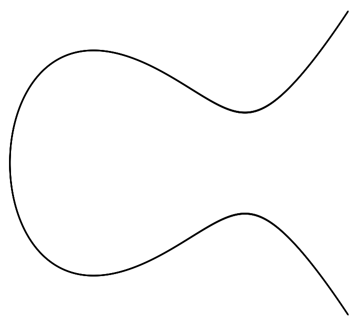
Efficient Scalar Multiplication and Security against Power Analysis in Cryptosystems based on the NIST Elliptic Curves Over Prime Fields

Lars Elmegaard-Fessel*

Joint work with IBM Danmark A/S

August 1, 2006



Supervisors:

Associate Professor Anders Thorup,
Institute for Mathematical Sciences,
University of Copenhagen

Associate Professor Tanja Lange,
Department of Mathematics,
Technical University of Denmark

—◇—

Thesis for the Master degree in Mathematics.
Study board for Mathematical Sciences,
Institute for Mathematical Sciences,
University of Copenhagen

* Supported by Oticon Fonden and Siemensfonden.

This text is printed using Computer Modern 12pt.
Layout by the author using LaTeX.
Graphs and figures are produced using Maple, Xfig and METAPOST.
Printed in Denmark.

To Cathrine

Abstract

In cryptosystems based on elliptic curves over finite fields (ECC-systems), the most time-consuming operation is scalar multiplication. We focus on the NIST elliptic curves over prime fields. An implementation of scalar multiplication, developed by IBM Danmark A/S for test purposes, serves as a point of reference.

In order to achieve maximal efficiency in an ECC-system, one must choose an optimal method for scalar multiplication and the best possible coordinate representation for the curve being used. We perform an analysis of known scalar multiplication methods. This analysis contains a higher degree of detail than existing publications on the subject and shows that the NAF_w scalar multiplication method with precomputations in affine coordinates, intermediate doublings in Jacobian coordinates and additions in mixed coordinates is the optimal choice. We compare our scalar multiplication scheme with the one implemented by IBM and conclude that a substantial improvement of efficiency is achieved by using our scheme. We implement our efficient scheme and support our conclusions with timings of the implementations.

Side channel attacks using power analysis is considered to be a major threat against the security of ECC-systems. Mathematical countermeasures exist but reduce the performance of the system. So far, no comparison of the countermeasures has been published. We perform such a comparison and conclude that if a sufficient amount of storage is available, a combination of side channel atomicity and scalar randomization should be used as a countermeasure. If storage is limited, countermeasures should be based on a combination of Montgomery's ladder algorithm and scalar randomization. We specify side channel atomic elliptic curve operations on the NIST elliptic curves in mixed coordinates. So far, no such specifications have been published. We develop an efficient and secure scalar multiplication scheme and conclude that this scheme is more efficient than the scheme used in the IBM implementation, which provides no security against side channel attacks. We implement our efficient, secure scheme and support our conclusions with timings of the implementations.

Resumé

I kryptosystemer baseret på elliptiske kurver over endelige legemer (ECC-systemer) er den mest omkostningsfulde operation skalarmultiplikation. Vi fokuserer på NIST elliptiske kurver over endelige legemer $\mathbb{F}_p$, hvor p er et primtal. En implementation af skalarmultiplikation udviklet af IBM Danmark A/S til testformål tjener som sammenligningsgrundlag.

For at opnå en maksimal grad af effektivitet i et ECC-system skal man vælge en optimal metode til skalarmultiplikation og den bedst mulige koordinat-repræsentation af den anvendte kurve. Vi gennemfører en analyse af kendte metoder til skalarmultiplikation. Denne analyse indeholder en højere detaljeringsgrad end eksisterende publikationer indenfor emnet og viser, at NAF_w metoden til skalarmultiplikation med præ-beregninger i affine koordinater, mellemliggende fordoblinger i Jakobianske koordinater og additioner i blandede koordinater er det optimale valg. Vi sammenligner vores metode med den af IBM anvendte og konkluderer, at en betydelig effektivitetsforøgelse opnås ved at anvende vores metode. Vi implementerer vores effektive metode og understøtter vores konklusioner med tidsmålinger af implementationerne.

