiplication* with reused precomputation data of window size $w = 4$, and M_8 denotes a multiplication with reused precomputation data of window size $w = 8$. For prime fields, the following optimizations are implemented:

1. Where possible, all prime field arithmetic works in-place.
2. For NIST standardized *prime* curves, fast reduction formulas according to [Nat13] are used.
3. If possible, several field elements are inverted simultaneously using [HMV04, Algorithm 2.26].

This list is not exhaustive; only the most relevant *finite field level* optimizations for the scalar multiplication benchmarks are given. The scalar multiplication methods implemented cannot be chosen directly by an application. Via a so-called `OptimizationLevel` parameter, the desired time-memory trade-off can be adjusted to best fit the target platform requirements. Depending on this parameter, the *ECCelerate*™ library chooses the scalar multiplication method and precomputation effort accordingly. Another important configuration parameter is `FullCheckEnabled`, which enables public-key verification. As mentioned in Section 3.5, disabling this check can lead to security problems if untrusted public-keys have to be processed. If all used keys can be trusted, disabling this check improves performance as the total number of checks is decreased.

5.4 Summary

In this chapter, we gave an introduction to the high level concepts of the *Java Cryptography Architecture* (JCA). As Java is an object-oriented programming language, several interesting software design concepts where shown. This was followed by a closer look at the *Java Cryptography Extension* (JCE) design. We detailed how a JCE implementation works, what interface restrictions are in place, and how the JCE is integrated into the bigger structure of the JCA. Additionally we focused on the IAIK-JCE, and especially on the *ECCelerate*™ add-on as it provides all the ECC support for the IAIK-JCE library. We took a deeper look at the *finite field level* optimizations of *prime* and *binary* fields. This should give an idea about how ECC can be optimized on different levels, e.g., *finite field level* or the *elliptic curve level.* Additionally, we provided some interesting parameters in the *ECCelerate*™ configuration and described their impact on performance. This gives all readers a bit of context for the upcoming performance measurements and timing results in Chapter 6.

Chapter 6

Results

In this chapter, we will detail interesting implementation aspects for some scalar multiplication methods mentioned in Chapter 4. Additionally, the results for our Huff curve related research are given. All implementations were made for the *ECCelerate*™ [Han14] software library, which provided a very good and well-structured code base. The *ECCelerate*™ add-on and some of its optimizations were already discussed in Chapter 5.

This chapter is structured as follows. In Section 6.1, we begin with giving insight into the implementations of various scalar multiplication algorithms. Next, in Section 6.2, we give details on the *Huff curve* related implementations, starting with the mapping formulas between Huff and Weierstrass curves. This section is followed by benchmark and timing results in Section 6.3. The chapter concludes with a short summary in Section 6.4.

6.1 Scalar Multiplication Method Implementations

In this section, we will detail noteworthy implementation specific decisions and formulas for some of the scalar multiplication methods we implemented. The intent is to show solutions for implementation problems one may encounter, and to offer some smaller improvements we made while working on scalar multiplication algorithms.

Improved Fixed-Base Comb Multiplication Implementation

In addition to the *improved fixed-base comb method for fast scalar multiplication* (discussed in Section 4.2.2), which is stated for *prime* Weierstrass curves, we implemented a version of the *improved fixed-base comb method for fast scalar multiplication* for *binary* Weierstrass curves. The *Sakai-Sakurai method for direct doubling* is only available on *prime Weierstrass* curves so we needed a suitable replacement. The authors of [CF05] give an algorithm for *repeated doublings* for *affine* coordinates on binary curves which is applicable to this task. This algorithm is shown in Algorithm 23.

Algorithm 23 Repeated doublings [CF05, Algorithm 13.42]

```
Input: A point $P = (x_1, y_1)$ on $E$ such that $[2^k]P \neq \mathcal{O}$ and an integer $k \geq 2$.
Output: The point $[2^k]P$ of coordinates $(x_3, y_3)$.
 1: $\lambda = \frac{x_1+y_1}{x_1}$
 2: $u = x_1$
 3: for $i = 1$ to $k - 1$ do
 4:     $x' = \lambda^2 + \lambda + a_2$
 5:     $\lambda' = \lambda^2 + a_2 + \frac{a_6}{u^4+a_6}$
 6:     $u = x'$
 7:     $\lambda = \lambda'$
 8: end for
 9: $x_3 = \lambda^2 + \lambda + a_2$
10: $y_3 = u^2 + (\lambda + 1)x_3$
11: return $(x_3, y_3)$
```

The runtime of Algorithm 23 for a scalar k can be estimated as:

$$kI + (k + 1)M + (3k - 1)S.$$

Here, I denotes a *binary* field inversion, M denotes a *binary* field multiplication and S denotes a *binary* field squaring. We adapted and implemented the algorithm for several *projective* coordinate systems, as can be seen in Section 6.3. The *binary Weierstrass* version of the *improved fixed-base comb method for fast scalar multiplication* performs very well; see Table 6.4 for details.

Montgomery Ladder Implementation

There are some interesting circumstances one has to take into account when implementing a *Montgomery ladder*, otherwise problems might arise. In the case of a *differential Montgomery ladder*, if the scalar k is given as $k = |E| - 1$. Then, in the last step of the *differential Montgomery ladder*, the point $(k + 1)P$ becomes $\mathcal{O}$. This causes problems when trying to recover the full coordinates for the result of kP. A possible workaround is to calculate the *differential Montgomery ladder* with scalar $k - 1$ and add P to the result of the *differential Montgomery ladder*.

One should also be aware that using a *differential Montgomery ladder* may make the implementation vulnerable to *fault attacks*. This is mainly because the y-coordinate is not used in the calculation. An example of such an attack is given by the authors of [FLRV08], where they use the elliptic curve's *twist* to transfer the calculation to a cryptographically weak elliptic curve given by a set of points in a subgroup of the curve's *twist*. This subgroup is chosen such that the group order has small factors. This enables them to solve the ECDLP and find the scalar k of the scalar multiplication kP. As a countermeasure, the authors of [FLRV08] suggest regular checks if the point is still on the cryptographically strong elliptic curve while executing the *differential Montgomery ladder*. The authors of [BL14a] suggest choosing a *twist safe* curve or enabling *point compression* to avoid this pitfall.

Montgomery Ladder Multiplication with Co-Z Coordinates Implementation

For our *Montgomery ladder with Co-Z coordinates* implementation, we used formulas given by the authors of [GJM10]. For the *conjugate Co-Z point addition*, we used [GJM10, Algorithm 6] and for the *Co-Z point addition with update* we used [GJM10, Algorithm 1].

The authors do not explicitly state an algorithm for the *Co-Z point doubling with update* operation; rather they described it in [GJM10, Section 4.3]. We implemented the *Co-Z point doubling with update* as stated in Algorithm 24.

Algorithm 24 Co-Z point doubling with update

```
Input: $X_1, Y_1, Z_1$.
Output: $(X_2 : Y_2 : Z_2), (X_3 : Y_3 : Z_3 : Z_2)$.
1:  1. $E = Y_1^2$           9. $N = Z_1^2$                  17. $Z_2 = Y_1 + Z_1$
    2. $B = X_1^2$          10. $M = 3B$                     18. $Z_2 = Z_2^2$
    3. $L = E^2$            11. $M = M + (N^2 a_4)$          19. $Z_2 = Z_2 - E$
    4. $S = X_1 + E$        12. $X_2 = M^2$                  20. $Z_2 = Z_2 - N$
    5. $S = S^2$;           13. $X_2 = X_2 - 2S$             21. $S_3 = E - X_1$
    6. $S = S - B$          14. $Y_2 = S - X_2$              22. $S_3 = 4S_3$
    7. $S = S - L$          15. $Y_2 = Y_2 M$                23. $X_3 = S$
    8. $S = 2S$             16. $Y_2 = Y_2 - 8L$             24. $Y_3 = 8L$
2: return $(X_2 : Y_2 : Z_2), (X_3 : Y_3 : Z_2)$
```

Algorithm 24 returns a representation of the input point, as well as the doubled input point. Both points have the same Z-coordinate.

Montgomery Ladder Multiplication with XZ-only Co-Z Coordinates Implementation

The authors of [HJS11] gave several algorithms for the *differential Montgomery ladder step* as well as the coordinate recovery. Their formulas reflect the typical time-memory trade-off, where memory in this case is used registers of a processor. As in our Java implementation, we are not concerned with processor registers, so we chose the faster algorithms. In this Section, M denotes a *prime* field multiplication, S denotes a *prime* field squaring and A denotes a *prime* field addition, additionally M_{a_4} and M_{4a_6} denote the costs of a *prime* field multiplication with curve parameters a_4 or $4a_6$ respectively.