Såkaldte *side channel* angreb baseret på strøm-analyse betragtes som en alvorlig trussel mod ECC-systemers sikkerhed. Matematiske modtræk eksisterer men påvirker systemets ydeevne negativt. Hidtil er ingen sammenligning af modtrækkene blevet offentliggjort. Vi gennemfører en sådan sammenligning og konkluderer, at hvis en tilstrækkelig mængde hukommelse er til rådighed, bør en kombination af side channel atomisme og tilfældigt skalar anvendes som modtræk. Hvis mængden af hukommelse er begrænset, bør man anvende et modtræk bestående af Montgomery's stige-algoritme og tilfældigt skalar. Vi specificerer side channel atomiske operationer på NIST elliptiske kurver i blandede koordinater. Sådanne specifikationer er ikke tidligere blevet offentliggjort. Vi udvikler en effektiv og sikker metode til skalarmultiplikation og konkluderer, at denne metode er mere effektiv end metoden der anvendes i IBM-implementationen, som ikke er sikret mod side channel angreb. Vi implementerer vores effektive, sikre algoritme og understøtter vores konklusioner med tidsmålinger af implementationerne.

Contents

Preface

This text is a thesis for the master degree in mathematics at the University of Copenhagen. It was produced in the period February-July 2006. The project proposal for the thesis was established in collaboration with IBM Danmark A/S.

The prerequisite for reading the thesis is basic mathematical knowledge corresponding to what is held by graduate students of mathematics. However, in order to ease the understanding of the motivation for using cryptosystems based on elliptic curves, basic knowledge of cryptography (such as the principles behind RSA and the discrete logarithm problem) is recommended.

The thesis contains a report and a collection of implementations of algorithms, for which commented Java source code is enclosed. The details of the implementations are in the report, and anyone with a programming background corresponding to the level presented at introductory programming courses should be able to understand the code.

To avoid confusion with regular text, the end of all definitions and examples are marked with $\circ$ (except when the definition/example is the last part of a section or appears immediately before another environment). The end of proofs are marked with $\blacksquare$.

The author would like to thank A. Thorup at the University of Copenhagen and T. Lange at The Technical University of Denmark for competent supervision, prompt answers of my many queries and for commenting on various portions of the manuscript. All mistakes or problems remaining in the text are my own, and I apologize in advance for any such you may find. I would also like to thank M. Clausen and L. Moesgaard at IBM Danmark A/S. for allocating time and resources to my project, commenting on my work and answering numerous questions. Thanks are due also to I. Kiming, A. Thorup and F. Topsøe at the University of Copenhagen for their assistance with my applications for grants. The author would like to thank Oticon Fonden and Siemensfonden for believing in my project. I would also like to thank my parents for their support and my brother N. Elmegaard-Fessel for his inputs during our conversations. Last, but certainly not least, I thank my beloved wife Cathrine for many valuable comments on the manuscript and for her priceless encouragement during the writing of the thesis.

Due to copyright considerations Section 6.1.3 (pages 87-95) is excluded from the publicly available version of this report. Section 6.1.3 has, however, been made available to the parties involved in grading the thesis.

Copenhagen, July 2006

Introduction

Today, most public key cryptosystems are based on the use of RSA. The advances in information technology during recent years has resulted in a demand for longer RSA keys, in order to uphold an acceptable level of security. At the time of writing (July 2006), RSA Security recommends[1] a key size of 1024 bits for corporate use and 2048 bits for extremely valuable keys, e.g. the root key pair for a certifying authority. The need for long keys makes systems based on RSA difficult to implement in devices with constrained memory and/or processing power, e.g. smart cards.

As an alternative to using RSA, one can construct public key cryptosystems based on the discrete logarithm problem (DLP) in a finite abelian group G. The DLP is: Given $g \in G$ and $g^x \in G$, determine x. The group is most commonly taken to be $\mathbb{F}_q^*$, where $q = p^n$ for a prime p and a positive integer n. There exists, however, sub-exponential methods (e.g. the Pohlig-Hellmann algorithm and the "Index Calculus" algorithm by Adlemann and Western, and Miller) for solving the DLP in $\mathbb{F}_q^*$ (see [Kob94] or [BSS99]). For many purposes, the q being used, therefore, has to be very large in order to uphold a sufficient level of security. These large values of q imply a large storage requirement and a need for high processing power, so, like cryptosystems based on RSA, cryptosystems based on the DLP in $\mathbb{F}_q^*$ are often not suitable for implementation in devices with limited resources.

Miller [Mil85] and Koblitz [Kob87] has suggested the use of elliptic curves in cryptography. Their proposal was to use cryptosystems based on the DLP in a group constructed from the points on an elliptic curve over a finite field. In this setting the DLP is called the *Elliptic Curve Discrete Logarithm Problem* (ECDLP). There is no known direct analog of the "Index Calculus" algorithm for attacks on systems based on the ECDLP, and, by choosing suitable system parameters, one can achieve a group order equal to a large prime number (the meaning of "large" is determined by the desired strength of the system). This makes attacks based on the Pohlig-Hellmann algorithm infeasible. These properties make it possible to construct an Elliptic Curve Cryptosystem (ECC-system) which offers the same level of security as "conventional" systems (based on RSA or the DLP in $\mathbb{F}_q^*$) and uses shorter keys. In [RY97] Robshaw and Yin estimate that

[1] Recommendations are published at `http://www.rsasecurity.com/rsalabs/`.

an ECC-system using a 160 bit key potentially offers the same level of security as a conventional system using a 1024 bit key. Similar conclusions can be found in the recommendations by The National Institute of Standards and Technology (NIST) [NIS06], The European Network of Excellence for Cryptology (ECRYPT) [ECR05], and Lenstra and Verheul [LV00].

All cryptographic schemes based on the DLP in $\mathbb{F}_q^*$ have an elliptic curve analog. We will focus on the Digital Signature Algorithm (DSA) and the ElGamal cryptosystem. The elliptic curve analog of the DSA is the Elliptic Curve DSA (ECDSA), described in [X9.98]. The ElGamal cryptosystem is not standardized (partially due to certain security issues). Instead, one uses the Elliptic Curve Integrated Encryption Scheme ([P1300]). For our purposes, the ElGamal cryptosystem will, however, suffice. Using the time required to perform a 1024 bit modular multiplication as a time unit, Robshaw and Yin [RY97] compare the time required by a 160 bit ECC-system, a 1024 bit RSA cryptosystem and a 1024 bit DLP cryptosystem to perform an encryption, a decryption, a signing and a signature verification. Their figures show that for decryption and signing the ECC-system is four times faster than the cryptosystem based on the DLP. It is more than six times faster than the cryptosystem based on RSA. The RSA cryptosystem is the fastest when doing encryption and signature verification[2].

The possibility to maintain an unchanged level of security while using shorter keys makes ECC-systems interesting for use in smart cards and similar devices. Also, key generation in ECC-systems is simple, as it only involves choosing a random positive integer in a fixed interval, whereas key generation in an RSA cryptosystem involves primality testing of large numbers, which is very time consuming. Due to this, elliptic curves have received a lot of attention during recent years. However, not all elliptic curves are equally secure for use in ECC-systems (see [BSS99]). NIST has selected a number of elliptic curves (NIST curves) over finite fields which are considered to be safe for use in cryptographic applications. We will focus on a selection of the NIST curves in the sequel. The text is divided into five main parts:

Part I: In Part I we present a brief introduction to the theory of elliptic curves and the use of elliptic curves in cryptography. Also, we specify the details of the NIST curves.

Part II: The most time-consuming operation performed in an ECC-system is the so-called *scalar multiplication*. When implementing an ECC-system, one has to make two important choices. The first one is which method to use for scalar multiplication. The second one is which coordinate representation to use for the elliptic curve being used. The efficiency of the system depends heavily on these

[2] It should be taken into consideration that Robshaw and Yin provide very little information about the degree of optimization performed on the systems.

choices. We are presented with a Java implementation of a scalar multiplication scheme, developed by IBM Danmark A/S for test purposes. Part II deals with the task of constructing a scheme which is more efficient than the one implemented by IBM. We examine a number of known scalar multiplication methods in order to find the most efficient one. Subsequently, we perform an evaluation of the use of different coordinate representations. As the choice of an optimal representation depends on the specific computational environment (processor power, memory, available software etc.), we make an optimal choice based on the computational environment at hand. We implement our resulting scalar multiplication scheme and document the efficiency of our scheme both theoretically (counting the number of required operations to be performed in the ground field) and empirically (documenting timings of our implementation). The test implementation developed by IBM will serve as a point of reference, when evaluating the efficiency of our scheme.

Part III: A technique for doing cryptanalysis known as *side channel analysis* has become a threat to many types of cryptosystems. Attacks based on this technique are known as *side channel attacks*. These attacks have drawn much attention, since Paul Kocher [KJJ99] described the first attack of its kind in 1999. Coron [Cor99] transferred the idea to ECC-systems. Mathematical countermeasures against side channel attacks on ECC-systems exist, but implementing these countermeasures affects the performance of the system. So far, no comparisons between the efficiencies of known mathematical countermeasures against side channel attacks have been published. In Part III we perform such a comparison. We evaluate both the efficiency and security of a number of known countermeasures. Implementations of all countermeasures are developed, and timings of the implementations are documented. Based on our comparison, we select countermeasures which introduce the smallest possible performance reduction. The countermeasures are used to construct a scalar multiplication scheme which is secure against side channel attacks using power analysis. We compare the efficiency of our secure scheme to the efficiency of our original scheme as well as to the efficiency of the scheme implemented by IBM, which offers no security against side channel attacks. Our secure scheme is implemented, and timings of the implementations are documented.