One of the named *differential Montgomery ladder step* algorithms is Algorithm [HJS11, Algorithm 5] with costs of $9M + 5S + 14A + 1M_{a_4} + 1M_{4a_6}$ and Algorithm [HJS11, Algorithm 6] with costs of $10M + 5S + 13A$. On *prime* fields, there is no field element multiplication method implemented which can gain a speed advantage by reusing precomputed data. Therefore, Algorithm [HJS11, Algorithm 6] (given in Algorithm 25) is faster and, hence, the preferable option.

Algorithm 25 Out-of-place differential addition-and-doubling in a projective Co-Z coordinate system [HJS11, Algorithm 6]

```
Input: $X_1, X_2, T_D = x_D Z, T_a = a_4 Z^2, T_b = 4a_6 Z^3$.
Output: $X_1, X_2, T_D, T_a, T_b$.
1:  1. $R_2 = X_1 - X_2$    11. $R_3 = R_5 R_2$        21. $X_2' = R_1 R_4$
    2. $R_1 = R_2^2$        12. $R_3 = R_3 + T_b$      22. $R_2 = R_1 R_3$
    3. $R_2 = X_2^2$        13. $R_5 = X_1 + X_2$      23. $R_3 = R_2 T_b$
    4. $R_3 = R_2 - T_a$    14. $R_2 = R_2 + T_a$      24. $R_4 = R_2^2$
    5. $R_4 = R_3^2$        15. $R_2 = R_2 - R_1$      25. $R_1 = T_D R_2$
    6. $R_5 = X_2 + X_2$    16. $X_2 = X_1^2$          26. $R_2 = T_a R_4$
    7. $R_3 = R_5 T_b$      17. $R_2 = R_2 + X_2$      27. $T_b = T_3 R_4$
    8. $R_4 = R_4 - R_3$    18. $X_2 = R_5 R_2$        28. $X_1 = X_1 - R_1$
    9. $R_5 = R_5 + R_5$    19. $X_2 = X_2 + T_b$      29. $T_D = R_1$
   10. $R_2 = R_2 + T_a$    20. $X_1 = R_3 X_2$        30. $T_a = R_2$
2: return $X_1, X_2, T_D, T_a, T_b$
```

A similar choice of algorithms is available for the coordinate recovery process. Again, the authors of [HJS11] state several options. In Algorithm [HJS11, Algorithm 7], a version with costs of $8M+2S+8A+1M_{a_4}+1M_{4a_6}$ is given and in Algorithm [HJS11, Algorithm 8] a version with costs of $10M+3S+8A$ is introduced. As the number of used registers is irrelevant to us, we chose Algorithm 8 (given in Algorithm 26), as it is cheaper in terms of computation effort.

Algorithm 26 Out-of-place $(X : Y : Z)$-recovery in projective Co-Z coordinate system [HJS11, Algorithm 8]

```
Input: $X_1, X_2, T_D = x_D Z, T_a = a_4 Z^2, T_b = 4a_6 Z^3, x_D, y_D$.
Output: $(X_1 : X_2 : Z)$.
1:  1. $R_1 = T_D X_1$        8. $R_4 = R_4 - R_3$     15. $R_3 = R_3 + R_3$
    2. $R_2 = R_1 + T_a$      9. $R_4 = R_4 + R_4$     16. $X_1 = R_3 R_1$
    3. $R_3 = X_1 + T_D$     10. $R_4 = R_4 + T_b$     17. $R_1 = R_2 T_D$
    4. $R_4 = R_2 R_3$       11. $R_2 = T_D^2$         18. $Z = R_3 R_1$
    5. $R_3 = X_1 - T_D$     12. $R_3 = X_1 R_2$       19. $R_2 = x_D^2$
    6. $R_2 = R_3^2$         13. $R_1 = x_D R_3$       20. $R_3 = R_2 x_D$
    7. $R_3 = R_2 X_2$       14. $R_3 = y_D + y_D$     21. $X_2 = R_3 R_4$.
2: return $(X_1 : X_2 : Z)$
```

6.2 Huff Curve Related Implementation

To our knowledge we were the first to implement *generalized binary Huff curves* as introduced in [DJ11]. We were also the first to publicly state, in [GH13], so-called *Huff curve parameters* for all standardized *binary* NIST curves to reduce the implementation efforts for others. These parameters are given in Table 6.1, and all additionally necessary *auxiliary* parameters are given in Table 6.2.

As there are no standardized Huff curves yet, all points need to be mapped from Weierstrass curves to Huff curves and vice versa. We introduced highly efficient all-in-one formulas for this task. As outline in Section 3.3, a mapping in either direction consists of multiple steps because an *isomorphism* as well as a *birational equivalence* have to be taken into account. As discussed in Section 3.3, the *isomorphism* between the Weierstrass curve E_W and a Weierstrass curve $E_{W'}$, which is *birationally equivalent* to a *Huff* curve, is given as (taken from [GH13]):

$$\begin{aligned}\Theta : E_W(\mathbb{F}_{2^m}) &\to E_{W'}(\mathbb{F}_{2^m}),\\ (u, v) &\longmapsto (\mu^2 u, \mu^3(v + su + \sqrt{a_6})),\end{aligned}$$

with its inverse given as:

$$\begin{aligned}\Phi : E_{W'}(\mathbb{F}_{2^m}) &\to E_W(\mathbb{F}_{2^m}),\\ (u', v') &\longmapsto (v^2 u', v^3 v' + s v^2 u' + \sqrt{a_6}).\end{aligned}$$

Here $\mu = (a+b)f, v = \mu^{-1}$ where f and s are Huff parameters, as given in Section 3.3. The *birational equivalence* is given in Section 3.3 as:

$$\Psi : H \to W' \text{ with } (x, y) \longmapsto \left(\frac{ab}{xy}, \frac{ab(axy+b)}{x^2 y}\right),$$

$$\Phi : W' \to H \text{ with } (u, v) \longmapsto \left(\frac{b(u+a^2)}{v}, \frac{a(u+b^2)}{v+(a+b)fu}\right).$$

Table 6.1: Generalized Huff curve parameters for NIST recommended binary curves (cf. [Nat13]), (taken from [GH13, Table 1])

Curve	a	b	f
B-163	0x1	0x253f3c45a6d779b47e63758c35336f0679b42f4c0	0x6
K-163	0x1	0x20000000000000000000000000000000000000033	0x6
B-233	0x1	0x115b7c737bec7a5cc19212911c2bd03cadb9a29ddf9b1dc64b\\ 8b3550fb3	0x3
K-233	0x1	0x1e5ff5c884156aaebbd38370425882dff04f04ba05a7f40740\\ 82385c149	0x2
B-283	0x1	0x6a263cdd28c309d3d3068046747abe51375b0d763dccc64868\\ 251918d59c21842dd4fe1	0x3
K-283	0x1	0x7a0fa1ffdaf44208a4efb593c405714e0fbc4423dd0db57384\\ 89cb583073c2cae153d0d	0x2
B-409	0x1	0x3f2918c0e689aca093d4cf5a389aeda96eb5cdcb930617991d\\ 09111a3f91dc7283123ef8ab912744e193c34c9bd3cd532e17b7	0x9
K-409	0x1	0x846538361ed11b7c42b9e302169a3ea16009df82f80a155d56\\ 39d78d4ba8dd02284110d6b3fbc05dda9c0ed1c0d6316c72d676	0x2
B-571	0x1	0x3c0904534c17c94a947b971ee5e6a3f3fb917dd3b57d7ad1f6\\ ea35ec2593bae024934b8efe08d2a5bb97c4286665408d50f80c\\ afc8dfbee0011c03e785fe39c94c977d5e3a7f065	0xf
K-571	0x1	0x6a28a2cf6fb77a9485f438a79f8832d86c465b689fd80b3d9c\\ 4b1ef40380b5d92f85044e450336618a69b209eb37ecdd23da7b\\ f7ee9e0fc1e98248edb0dc92f3510027be50cd2bb	0x2

Given a point $P_W = (u_1, v_1)$ on $E_W(\mathbb{F}_{2^m})$ and a point $P_{W'} = (u'_1, v'_1) \in E_{W'}(\mathbb{F}_{2^m})$, where $\Theta(P_W) \in E_{W'}(\mathbb{F}_{2^m})$ and $P = \Omega(P_{W'})$ is the corresponding point on $E_H(\mathbb{F}_{2^m})$. Our all-in-one formulas for directly mapping points were derived as follows.

Weierstrass to Huff Curve

Given a point $P_W = (u, v)$ on the Weierstrass curve $E_W(\mathbb{F}_{2^m})$, the following formula maps this point to a point $P_H(W : Z)$ in WZ-coordinates on the curve $E_H(\mathbb{F}_{2^m})$. This is achieved by inserting the steps of $\Phi \circ \Theta$ in a formula. The formula for directly mapping P_W to $P_H(W : Z)$ is given as (taken from [GH13]):

$$(W : Z) = \left(ab\left(\mu^2 u + a^2\right)\left(\mu^2 u + b^2\right) : \mu^6\left(v + su + \sqrt{a_6}\right)\left(\left(v + su + \sqrt{a_6}\right) + u\right)\right).$$

The authors of [GH13] give the following decomposition to minimize the calculation effort of the above formula:

$$\begin{aligned} A &= \mu^2 u, & B &= (A + a^2), & C &= (A + b^2), \\ D &= v_1 + su + \sqrt{a_6}, & E &= D + u, & & \\ W &= ab \cdot B \cdot C, & Z &= \mu^6 \cdot D \cdot E. & & \end{aligned}$$

As one can see, there are several values which can be determined *a priori*, namely a^2, b^2, and $\sqrt{a_6}$. For μ^2, μ^6, s, and ab the lookup tables for the *binary field multiplication* method can be precomputed. In our implementation, we precompute these values with window size $w = 8$. Therefore, we are able to evaluate the decomposed formula with $\mathbf{2M} + \mathbf{4M_8}$.

Table 6.2: Auxiliary parameter s for mapping between NIST recommended binary Weierstrass curves and the corresponding binary Huff curves (taken from [GH13, Table2])

Curve	s
`B-163`	`0x4058c6f9ae170f30f3ec9def6b2ddf2a28f0c0872`
`K-163`	`0x4058c6f9ae170f30f3ec9def6b2ddf2a28f0c0872`
`B-233`	`0xe7b4bcdb4af3163783507af91971d49927298e32e548d55b3a0b602a42`
`K-233`	`0xb1ce7164613a37fed984a32b18d265ada947ed207757d373d4e139835a`
`B-283`	`0x21b24c4336e195a894fc9021fabac4e6988ff780c29522af3261508be\` `fb321108eda3fa`
`K-283`	`0x387fd1da986a7fa48458bf27d26d9162d60c2f7f6e3da61f30d215c6b\` `193e73f223326e`
`B-409`	`0x106176448c66d8e7d0ddc074d76277a7ac8093ee53499d108099266d9\` `82c68dae5cb61d5054b30ecfce3c3beebc8cbecb904fd0`
`K-409`	`0x15dedff38eafec7e43a277eea795fbb1d52e6075a8bfe6a275be0dcba\` `f6b781f8c9d37e4f414e8de3634d946434d9b6a6d62e20`
`B-571`	`0x12007d9377488ff6122ccce941d1cef856279188c8e82a6696a918b2f\` `ccd78353385beb5e972f83d491d22db627117ab1580dabd23c6e8adeb99\` `d3bdbc95d6fb645833ba6b4f182`
`K-571`	`0x2780c6d786569591600518d211a5d6fbd900d9b44a1e4e65016d2331d\` `d243a6b31db129832a46326c7e3fd9b43f900ee58ed165e550a3cc3a41f\` `b88b001fa79f398351bb7c35dea`

Here, M denotes a *binary field multiplication* and M_8 denotes a *binary field multiplication* with reused precomputation data of window size $w = 8$.

Huff Curve to Weierstrass

The intent of the following formulas is to convert the WZ-coordinate result of a *differential Montgomery ladder* $W(k \cdot P) \in E_H(\mathbb{F}_{2^m})$, as given in Section 4.4.5, to an *affine* point $P_W = (u_1, v_1)$ on a Weierstrass curve $E_W(\mathbb{F}_{2^m})$. These formulas work only with WZ-coordinates. Similar to Section 6.2, a *birational equivalence*, namely $\Omega \circ \Psi$, has to be evaluated. To save costly inversions, the steps of Ω are *implicitly* applied by inserting them into the coordinate recovery formula. A coordinate recovery formula needs to be applied regardless after calculating the *differential Montgomery ladder.* The authors of [GH13] show this as follows. Given the y-coordinate recovery formula (taken from [LD99])

$$v_1 = \frac{(u_1 + u)\left((u_1 + u)(u_2 + u) + u^2 + v\right)}{u} + v, \tag{6.1}$$

where $u_2 = \frac{ab}{\omega((k+1)\cdot P)}$ is calculated via the point $W((k+1)P) = (W_2 : Z_2)$ which is calculated during the last step of the *Montgomery* ladder. The insertion step consists of replacing $u_1 = \frac{ab}{\omega(k\cdot P)}$ with $u_1 = \frac{\delta \cdot Z_1}{W_1}$. Here $\omega(K\cdot P) = \frac{W_1}{Z_1}$, given that $W(k\cdot P) = (W_1 : Z_1)$, $\delta = \frac{ab}{\mu^2}$ and $\mu = (a+b)f$. This gives the following formula:

$$v_1 = \frac{(\delta Z_1 + uW_1)\left((\delta Z_1 + uW_1)(\delta Z_2 + uW_2) + \left(u^2 + v\right) W_1 W_2\right)}{uW_1^2 W_2} + v,$$

and leads to:

$$\begin{aligned} U_1 &= \delta Z_1 u W_1 W_2, \\ V_1 &= \beta\left(\beta\left(\delta Z_2 + uW_2\right) + \left(u^2+v\right)W_1W_2\right) + uvW_1^2W_2, \end{aligned}$$

where $\beta = \delta Z_1 + uW_1$. Finally, $(u_1, v_1) = \left(\frac{U_1}{uW_1^2W_2}, \frac{V_1}{uW_1^2W_2}\right)$ can be calculated. The authors of [GH13] give the following decomposition for evaluating this formula:

$$\begin{aligned} &A = \delta Z_1, && B = \delta Z_2, && C = uW_1, \\ &D = uW_2, && E = A + C, && F = B + D, \\ &G = W_1W_2, && H = (u^2+v), && I = (C \cdot G)^{-1}, \\ &J = A \cdot G, && K = J \cdot I, && L = E \cdot I, \\ &u_1 = uK, && v_1 = L \cdot (E \cdot F + H \cdot G) + v. && \end{aligned}$$

Again δ is a constant, e.g., a precomputable value, meaning that multiplying by δ requires $\mathbf{1M_8}$. Furthermore, several *binary* field scalar multiplication lookup tables of intermediate results can be reused, namely the tables of I and G, given a reduction of $\mathbf{4M}$ to $\mathbf{2M} + \mathbf{2M_4}$ and the lookup table of u_1, replacing $\mathbf{3M}$ with $\mathbf{1M} + \mathbf{2M_4}$. These improvements can be further advanced if the point $P_W = (u, v)$ is fixed. Then the lookup tables of u and H can be precomputed with window size $\omega = 8$, which replaces the required $\mathbf{1M} + \mathbf{3M_4} + \mathbf{1S}$ with $\mathbf{4M_8}$. This gives the following overall costs for transferring the result to the Weierstrass curve of origin, including the y-coordinate recovery, of $\mathbf{1I} + \mathbf{5M} + \mathbf{2M_4} + \mathbf{6M_8}$ with a fixed P_W and $\mathbf{1I} + \mathbf{6M} + \mathbf{5M_4} + \mathbf{2M_8} + \mathbf{1S}$ in the general case.

This gives total costs of $\mathbf{1I} + \mathbf{8M} + \mathbf{5M_4} + \mathbf{6M_8} + \mathbf{1S}$ for a back and forth conversion, including a y-coordinate recovery, for any arbitrary point P_W. If a conversion to WZ-coordinates is necessary, additional costs of $\mathbf{2M} + \mathbf{4M_8}$ have to be taken into account. Here, I denotes a *binary field inversion*, M denotes a *binary field multiplication*, M_4 denotes a *binary field multiplication* with reused precomputation data of window size $w = 4$, and M_8 denotes a *binary field multiplication* with reused precomputation data of window size $w = 8$.

6.3 Benchmark

In this section, we show benchmark results for some of our *Montgomery ladder* implementations and for some *fixed-base comb scalar multiplication* methods as well as for our *Huff* curve related implementation.