Parts IV & V: In Part IV we draw conclusions based on the results obtained in Part II and Part III. In Part V (appendix) we enclose an introduction to the theory of Markov chains, as results from this theory are used in connection with analyzing scalar multiplication algorithms. Also, we enclose test vectors and source code for all implementations developed.

Part I

Elliptic Curves

Chapter 1

Arithmetic on Elliptic Curves

Elliptic curves are not ellipses. The study of elliptic curves arose from calculating arc lengths on ellipses which leads to so-called *elliptic integrals* of the form

$$\int \frac{dx}{\sqrt{4x^3 - g_2x - g_3}}.$$

By evaluating this integral for suitable complex numbers g_2 and g_3, one can find complex numbers ω_1 and ω_2 which are linearly independent over $\mathbb{R}$. These numbers, called *periods*, are used to define the lattice

$$L = \mathbb{Z}\omega_1 + \mathbb{Z}\omega_2 = \{n_1\omega_1 + n_2\omega_2 \,|\, n_1, n_2 \in \mathbb{Z}\}.$$

A meromorphic function is given by

$$\wp(u) = \frac{1}{u^2} + \sum_{\substack{\omega \in L \\ \omega \neq 0}} \left(\frac{1}{(u-\omega)^2} - \frac{1}{\omega^2} \right).$$

The function $\wp$ is called the *Weierstraß* $\wp$ *function*. It is doubly periodic and satisfies the differential equation

$$(\wp')^2 = 4\wp^3 - g_2\wp - g_3,$$

so for every $u \in \mathbb{C}$ we get a point $(x, y) = (\wp(u), \wp'(u))$ which satisfies the equation

$$y^2 = 4x^3 - g_2x - g_3.$$

Equations of this form define elliptic curves over $\mathbb{C}$, and every elliptic curve over a field of characteristic different from 2 and 3 can be defined by an equation of this form.

This section presents a brief introduction to the theory of elliptic curves. The presentation is not an exhaustive examination, as only a sparse selection of the aspects of the theory is presented. The section is self-contained, as far as our need for an applied introduction to the theory goes, but readers interested in the vast field of elliptic curves will benefit from the introductions found in [Sil92] and [ACD$^+$05].

1.1 General Definitions

Let $\mathbf{K}$ be a field, let $\mathbf{K}[X]$, $\mathbf{K}[X,Y]$ and $\mathbf{K}[X,Y,Z]$ be the polynomial rings over $\mathbf{K}$ in one,two and three variables respectively. Let $f \in \mathbf{K}[X,Y]$. Then, f can be written as a finite sum

$$f(x,y) = \sum_{i,j} a_{i,j} x^i y^j, \quad a_{i,j} \in \mathbf{K}. \tag{1.1}$$

If $f \neq 0$, the *degree* of f is $\deg(f) = \max\{i + j \mid a_{i,j} \neq 0\}$.

An element $F \in \mathbf{K}[X,Y,Z]$ is said to be *homogeneous* of degree d if

$$F(X,Y,Z) = \sum_{\substack{i,j,k \\ i+j+k=d}} b_{i,j,k} X^i Y^j Z^k, \quad b_{i,j,k} \in \mathbf{K}.$$

The *homogenization* of f in equation (1.1), where $f \neq 0$, is a homogeneous polynomial $F \in \mathbf{K}[X,Y,Z]$ of degree $\deg(f)$ given by

$$F(X,Y,Z) = \sum_{i,j} a_{i,j} X^i Y^j Z^{deg(f)-i-j}.$$

Let F be the homogenization of f, and consider the equation

$$F(X,Y,Z) = 0. \tag{1.2}$$

Equation (1.2) has solutions $(x,y,1)$, where (x,y) is a solution of $f(x,y) = 0$. If $(X,Y,Z) \in \mathbf{K}^3$ is a solution of equation (1.2), then so is $(\lambda X, \lambda Y, \lambda Z)$ for any $\lambda \in \mathbf{K}^*$ (as F is homogeneous). We introduce an equivalence relation $\sim$ on $\dot{\mathbf{K}} := \mathbf{K}^3 \setminus \{(0,0,0)\}$ by

$$\begin{aligned}&(X,Y,Z) \sim (X',Y',Z') \text{ if} \\ &\exists \lambda \in \mathbf{K}^* \; : X = \lambda X' \wedge Y = \lambda Y' \wedge Z = \lambda Z'.\end{aligned}$$

The quotient space $\dot{\mathbf{K}}/\sim$ is called the *projective plane* over $\mathbf{K}$ and is denoted $\mathbb{P}^2(\mathbf{K})$ (or simply $\mathbb{P}^2$), while $\mathbf{K}^2$ is called the *affine plane* over $\mathbf{K}$ and is denoted $\mathbb{A}^2(\mathbf{K})$ (or simply $\mathbb{A}^2$). A point in $P \in \mathbb{P}^2(\mathbf{K})$ is thus an equivalence class. We write $P = (X : Y : Z)$ for the equivalence class containing (X,Y,Z).