We start with scalar multiplication method benchmarks in Section 6.3.1, followed by Section 6.3.2 where we give timings for our *Huff* curve related implementation.

6.3.1 Scalar Multiplication Method Timings

All benchmarks given in this section were measured on an Intel Core™2 Duo T7500 running Ubuntu Linux 12.04/amd64 and OpenJDK 7u55/amd64 in server mode.

In Table 6.3, we give the benchmark results for several Co-Z scalar multiplication methods. We took these measurements on NIST *prime* curves. As one can see, the coordinate system used heavily influences the speed of the scalar multiplication. This benchmark also shows the effect of the improvement from standard Co-Z formulas to *differential* Co-Z formulas. We added the improved *differential Joye's Double-and-Add* method to give some context to the *differential Montgomery ladder* implementations.

Table 6.3: Comparison of Co-Z scalar multiplication methods on NIST *prime* curves

Montgomery ladder	`P-192` [μs]	`P-224` [μs]	`P-256` [μs]	`P-384` [μs]	`P-521` [μs]
Co-Z XY (Section 4.4.3)					
Jacobian, $a = 1$	4492.89	5967.36	9225.52	20836.66	30661.05
Jacobian, $a = 0$	13346.15	17311.02	19557.10	20850.56	30676.73
Co-Z (Section 4.4.2)					
Jacobian, $a = 1$	4750.19	6401.69	9920.93	23101.62	34383.17
Jacobian, $a = 0$	14282.25	18691.94	21170.29	23109.81	34420.64
Co-Z Double-and-Add					
(Section 4.5)					
Jacobian, $a = 1$	5102.52	6877.36	10476.91	23667.33	34929.21
Jacobian, $a = 0$	15240.88	19864.00	22445.01	23715.42	34915.39

In Table 6.4 and Table 6.5 we introduce benchmarks for comb based scalar multiplication methods on NIST curves. The *improved fixed-base comb method* is implemented as detailed in Section 6.1. The *comb scalar multiplication* is implemented as given in [HMV04, Algorithm 3.44], and the *two-table comb scalar multiplication* is implemented as given in [HMV04, Algorithm 3.45]. We took our timings with window size $w = 8$, and set the v parameter for the *improved fixed-base comb method* to $v = 8$. Again, the choice of coordinate system influences the execution time heavily; *affine* coordinates are especially slow. It is of special interest that on *binary* NIST curves, the *improved fixed-base comb method* outperforms even the *two-table comb scalar multiplication method* in almost all measured cases.

Table 6.4: Comparison of comb scalar multiplication methods on *binary* NIST curves

Type	`B-163` [μs]	`B-233` [μs]	`B-283` [μs]	`B-409` [μs]	`B-571` [μs]
Improved Comb multiplications					
Affine, $a = 1$	1933.98	3407.40	6466.20	13187.45	32572.94
Projective, $a = 1$	1091.53	1923.82	2945.98	6030.58	10902.75
Jacobian, $a = 1$	1168.78	1969.22	3017.64	6111.81	11058.73
Lopez-Dahab, $a = 2$	1021.16	1766.04	2747.68	5554.28	10082.61
Lopez-Dahab, $a = 1$	993.47	1756.92	2737.76	5552.52	10097.10
Two-table comb scalar multiplications					
Affine, $a = 1$	2029.90	4247.07	7010.36	14431.47	36167.24
Projective, $a = 1$	1426.35	2653.09	3876.52	7540.47	14948.43
Jacobian, $a = 1$	1262.27	2420.91	3512.70	6723.64	13338.85
Lopez-Dahab, $a = 2$	1006.42	1913.49	2784.24	5325.40	10612.48
Lopez-Dahab, $a = 1$	1003.63	1910.59	2783.85	5330.17	10588.21
Comb scalar multiplications					
Affine, $a = 1$	2715.91	5687.72	9478.46	19232.25	48459.79
Projective, $a = 1$	1704.28	3341.97	4892.63	9465.66	18720.98
Jacobian, $a = 1$	1514.15	2907.22	4252.55	8091.62	15985.59
Lopez-Dahab, $a = 2$	1190.57	2264.44	3298.65	6313.03	12596.84
Lopez-Dahab, $a = 1$	1188.27	2259.87	3295.19	6324.06	12631.37

Note that this is not the case on *prime* NIST curves. Here, the *improved fixed-base comb method* is especially fast compared to the other scalar multiplication methods when operating on *affine* coordinates. With all *projective* coordinate systems, it is on the same level as the *comb scalar multiplication method* but significantly slower than the *two-table comb scalar multiplication method.*

Table 6.5: Comparison of comb scalar multiplication methods on *prime* NIST curves

Type	`P-192` [μs]	`P-224` [μs]	`P-256` [μs]	`P-384` [μs]	`P-521` [μs]
Improved Comb multiplications					
Affine, $a = 1$	1542.81	2587.32	3384.31	8159.78	15745.06
Projective, $a = 1$	747.41	1256.29	1665.86	4108.33	7512.85
Jacobian, $a = 1$	766.82	1279.43	1700.23	4164.93	7466.10
Extended Jacobian, $a = 1$	773.89	1276.26	1708.14	4168.49	7489.65
Extended Jacobian, $a = -3$	752.61	1267.43	1707.86	4167.98	7512.09
Jacobian, $a = 0$	1888.45	2932.00	3075.61		
Extended Jacobian, $a = 0$	1901.00	3004.32	3089.97		
Two-table comb scalar multiplications					
Affine, $a = 1$	3153.49	4429.91	6111.41	15632.57	32048.20
Projective, $a = 1$	595.41	792.15	1250.75	2951.95	4460.77
Jacobian, $a = 1$	608.49	810.07	1252.00	2857.46	4237.68
Extended Jacobian, $a = 1$	592.92	789.08	1211.76	2758.87	4106.73
Extended Jacobian, $a = -3$	595.46	789.65	1210.33	2758.00	4112.20
Jacobian, $a = 0$	1639.26	2120.67	2424.20		
Extended Jacobian, $a = 0$	1715.91	2221.35	2535.91		
Comb scalar multiplications					
Affine, $a = 1$	4372.53	6068.53	8266.65	21140.31	43218.95
Projective, $a = 1$	814.42	1080.55	1708.04	4003.71	5999.15
Jacobian, $a = 1$	807.01	1075.89	1665.27	3737.86	5496.67
Extended Jacobian, $a = 1$	757.47	1010.27	1551.90	3505.59	5163.89
Extended Jacobian, $a = -3$	757.85	1008.40	1549.70	3502.39	5171.99
Jacobian, $a = 0$	2074.50	2677.99	3059.65		
Extended Jacobian, $a = 0$	2221.76	2863.27	3264.95		

6.3.2 Huff Curve Timings

All benchmarks given in this section were measured on an Intel Core i5-2540M running Ubuntu Linux 12.10/amd64 and OpenJDK 7u15/amd64 in server mode. We measured two different timings for our implementation. First, we measured the timings of one application of the *Montgomery ladder*. This was done for two different coordinate systems on the standardized *binary* NIST curves. We used XZ-coordinates for a *differential Montgomery ladder* on the Weierstrass curves and WZ-coordinates, as stated in Section 6.2, for the Huff curve *differential Montgomery ladder*. We took these measurements for two different scenarios, a fixed point P and a randomly chosen point P. One should keep in mind that all timings include all necessary operations to retrieve the result. This includes the y-coordinate recovery, and in the case of WZ-coordinates, all necessary mappings from and to Huff curves. All timings are given in Table 6.6.

Table 6.6: Timings of the Montgomery Ladder for WZ and XZ coordinates using binary NIST curves (taken from [GH13, Table 3])

Coordinate Type	`B-163` [ms]	`B-233` [ms]	`B-283` [ms]	`B-409` [ms]	`B-571` [ms]
XZ	0.709	1.315	1.896	4.203	8.403
WZ	0.692	1.251	1.826	4.040	8.143
Speedup	**2.46%**	**5.12%**	**3.83%**	**4.03%**	**3.19%**
WZ (P fixed)	0.662	1.224	1.778	3.928	8.039
Speedup	**7.10%**	**7.43%**	**6.64%**	**7.00%**	**4.53%**

We looked more deeply into the performance improvements, and validated our results by retrieving additional measurement data. This data gives a detailed explanation of the saved costs for our implementation.

Table 6.7: Costs of squarings and of multiplications with curve parameters in relation to ordinary multiplications (taken from [GH13, Table 4])

	$\mathbb{F}_{2^{163}}$	$\mathbb{F}_{2^{233}}$	$\mathbb{F}_{2^{283}}$	$\mathbb{F}_{2^{409}}$	$\mathbb{F}_{2^{571}}$
$\mathbf{1S} =$	$0.094M$	$0.080M$	$0.077M$	$0.061M$	$0.055M$
$\mathbf{1m_8} =$	$0.369M$	$0.411M$	$0.387M$	$0.418M$	$0.430M$
$\sum =$	$\mathbf{0.463M}$	$\mathbf{0.491M}$	$\mathbf{0.464M}$	$\mathbf{0.479M}$	$\mathbf{0.485M}$

We measured the *relative* costs of a *binary field squaring* $\mathbf{S}$ and a *precomputed binary field multiplication* $\mathbf{M_8}$, compared to a *binary field multiplication* $\mathbf{M}$. The motivation is that with WZ-coordinates, compared to XZ-coordinates, one trades $\mathbf{1M}$ for $\mathbf{1S} + \mathbf{1M_8}$ per *Montgomery ladder step*. This shows that for our implementation, per bit of scalar up to $\mathbf{0.54M}$ are saved. This also illustrates the necessity of fast formulas for recovering the y-coordinate and the additional mapping from and to Huff curves. Otherwise the obtained speedup is diminished by the mapping and coordinate recovery costs. Given the all-in-one back-and-forth conversion formulas with implicit y-coordinate recovery for Huff curves, we would like to emphasize the following: The implementation effort for the faster Huff curve backed WZ-coordinate *differential Montgomery ladder* is virtually the same as for a *differential Montgomery ladder* on Weierstrass curves.

6.4 Summary

In this chapter, we gave detailed information on our implementation efforts. We showed interesting decisions made while implementing the scalar multiplication method implementations, and some of the algorithms used. Later on, we showed how our *differential Montgomery ladder* on Huff curves works. We introduced decomposed formulas for mapping points on Weierstrass curves to points on Huff curves and vice versa. In the subsequent section, we gave benchmark results for all our implementations. We compared several competing scalar multiplication methods, and showed the importance of fast and efficient mapping formulas for the *Huff curve differential Montgomery ladder*. Our *Huff curve differential Montgomery ladder* implementation achieved a speedup of up to 7.4% compared to our implementation of the standard XZ-coordinate *differential Montgomery* *ladder* on Weierstrass curves. We showed that with our WZ-coordinate *differential Montgomery ladder* implementation up to **0.54** *binary scalar multiplications* are saved per bit of the scalar.

Chapter 7

Conclusions

In this thesis, we focused on elliptic curve cryptography in general, and on scalar multiplication methods in particular. Scalar multiplication is heavily utilized in *elliptic curve public-key cryptography* and dominates the execution time of most *elliptic curve public-key cryptography* algorithms. Faster scalar multiplication methods thus benefit a wide range of ECC schemes and are vital for achieving competitive performance.

This thesis started with Chapter 2, where we gave the necessary theoretical background of the mathematical concepts upon which ECC is build on. In Chapter 3 we presented standard elliptic curves used in ECC, as well as the new *binary Huff curves* which are suitable for cryptographic purposes. Additionally we explained known attacks on ECC. This chapter included purely mathematical attacks which influence the choice of elliptic curves available for cryptography. It also included a look into the rather different field of implementation attacks. Over the years, implementation attacks have broken the security of numerous cryptographic scheme implementations. They simply cannot be disregarded when implementing cryptographic software solutions.

In Chapter 4, we focused on scalar multiplication methods and discussed several approaches, beginning with several high performance scalar multiplication methods which use precomputations. We focused on fixed-base comb multiplication methods which provide especially good performance. A different avenue for optimizations was introduced by a technique that speeds up scalar multiplication on elliptic curves with efficiently computable endomorphisms. The third class of scalar multiplication methods we investigated in detail was highly regular and memory efficient scalar multiplication methods, namely the *Montgomery ladder* and *Joye's Double-and-Add* method. In this context, we implemented *binary Huff curves* and a *differential Montgomery ladder* scalar multiplication method on said curves. Additionally, we stated all-in-one, back-and-forth conversion formulas with included y-coordinate recovery for *differential Montgomery ladders* on *Huff curves.* Furthermore, we explicitly stated curve parameters of the *binary Huff curves* corresponding to NIST curves.

In Chapter 5, we detailed the environment for cryptography related implementations in Java. As we implemented all our scalar multiplication algorithms in Java, this chapter gives additional context for the following results chapter. Finally, in Chapter 6 we presented the results of our research, complemented by timing and benchmarking of our implementations. We showed that our *differential Huff curve Montgomery ladder* implementation is up to 7.4% faster than our implementation of the fastest known *Montgomery ladder* formulas known up to that point. Additionally, we gave timing and implementation details for some implemented scalar multiplication methods.

7.1 Future Work

For future work, it would be interesting to further investigate different aspects of *Huff curves* in detail. Most noteworthy are the *unified point addition* formulas for *binary Huff curves* and the evaluation of pairings on *Huff curves* over non-binary finite fields.

Furthermore, it is always of interest to find new speedups for scalar multiplication methods in general, and *regular* scalar multiplication methods in particular. Speeding up the *Montgomery ladder* formulas has significant practical impact as it is a widely deployed, proven and tested concept. We believe that additional implementation specific speedups can be gained by further merging the combination of *finite field level* optimizations with fast *elliptic curve models*.

Appendix A

Definitions

A.1 Abbreviations

API	application programming interface
CRT	Chinese remainder theorem
DBL	Co-Z doubling
DLP	discrete logarithm problem
D-H	Diffie-Hellman
ECC	elliptic curve cryptography
ECDLP	elliptic curve discrete logarithm problem
ECDSA	elliptic curve digital signature algorithm
GNFS	general number field sieve
IAIK	Institute for Applied Information Processing and Communications
IAIK-JCE	IAIK Java Cryptography Extension
IFC	integer factorisation cryptography
JCA	Java Cryptography Architecture
JCE	Java Cryptography Extension
MLS	*Montgomery ladder* step
MOV	Menezes-Okamoto-Vanstone attack
NAF	non-adjacent form
$\mathbf{NAF_w}$	width-w non-adjacent form
NIST	National Institute of Standards and Technology
RSA	Rivest-Shamir-Adleman cryptosystem
SPI	service provider interface
SSSA	Semaev-Smart-Satoh-Araki attack
ZADDU	Co-Z addition
ZDAU	Co-Z point double-add with update

A.2 Used Symbols

$\exists x$	existential quantification
$\forall x$	universal quantification
$x \in G$	x is part of a set G
$\sqrt{x}$	square root of a real value or a field element x
$\lvert x\rvert$	absolute value of the real number x
$a_1, \ldots, a_6$	Weierstrass coefficients
$x + y$	addition of real values or field elements or polynomials
$x \cdot y$	multiplication of real values or field elements
x^y	exponentiation of real values
$x \bmod y$	the reminder of the division $\frac{x}{y}$
$char(F)$	characteristic of field F
$deg(f)$	degree of polynomial f
E	an elliptic curve in Weierstrass form
H	an elliptic curve in Huff form
$E(F)$	a set of points on field F, defined by curve E, plus the point at infinity
$\Delta(E)$	discriminant of curve E
$(f \circ g)(x)$	composition of functions, equal to $f(g(x))$
F	a field F
F^*	the multiplicative group of field F
$\mathbb{F}_q$	finite field F with q elements
$\langle g\rangle$	group generated by generator g
$F_1 \simeq F_2$	group/field F_1 and F_2 are isomorphic
$k!$	factorial of integer k
$\mathbb{N}$	all positive numbers i.e. $\{1, 2, 3, \ldots\}$
$\mathbb{N}_0$	all non-negative numbers i.e. $\{0, 1, 2, 3, \ldots\}$
$\left(\begin{array}{c} q \\ n \end{array}\right)$	binomial coefficient, i.e., $\left(\begin{array}{c} q \\ n \end{array}\right) = \frac{q!}{n!(q-n)!}$
$\mathcal{O}$	point at infinity/neutral element of an elliptic curve
G	a group G
$\lvert G\rvert$	order of group G
$\lvert g\rvert$	order of group element g
(x, y)	an *affine* point on a Weierstrass curve
$(X : Y : Z)$	a *projective* point on a Weierstrass curve
$\mathbb{P}$	set of all prime numbers
$\mathbb{Z}$	ring of integers
$\mathbb{Z}_m$	ring of integers modulo m