If $Z \neq 0$, the projective point $(X : Y : Z)$ corresponds to the affine point $\left(\frac{X}{Z}, \frac{Y}{Z}\right) \in \mathbf{K}^2$. If $Z = 0$, the projective point $(X : Y : Z)$ has no affine representation. Projective points with no affine representation are called *points at infinity.* Using informal notation,

$$\mathbb{P}^2(\mathbf{K}) = \mathbb{A}^2(\mathbf{K}) \cup \{\text{Points at infinity}\}.$$

If one representative of the equivalence class $P = (X : Y : Z)$ satisfies equation (1.2), all representatives of the class satisfy equation (1.2) (as F is homogeneous).

Therefore, it makes sense to ask whether $F(X,Y,Z) = 0$ for some point $(X : Y : Z) \in \mathbb{P}^2(\mathbf{K})$.

Let $\overline{\mathbf{K}}$ be the algebraic closure[1] of $\mathbf{K}$, i.e. $\overline{\mathbf{K}}$ is an algebraic extension of $\mathbf{K}$ such that every $p \in \overline{\mathbf{K}}[X]$ with $\deg(p) \geq 1$ has a root in $\overline{\mathbf{K}}$. We now define:

Definition 1.1 (Projective curve)**.** Let $F \in \mathbf{K}[X,Y,Z]$. Assume that $F \neq 0$ and that F is homogeneous. A *projective curve* C over $\mathbf{K}$ is the set of solutions in $\mathbb{P}^2(\overline{\mathbf{K}})$ of the equation

$$C \,:\, F(X,Y,Z) = 0.$$

The *degree* of C is the degree of F. Let $\mathbf{L}$ be a field with $\mathbf{K} \subseteq \mathbf{L} \subseteq \overline{\mathbf{K}}$. A point $(X : Y : Z)$ on C is said to be $\mathbf{L}$-*rational* if there exists $\lambda \in \overline{\mathbf{K}}^*$ and $(X', Y', Z') \in \mathbf{L}^3 \setminus \{(0,0,0)\}$ such that $X = \lambda X'$, $Y = \lambda Y'$ and $Z = \lambda Z'$. The set of $\mathbf{L}$-rational points is denoted $C(\mathbf{L})$.

If the field $\mathbf{L}$ is apparent from the context, then $C(\mathbf{L})$ is simply called the *rational* points. We say that a projective curve is *non-singular* if the (formal) partial derivatives of F do not vanish simultaneously at any point of C.

$\circ$

With these notions in place, we are ready to define the concept of an *elliptic curve.*

Definition 1.2 (Elliptic curve)**.** Let E be a projective curve over $\mathbf{K}$ given by

$$E \,:\, F(X,Y,Z) = 0,$$

where F has the form

$$F(X,Y,Z) = Y^2Z - X^3 + a_1XYZ - a_2X^2Z + a_3YZ^2 - a_4XZ^2 - a_6Z^3.$$

If F is non-singular, the projective curve E is called an *elliptic curve.* The equation for E is written as

$$E \,:\, Y^2Z + a_1XYZ + a_3YZ^2 = X^3 + a_2X^2Z + a_4XZ^2 + a_6Z^3. \tag{1.3}$$

Equation (1.3) is called the *Weierstraß form* of E.

$\circ$

Let E be an elliptic curve given by equation (1.3), and let $P = (X : Y : Z)$ be a point on E. If $Z \neq 0$, we can put $x' := \frac{X}{Z}$ and $y' := \frac{Y}{Z}$ (notice that x' and y'

[1] Strictly speaking, the algebraic closure of $\mathbf{K}$ can (using Zorn's lemma) only be shown to be unique up to an isomorphism which fixes the elements of $\mathbf{K}$. We will disregard this and simply speak of *the* algebraic closure of $\mathbf{K}$.

are independent of the choice of representative of P). Then, the point (x', y') is a solution of the equation

$$y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6. \tag{1.4}$$

This corresponds to the equation

$$f(x, y) = 0$$

with

$$f(x, y) = y^2 + a_1xy + a_3y - x^3 - a_2x^2 - a_4x - a_6.$$

Equation (1.4) is called the *affine* Weierstraß form of E. Conversely, if (x', y') is a solution of equation (1.4), the projective point $(x' : y' : 1)$ is a solution of equation (1.3). This gives a $1 - 1$ correspondence between solutions of equation (1.3) with $Z \neq 0$ and solutions of equation (1.4).