Appendix B

Sage Source Code

```
#!/usr/bin/env sage

import sys
from collections import defaultdict

# Implements all necessary calculations to obtain generalized
# binary Huff curve parameters. See Section 5 in [1].
#
# [1] Julien Devinge and Marc Joye, Binary Huff Curves
#
class CalcHuffParameters(object):

  # constructor
  #
  # curveName ... the NIST name e.g., NIST-B163
  # F           ... underlying finite field
  #
  def __init__(self, curveName, F):
    self.curveName = curveName
    self.F = F

  # transforms hex values into field elements
  #
  # b ... a hex value
  #
  def getElement(self, b):
    return getElement(b, self.F)

  # calculates the Huff curve parameter f
  #
  # aTwo ... a_2 ellitic curve parameter
  # aSix ... a_6 ellitic curve parameter
  #
  def calculateF(self, aTwo, aSix):
    check = aTwo.trace()
    tmp = aSix.nth_root(8)
    i = 1

    while True:
        f = self.getElement(i)
        #checks: TR(f*a_6.nth_sqrt(8) == 0
        if ((f * tmp).trace() == 0):
            #checks: TR(a_2) == TR(f^-1)
```

```
          if ((f^-1).trace() == check):
              break
      i += 1
  return i

# calculates the auxiliary Huff parameter s
#
# aTwo ... a_2 ellitic curve parameter
# f    ... f Huff curve parameter
def calculateS(self, aTwo, f):
  R.<s> = PolynomialRing(self.F, 's')
  f = (s^2 + s + aTwo + (f^-2))
  return f.roots()

# solves the equation t^2 + 1/(f^4 * a6.sqrt()) + 1 = 0
#
# aSix ... a_6 elliptic curve parameter
# f    ... f Huff curve parameter
#
def calculateT(self, aSix, f):
  R.<x> = PolynomialRing(self.F,'x')
  tmp = 1/(((self.getElement(f))^4) * aSix.sqrt())
  f = (x^2 + tmp*x +1)
  return f.roots()

# calculates Huff parameter a and b
#
# ts ... t, s Huff curve parameters
#
def findSolutionAB(self, ts):
  # highestDegreeSolution, there are two solutions for
  # NIST curves and the tuple is sorted by degree.
  # therefore, the second one has the higher degree
  t = ts[1][0]
  tSqrt = t.sqrt()
  return 1, tSqrt

# prints the calculated Huff curve parameters in hex
#
# f  ... the Huff parameter f
# ab ... array containg Huff parameters a and b
# s  ... the Huff parameter s
#
def printResults(self, f, ab, s):
  print "=========== " + self.curveName + " ==============="
  print "f: " + hex(f)
  print "a: " + hex(ab[0])
  print "b: " + hex(ab[1].integer_representation())
  print "s: " + hex(s[0][0].integer_representation())
  print "========================================="

# calculates all generalized huff parameters and
# calls the print method afterwards
#
# aTwo ... a_2 elliptic curve parameter
# aSix ... a_6 elliptic curve parameter
#
def calculateAndPrintHuffParams(self, aTwo, aSix):
  f = self.calculateF(aTwo, aSix)
```

```
    t = self.calculateT(aSix, f)
    ab = self.findSolutionAB(t)
    s = self.calculateS(aTwo, self.getElement(f))
    self.printResults(f, ab, s)


# transforms hex values into field elements
#
# b ... a hex value
# g ... generate of underlying field
#
def getElement(b, F):
  j = 0
  x = 0
  g = F.gen()
  for i in b.bits():
    if (i == 1):
      x += g^j
    j += 1
  return F(x)

# adds all the required parameters into a dictionary
#
def getNISTCurveParameters():
  F1.<t> = GF(2)[]

  # NIST B-163
  F.<g> = GF(2^163, name='g', modulus=t^163 + t^7 + t^6 + t^3 + 1)
  aSix = getElement(0x020A601907B8C953CA1481EB10512F78744A3205FD, F)
  aTwo = getElement(0x1, F)
  nistB163 = {"F": F, "aTwo": aTwo, "aSix": aSix}

  # NIST B-233
  F.<g> = GF(2^233, name='g', modulus=t^233 + t^74 + 1)
  # reformating to accomodate large number
  aSixString = ("0x0066647EDE6C332C7F8C0923BB58213B333B20E9CE4281FE"
  "115F7D8F90AD")
  aSix = getElement(Integer(aSixString), F)
  aTwo = getElement(0x1, F)
  nistB233 = {"F": F, "aTwo": aTwo, "aSix": aSix}

  # NIST B-283
  F.<g> = GF(2^283, name='g', modulus=t^283 + t^12 + t^7 + t^5 + 1)
  # reformating to accomodate large number
  aSixString = ("0x027B680AC8B8596DA5A4AF8A19A0303FCA97FD7645309FA2"
  "A581485AF6263E313B79A2F5")
  aSix = getElement(Integer(aSixString), F)
  aTwo = getElement(0x1, F)
  nistB283 = {"F": F, "aTwo": aTwo, "aSix": aSix}

  # NIST B-409
  F.<g> = GF(2^409, name='g', modulus=t^409 + t^87 + 1)
  # reformating to accomodate large number
  aSixString = ("0x0021A5C2C8EE9FEB5C4B9A753B7B476B7FD6422EF1F3DD67"
  "4761FA99D6AC27C8A9A197B272822F6CD57A55AA4F50AE317B13545F")
  aSix = getElement(Integer(aSixString), F)
  aTwo = getElement(0x1, F)
  nistB409 = {"F": F, "aTwo": aTwo, "aSix": aSix}

```

```
    # NIST B-571
    F.<g> = GF(2^571, name='g', modulus=t^571 + t^10 + t^5 + t^2 + 1)
    # reformating to accomodate large number
    aSixString = ("0x02F40E7E2221F295DE297117B7F3D62F5C6A97FFCB8CEFF1"
    "CD6BA8CE4A9A18AD84FFABBD8EFA59332BE7AD6756A66E294AFD185A78FF12AA"
    "520E4DE739BACA0C7FFEFF7F2955727A")
    aSix = getElement(Integer(aSixString), F)
    aTwo = getElement(0x1, F)
    nistB571 = {"F": F, "aTwo": aTwo, "aSix": aSix}

    # NIST K-163
    F.<g> = GF(2^163, name='g', modulus=t^163 + t^7 + t^6 + t^3 + 1)
    aSix = getElement(0x1, F)
    aTwo = getElement(0x1, F)
    nistK163 = {"F": F, "aTwo": aTwo, "aSix": aSix}

    # NIST K-233
    F.<g> = GF(2^233, name='g', modulus=t^233 + t^74 + 1)
    aSix = getElement(0x1, F)
    aTwo = getElement(0x0, F)
    nistK233 = {"F": F, "aTwo": aTwo, "aSix": aSix}

    # NIST K-283
    F.<g> = GF(2^283, name='g', modulus=t^283 + t^12 + t^7 + t^5 + 1)
    aSix = getElement(0x1, F)
    aTwo = getElement(0x0, F)
    nistK283 = {"F": F, "aTwo": aTwo, "aSix": aSix}

    # NIST K-409
    F.<g> = GF(2^409, name='g', modulus=t^409 + t^87 + 1)
    aSix = getElement(0x1, F)
    aTwo = getElement(0x0, F)
    nistK409 = {"F": F, "aTwo": aTwo, "aSix": aSix}

    # NIST K-571
    F.<g> = GF(2^571, name='g', modulus=t^571 + t^10 + t^5 + t^2 + 1)
    aSix = getElement(0x1, F)
    aTwo = getElement(0x0, F)
    nistK571 = {"F": F, "aTwo": aTwo, "aSix": aSix}

    curveParams = {}
    curveParams.update({"B-163": nistB163})
    curveParams.update({"B-233": nistB233})
    curveParams.update({"B-283": nistB283})
    curveParams.update({"B-409": nistB409})
    curveParams.update({"B-571": nistB571})

    curveParams.update({"K-163": nistK163})
    curveParams.update({"K-233": nistK233})
    curveParams.update({"K-283": nistK283})
    curveParams.update({"K-409": nistK409})
    curveParams.update({"K-571": nistK571})

    return curveParams

# calculate Huff curve parameters
# using the following classes:
# CalcHuffParameters and ParameterStorage
#
```

```
def main():

  curveParams = getNISTCurveParameters()

  for curveName, params in curveParams.items():
    calc = CalcHuffParameters(curveName, params["F"])
    calc.calculateAndPrintHuffParams(params["aTwo"], params["aSix"])

if __name__ == "__main__":
    main()
```

Listing B.1: Sage code for calculating generalized binary Huff curve parameters

Bibliography

[ABM$^+$03] Adrian Antipa, Daniel R. L. Brown, Alfred Menezes, René Struik, and Scott A. Vanstone. Validation of Elliptic Curve Public Keys. In Yvo Desmedt, editor, *Public Key Cryptography - PKC 2003, 6th International Workshop on Theory and Practice in Public Key Cryptography, Proceedings*, volume 2567 of *Lecture Notes in Computer Science*, pages 211–223. Springer, 2003.