If $Z = 0$, equation (1.3) says that $X^3 = 0$. The polynomial X^3 has the triple root $X = 0$, and equation (1.3) with $X = Z = 0$ holds for any value of Y. According to the definition of $\mathbb{P}^2$, we have $Y \neq 0$. Therefore, $P = (0 : 1 : 0)$ is a point on a curve in Weierstraß form, and it is the only projective point on the curve with $Z = 0$. It is a point at infinity, so it has no representation in affine coordinates. We count it as a rational point and represent it by the symbol $\mathcal{O}$, when affine coordinates are being used. In the affine case the $\mathbf{K}-$rational points are:

$$E(\mathbf{K}) = \{(x, y) \in \mathbf{K}^2 \mid f(x, y) = 0\} \cup \{\mathcal{O}\}. \tag{1.5}$$

In summary, the correspondence between the projective and the affine representation of points in $E(\mathbf{K})$ is given by

$$\begin{cases} (X : Y : Z) & \leftrightarrow \left(\frac{X}{Z}, \frac{Y}{Z}\right), \quad Z \neq 0 \\ (0 : 1 : 0) & \leftrightarrow \mathcal{O} \end{cases}.$$

This correspondence between affine and projective points on E allows us to switch between representations, and we will use both the projective and the affine description interchangeably. We use the notation (x, y) for affine points and the notation $(X : Y : Z)$ for projective points.

In order to get a shorter form of the equation for E, we use the following proposition:

Proposition 1.1. *Assume that char($\boldsymbol{K}$) $\neq 2, 3$. If E is an elliptic curve over $\boldsymbol{K}$ given by equation (1.4), there is a linear change of variables such that E can be written on the form*

$$E \,:\, y^2 = x^3 + ax + b. \tag{1.6}$$

Proof: The change of variables is given by

$$x' := x - \frac{a_2 + \frac{a_1^2}{4}}{3},$$
$$y' := y + \frac{a_1 x' + a_3}{2}.$$

■

A curve given by equation (1.6) is said to be in *short Weierstraß form.* As we will be working with fields which satisfy the condition in Proposition 1.1, we will use the short Weierstraß form in the sequel. The homogeneous version of equation (1.6) is

$$E \,:\, Y^2 Z = X^3 + aXZ + bZ^3.$$

So far, we have implicitly made the assumption that the variables in $\mathbf{K}[X, Y, Z]$ all have the same degree $\delta(X) = \delta(Y) = \delta(Z) = 1$. This is the standard choice, but there is nothing to stop us from assigning new degrees, or *weights*, to X, Y and Z. Our choice is to define that

$$\delta(X) := 2, \quad \delta(Y) := 3, \quad \delta(Z) := 1.$$

With this definition, the homogenization G of f, where f is given by equation (1.1), is

$$G(X, Y, Z) = \sum_{i,j} a_{i,j} X^i Y^j Z^{2 \cdot deg(f) - 2i - 3j}. \tag{1.7}$$

If a point $(\xi, \eta, \zeta) \in \mathbf{K}^3$ satisfies $G(\xi, \eta, \zeta) = 0$, then so will $(\lambda^2 \xi, \lambda^3 \eta, \lambda \zeta)$ for any $\lambda \in \mathbf{K}^*$. This motivates the definition of yet another equivalence relation on $\dot{\mathbf{K}}$. We define that

$$(\xi, \eta, \zeta) \sim (\xi', \eta', \zeta') \text{ if}$$
$$\exists \lambda \in \mathbf{K}^* \,:\, \xi = \lambda^2 \xi' \wedge \eta = \lambda^3 \eta' \wedge \zeta = \lambda \zeta'.$$

The quotient space $\dot{\mathbf{K}}/\sim$ is called the *weighted projective plane* over $\mathbf{K}$ with weights 2, 3 and 1. It is denoted $\mathbb{P}^2_{(2,3,1)}(\mathbf{K})$. Points in $\mathbb{P}^2_{(2,3,1)}(\mathbf{K})$ are written as $(\xi : \eta : \zeta)$ and are said to be in *Jacobian* coordinates.

If $\zeta \neq 0$, the Jacobian point $(\xi : \eta : \zeta)$ equals $\left(\frac{\xi}{\zeta^2} : \frac{\eta}{\zeta^3} : 1\right)$, corresponding to the affine point $\left(\frac{\xi}{\zeta^2}, \frac{\eta}{\zeta^3}\right)$. Points with $\zeta = 0$ are the points at infinity with no representation in affine coordinates.