[BB05] David Brumley and Dan Boneh. Remote timing attacks are practical. *Computer Networks*, 48(5):701–716, 2005.

[BBB$^+$12] Elaine Barker, William Barker, William Burr, William Polk, and Miles Smid. NIST Special Publication 800-57, Recommendation for Key Management Part 1: General (Revision 3), 2012.

[BL14a] Daniel J. Bernstein and Tanja Lange. SafeCurves: choosing safe curves for elliptic-curve cryptography. `http://safecurves.cr.yp.to`, 2014. Accessed: 2014-05-15.

[BL14b] Daniel J. Bernstein and Tanja Lange. Explicit-Formulas Database. `http://www.hyperelliptic.org/EFD/`, 2014. Accessed: 2014-04-17.

[Bro06] Daniel R. L. Brown. Conjectured Security of the ANSI-NIST Elliptic Curve RNG. Cryptology ePrint Archive, Report 2006/117, 2006. `http://eprint.iacr.org/`.

[BSS99] Ian F. Blake, G. Seroussi, and N. P. Smart. *Elliptic curves in cryptography.* Cambridge University Press, New York, NY, USA, 1999.

[BSS05] I.F. Blake, G. Seroussi, and N.P. Smart. *Advances in Elliptic Curve Cryptography.* London Mathematical Society Lecture Note Series. Cambridge University Press, 2005.

[BT11] Billy Bob Brumley and Nicola Tuveri. Remote Timing Attacks Are Still Practical. In Vijay Atluri and Claudia Díaz, editors, *Computer Security - ESORICS 2011 - 16th European Symposium on Research in Computer Security, Proceedings*, volume 6879 of *Lecture Notes in Computer Science*, pages 355–371. Springer, 2011.

[CC86] D V Chudnovsky and G V Chudnovsky. Sequences of numbers generated by addition in formal groups and new primality and factorization tests. *Adv. Appl. Math.*, 7(4):385–434, December 1986.

[CF05] Henri Cohen and Gerhard Frey, editors. *Handbook of elliptic and hyperelliptic curve cryptography.* CRC Press, 2005.

[cKKP99] Çetin Kaya Koç and Christof Paar, editors. *Cryptographic Hardware and Embedded Systems, First International Workshop, CHES'99, Worcester, MA, USA, August 12-13, 1999, Proceedings*, volume 1717 of *Lecture Notes in Computer Science.* Springer, 1999.

[Cor99] Jean-Sébastien Coron. Resistance against Differential Power Analysis for Elliptic Curve Cryptosystems. In Çetin Kaya Koç and Paar [cKKP99], pages 292–302.

[DH76] Whitfield Diffie and Martin E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, 1976.

[Die12] Claus Diem. What on earth is "index calculus"? `http://ellipticnews.wordpress.com/2012/05/07/246/`, 2012. Accessed: 2014-05-15.

[DJ11] Julien Devigne and Marc Joye. Binary huff curves. In Aggelos Kiayias, editor, *Topics in Cryptology - CT-RSA 2011 - The Cryptographers' Track at the RSA Conference, Proceedings*, volume 6558 of *CT-RSA'11*, pages 340–355, Berlin, Heidelberg, 2011. Springer-Verlag.

[DM89] International Business Machines Corporation. Research Division and K.S. McCurley. *The Discrete Logarithm Problem.* Research report. IBM Research Division, 1989.

[Flo67] Robert W. Floyd. Nondeterministic Algorithms. *Journal of the ACM*, 14(4):636–644, 1967.

[FLRV08] Pierre-Alain Fouque, Reynald Lercier, Denis Réal, and Frédéric Valette. Fault Attack onElliptic Curve Montgomery Ladder Implementation. In Luca Breveglieri, Shay Gueron, Israel Koren, David Naccache, and Jean-Pierre Seifert, editors, *Fifth International Workshop on Fault Diagnosis and Tolerance in Cryptography, 2008, FDTC 2008, Proceedings*, pages 92–98. IEEE Computer Society, 2008.

[FMR99] Gerhard Frey, Michael Müller, and Hans-Georg Rück. The Tate pairing and the discrete logarithm applied to elliptic curve cryptosystems. *IEEE Transactions on Information Theory*, 45(5):1717–1719, 1999.

[FR94] Gerhard Frey and Hans-Georg Rück. A Remark Concerning M-divisibility and the Discrete Logarithm in the Divisor Class Group of Curves. *Math. Comput.*, 62(206):865–874, April 1994.

[FV03] Pierre-Alain Fouque and Frédéric Valette. The Doubling Attack - *Why Upwards Is Better than Downwards.* In Colin D. Walter, Çetin Kaya Koç, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2003, 5th International Workshop, Cologne, Proceedings*, volume 2779 of *Lecture Notes in Computer Science*, pages 269–280. Springer-Verlag, 2003.

[GH13] Gerwin Gsenger and Christian Hanser. Improving the efficiency of elliptic curve scalar multiplication using binary huff curves. In Alfredo Cuzzocrea, Christian Kittl, Dimitris E. Simos, Edgar Weippl, Lida Xu, Alfredo Cuzzocrea, Christian Kittl, Dimitris E. Simos, Edgar Weippl, and Lida Xu, editors, *Security Engineering and Intelligence Informatics - CD-ARES 2013 Workshops: MoCrySEn and SeCIHD, Proceedings*, volume 8128 of *Lecture Notes in Computer Science*, pages 155–167. Springer, 2013.

[GJM10] Raveen R. Goundar, Marc Joye, and Atsuko Miyaji. Co-Z addition formulae and binary ladders on elliptic curves. In Stefan Mangard and François-Xavier Standaert, editors, *Cryptographic Hardware and Embedded Systems, CHES 2010, 12th International Workshop, Proceedings*, volume 6225 of *CHES'10*, pages 65–79. Springer, 2010.

[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, 21st Annual International Cryptology Conference, Proceedings*, volume 2139 of *CRYPTO '01*, pages 190–200. Springer, 2001.

[Han10] Christian Hanser. New Trends in Elliptic Curve Cryptography. Master Thesis, Graz University of Technology, Institute for Applied Information Processing and Communications, Graz University of Technology, April 2010.

[Han14] Christian Hanser. IAIK ECCelerate™. `http://jce.iaik.tugraz.at/sic/Products/Core-Crypto-Toolkits/ECCelerate`, 2011–2014.

[Hit07] Laura Hitt. On the Minimal Embedding Field. In Tsuyoshi Takagi, Tatsuaki Okamoto, Eiji Okamoto, and Takeshi Okamoto, editors, *Pairing-Based Cryptography - Pairing 2007, First International Conference, Proceedings*, volume 4575 of *Lecture Notes in Computer Science*, pages 294–301. Springer, 2007.

[HJS11] Michael Hutter, Marc Joye, and Yannick Sierra. Memory-Constrained Implementations of Elliptic Curve Cryptography in Co-Z Coordinate Representation. In Abderrahmane Nitaj and David Pointcheval, editors, *Progress in Cryptology - AFRICACRYPT 2011 - 4th International Conference on Cryptology in Africa, Proceedings*, volume 6737 of *AFRICACRYPT'11*, pages 170–187. Springer, 2011.

[HMV04] Darrel Hankerson, Alfred Menezes, and Scott Vanstone. *Guide to Elliptic Curve Cryptography.* Springer Professional Computing. Springer, 2004.

[Huf48] Gerald B. Huff. Diophantine problems in geometry and elliptic ternary forms. *Duke Math. J.*, 15:443–453, 1948.

[IT14] SIC Stiftung Secure Information and Communication Technologies. CRYPTO Toolkit. `http://jce.iaik.tugraz.at/`, 2014. Accessed: 2014-05-03.

[Joy07] Marc Joye. Highly Regular Right-to-Left Algorithms for Scalar Multiplication. In Pascal Paillier and Ingrid Verbauwhede, editors, *Cryptographic Hardware and Embedded Systems - CHES 2007, 9th International Workshop, Proceedings*, volume 4727 of *Lecture Notes in Computer Science*, pages 135–147. Springer, 2007.

[Joy08] Marc Joye. Fast Point Multiplication on Elliptic Curves without Precomputation. In Joachim von zur Gathen, José Luis Imaña, and Çetin Kaya Koç, editors, *Arithmetic of Finite Fields, 2nd International Workshop, WAIFI 2008, Proceedings*, volume 5130 of *Lecture Notes in Computer Science*, pages 36–46. Springer, 2008.

[JTV10] Marc Joye, Mehdi Tibouchi, and Damien Vergnaud. Huff's model for elliptic curves. *IACR Cryptology ePrint Archive*, 2010:383, 2010.

[Kob87] Neal Koblitz. Elliptic curve cryptosystems. *Mathematics of Computation*, 48:203–209, 1987.

[Kob91] Neal Koblitz. Cm-curves with good cryptographic properties. In Joan Feigenbaum, editor, *Advances in Cryptology - CRYPTO '91, 11th Annual International Cryptology Conference, Proceedings*, volume 576 of *Lecture Notes in Computer Science*, pages 279–287. Springer, 1991.

[Koc96] Paul C. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In Neal Koblitz, editor, *Advances in Cryptology - CRYPTO '96, 16th Annual International Cryptology Conference, Proceedings*, volume 1109 of *CRYPTO '96*, pages 104–113. Springer, 1996.

[Kop09] Clemens Koppensteiner. Mathematical Foundations of Elliptic Curve Cryptography. Master Thesis, Vienna University of Technology, Institute of Discrete Mathematics and Geometry, Vienna University of Technology, 2009.

[LD98] Julio Lopez and Ricardo Dahab. Improved Algorithms for Elliptic Curve Arithmetic in $GF(2^n)$. In Stafford E. Tavares and Henk Meijer, editors, *Selected Areas in Cryptography '98, SAC'98, Proceedings*, volume 1556 of *Lecture Notes in Computer Science*, pages 201 – 212. Springer, 1998.

[LD99] Julio López and Ricardo Dahab. Fast Multiplication on Elliptic Curves over GF(2m) without Precomputation. In Çetin Kaya Koç and Paar [cKKP99], pages 316–327.

[LHWL93] Arjen K. Lenstra and Jr. Hendrik W. Lenstra, editors. *The development of the number field sieve*, volume 1554 of *Lecture Notes in Mathematics*. Springer-Verlag, Berlin, 1993.

[LL94] Chae Hoon Lim and Pil Joong Lee. More Flexible Exponentiation with Precomputation. In Yvo Desmedt, editor, *Advances in Cryptology - CRYPTO '94,14th Annual International Cryptology Conference, Proceedings*, volume 839 of *Lecture Notes in Computer Science*, pages 95–107. Springer, 1994.

[LV01] Arjen K. Lenstra and Eric R. Verheul. Selecting cryptographic key sizes. *Journal of Cryptology*, 2001. URL: `http://cr.yp.to/bib/2001/lenstra.ps`.

[LZM09] Mario Lamberger, Volker Ziegler, and Manfred Madritsch. Mathematische Grundlagen der Kryptographie. University Lecture, 2009.

[Mel07] Nicolas Meloni. New Point Addition Formulae for ECC Applications. In Claude Carlet and Berk Sunar, editors, *Arithmetic of Finite Fields, First International Workshop, WAIFI 2007, Proceedings*, volume 4547 of *Lecture Notes in Computer Science*, pages 189–201. Springer, 2007.

[MHH12] Nashwa A. F. Mohammed, Mohsin H. A. Hashim, and Michael Hutter. Improved Fixed-base Comb Method for Fast Scalar Multiplication. In Aikaterini Mitrokotsa and Serge Vaudenay, editors, *Progress in Cryptology - AFRICACRYPT 2012 - 5th International Conference on Cryptology in Africa, Proceedings*, volume 7374 of *Lecture Notes in Computer Science*, pages 342 – 359. Springer, 2012.

[Mil86] Victor S. Miller. Use of Elliptic Curves in Cryptography. In Hugh C. Williams, editor, *Advances in Cryptology - CRYPTO '85,Proceedings*, volume 218 of *Lecture Notes in Computer Science*, pages 417–426. Springer, 1986.

[Mon87] Peter L. Montgomery. Speeding the Pollard and elliptic curve methods of factorization. *Mathematics of Computation*, 48:243–264, 1987.

[MVO91] Alfred Menezes, Scott Vanstone, and Tatsuaki Okamoto. Reducing Elliptic Curve Logarithms to Logarithms in a Finite Field. In Cris Koutsougeras and Jeffrey Scott Vitter, editors, *Proceedings of the 23rd Annual ACM Symposium on Theory of Computing*, STOC '91, pages 80–89. ACM, 1991.

[Nat13] National Institute of Standards and Technology. *FIPS PUB 186-4: Digital Signature Standard (DSS).* National Institute of Standards and Technology, 2013. Available at `http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf`.

[NS03] Phong Q. Nguyen and Igor E. Shparlinski. The insecurity of the elliptic curve digital signature algorithm with partially known nonces. *Des. Codes Cryptography*, 30(2):201–217, September 2003.

[OL09] Elisabeth Oswald and Mario Lamberger. Applied Cryptography 2. University Lecture, 2009.

[Ora14] Oracle. Java™ Cryptography Architecture (JCA) Reference Guide. `http://docs.oracle.com/javase/7/docs/technotes/guides/security/crypto/CryptoSpec.html`, 2014. Accessed: 2014-05-02.

[Riv11] Matthieu Rivain. Fast and regular algorithms for scalar multiplication over elliptic curves. *IACR Cryptology ePrint Archive*, 2011:338, 2011.

[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, February 1978.

[SA98] Takakazu Satoh and Kiyomichi Araki. Fermat quotients and the polynomial time discrete log algorithm for anomalous elliptic curves. *Commentarii Mathematici Universitatis Sancti Pauli*, 47:81–92, 1998.

[Sem98] Igor A. Semaev. Evaluation of discrete logarithms in a group of p-torsion points of an elliptic curve in characteristic p. *Mathematics of Computation*, 67(221):353–356, 1998.

[Sil92] Joseph H. Silverman. *The Arithmetic of Elliptic Curves, volume 106 of Graduate Texts in Mathematics (2nd Edition).* Springer-Verlag New York, 2nd edition 2009, 1992.

[Sma99] Nigel P. Smart. The Discrete Logarithm Problem On Elliptic Curves Of Trace One. *Journal of Cryptology*, 12:193–196, 1999.

[Sma02] Nigel P. Smart. *Cryptography, An Introduction: Third Edition.* Mcgraw-Hill Professional, 2002.

[Sol00] Jerome A. Solinas. Efficient Arithmetic on Koblitz Curves. *Des. Codes Cryptography*, 19(2/3):195–249, 2000.

[Sol01] Jerome A. Solinas. Low-Weight Binary Representations for Pairs of Integers. Technical report, National Security Agency, USA, 2001.

[SOOS95] Richard Schroeppel, Hilarie K. Orman, Sean W. O'Malley, and Oliver Spatscheck. Fast Key Exchange with Elliptic Curve Systems. In Don Coppersmith, editor, *Advances in Cryptology - CRYPTO '95, 15th Annual International Cryptology Conference, Proceedings*, volume 963 of *Lecture Notes in Computer Science*, pages 43–56. Springer, 1995.

[SS06] Berry Schoenmakers and Andrey Sidorenko. Cryptanalysis of the Dual Elliptic Curve Pseudorandom Generator. Cryptology ePrint Archive, Report 2006/190, 2006.

[TC05] Woei-Jiunn Tsaur and Chih-Ho Chou. Efficient algorithms for speeding up the computations of elliptic curve cryptosystems. *Applied Mathematics and Computation*, 168(2):1045–1064, 2005.

[Tes98] Edlyn Teske. Speeding Up Pollard's Rho Method For Computing Discrete Logarithms. In Joe Buhler, editor, *Algorithmic Number Theory, Third International Symposium, ANTS-III, Proceedings*, volume 1423 of *Lecture Notes in Computer Science*, pages 541–554. Springer, 1998.

[VD10] A. Venelli and F. Dassance. Faster Side-Channel Resistant Elliptic Curve Scalar Multiplication. *Arithmetic, Geometry, Cryptography and Coding Theory 2009, Contemporary Mathematics*, 521:29–40, 2010.

[WT02] Lawrence C. Washington and Wade Trappe. *Introduction to Cryptography: With Coding Theory.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 2002.