When using Jacobian coordinates, an elliptic curve in short Weierstraß form is given by:

$$E \,:\, Y^2 = X^3 + aXZ^4 + bZ^6. \tag{1.8}$$

This is seen by homogenizing equation (1.6) as shown in equation (1.7).

Assume that $(\xi : \eta : \zeta)$ is a point at infinity, i.e. $\zeta = 0$, satisfying equation (1.8). Then, $\eta^2 = \xi^3$. As we are working in $\mathbb{P}^2_{(2,3,1)}$, we see that $(\xi, \eta, 0) \sim (1, 1, 0)$, as it follows by taking $\lambda := \frac{\eta}{\xi}$ in the definition on page 7 (the definition of $\mathbb{P}^2_{(2,3,1)}$ ensures that $\xi \neq 0$). Indeed, this gives $(\lambda^2\xi, \lambda^3\eta, 0) = (\xi^2, \xi^3, 0)$, which is equivalent to $(1, 1, 0)$. Hence, the only point at infinity in Jacobian coordinates on E is $(1 : 1 : 0)$, so, as in the projective case, exactly one of the points at infinity is on the curve. We will represent this point by $\mathcal{O}$, when using affine coordinates. In summary, the correspondence between the Jacobian and the affine representation of points in $E(\mathbf{K})$ is given by

$$\begin{cases} (\xi : \eta : \zeta) & \leftrightarrow \left(\frac{\xi}{\zeta^2}, \frac{\eta}{\zeta^3}\right), \quad \zeta \neq 0 \\ (1 : 1 : 0) & \leftrightarrow \mathcal{O} \end{cases} .$$

1.2 The Group Law

Let E be an elliptic curve over the field $\mathbf{K}$ defined by

$$E : Y^2 Z = X^3 + aXZ^2 + bZ^3.$$

Let $\mathbf{L}$ be a field with $\mathbf{K} \subseteq \mathbf{L} \subseteq \overline{\mathbf{K}}$. The set $E(\mathbf{L})$ of $\mathbf{L}$-rational points on E has an interesting property. With a proper definition of a composition $\oplus$, called *addition* on $E(\mathbf{L})$, the pair $(E(\mathbf{L}), \oplus)$ is an abelian group. We will only present an overview of the construction of the composition and refer to [ACD+05] or [Sil92] for details.

When defining a composition on $E(\mathbf{L})$, it turns out that one has to distinguish between adding two distinct points and doubling a point. Let P, $Q \in E(\mathbf{L})$ with $P \neq Q$. We will need the following:

(i) The straight line joining P and Q intersects the curve at exactly one further point R. The point R is $\mathbf{L}$-rational. The cases $R = P$ or $R = Q$ are not excluded.

(ii) Let P be an $\mathbf{L}$-rational point on E. The tangent to E at P intersects E at exactly one further point R, which is $\mathbf{L}$-rational. The case $R = P$ is not excluded.

The statements above can be summarized in the following way: In the projective plane, any line which intersects the elliptic curve E intersects E at exactly three points, when counting *multiplicities* (with a suitable definition of what multiplicity should mean). We will not go into details with this. Instead, we will consider statements (i) and (ii) above as facts. Recall, from Section 1.1, that in $\mathbb{P}^2(\mathbf{L})$, the point $(0 : 1 : 0)$ is the only point at infinity on E. Denote the third point of intersection between E and the line through P and Q by $P * Q$. Similarly, $P * P$ denotes the other intersection point between E and the tangent to E at P. The group law on $E(\mathbf{L})$ is defined as follows:

Neutral element: As the neutral element we select $(0 : 1 : 0)$.

Inverse element: We define the inverse $-P$ of P as

$$-P := (0 : 1 : 0) * P.$$

Addition: We know that $P * Q \in E(\mathbf{L})$, and we define

$$P \oplus Q := -(P * Q).$$

Doubling: We know that $P * P \in E(\mathbf{L})$, and we define

$$P \oplus P := -(P * P).$$

The definition says that one gets $P \oplus Q$ by "drawing" the line determined by the two points P and Q, finding the third point of intersection $P * Q$ and taking the inverse of $P * Q$. A doubling is done similarly, only with the line being a tangent to E at P. The situation for $\mathbf{L} = \mathbb{R}$ is shown in Figure 1.1.

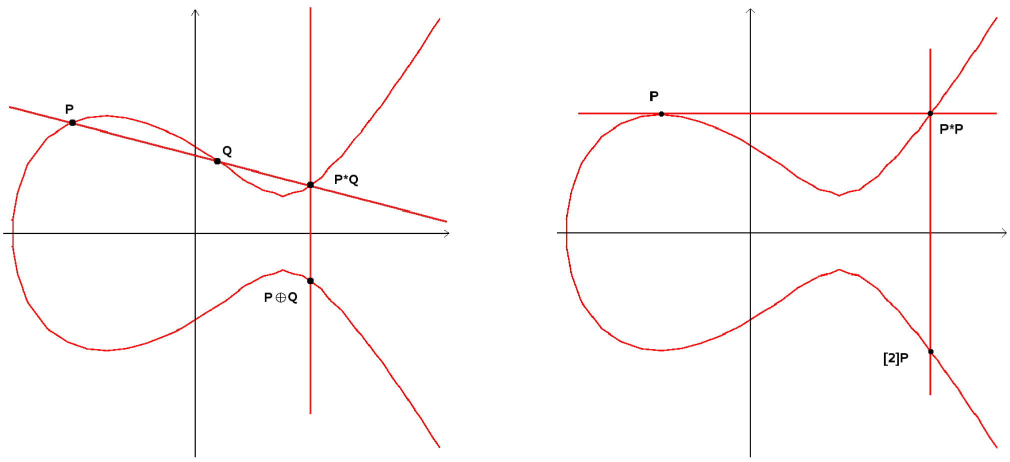


Figure 1.1: The figure shows addition (left) and doubling (right) on the elliptic curve $E : y^2 = x^3 - 10x + 15$ over $\mathbb{R}$.

Using Max Noethers's theorem or direct calculation, one can prove that, with these definitions, $(E(\mathbf{L}), \oplus)$ is an additive, abelian group. Most of the work involved in proving this lies in showing that $\oplus$ is associative. A proof can be found in [Sil92]. An alternative proof, using divisor theory, can be found in [ACD$^+$05].

Definition 1.3 (Scalar multiplication)**.** Let k be an integer, and let $P \in E(\mathbf{L})$. If k is non-negative, we define $[k]P$ as

$$[k]P := \begin{cases} \mathcal{O}, & k = 0 \\ \overbrace{P \oplus P \oplus \cdots \oplus P}^{k}, & k > 0 \end{cases}.$$

If k is negative, we define

$$[k]P := [-k](-P).$$

We say that $[k]P$ is the result of *scalar multiplication* of the point P by the scalar k.

1.2.1 Formulas for Addition and Doubling

The geometric definition of the composition $\oplus$ is not very useful in applied situations. If one has to implement the elliptic curve addition in hardware or software, it is more convenient to work with explicit formulas. We have introduced three different coordinate representations of an elliptic curve E. This section specifies formulas for addition and doubling in each representation. Deducing the formulas does not require any advanced mathematics, but a lot of special cases have to be considered. Therefore, the deduction is excluded from this examination.

Projective coordinates: The equation for E is

$$E \,:\, Y^2Z = X^3 + aXZ^2 + bZ^3.$$

The group is $(E(\mathbf{L}), \oplus)$ with neutral element $(0 : 1 : 0)$. Let $P, Q \in E(\mathbf{L})$ with $P = (X_1 : Y_1 : Z_1)$ and $Q = (X_2 : Y_2 : Z_2)$. Assume that $P \neq Q$. The inverse of P is $-P = (X_1 : -Y_1 : Z_1)$. Formulas for $P \oplus Q = (X_3 : Y_3 : Z_3)$ and $[2]P = (X_4 : Y_4 : Z_4)$ are:

Addition:

Set $A = Y_2Z_1 - Y_1Z_2$, $B = X_2Z_1 - X_1Z_2$ and $C = A^2Z_1Z_2 - B^3 - 2B^2X_1Z_2$.

Then, $X_3 = BC$, $Y_3 = A(B^2X_1Z_2 - C) - B^3Y_1Z_2$ and $Z_3 = B^3Z_1Z_2$.

Doubling:

Set $A = 3X_1^2 + aZ_1^2$, $B = Y_1Z_1$, $C = X_1Y_1B$ and $D = A^2 - 8C$.

Then, $X_4 = 2BD$, $Y_4 = A(4C - D) - 8Y_1^2B^2$ and $Z_4 = 8B^3$.

Affine coordinates: The equation for E is

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

The group is $(E(\mathbf{L}), \oplus)$ with $E(\mathbf{L})$ as in equation (1.5) and neutral element $\mathcal{O}$. Let $P \in E(\mathbf{L}) \setminus \{\mathcal{O}\}$. As $\mathcal{O}$ does not have an affine representation, we must consider the operations $-\mathcal{O}$, $P \oplus \mathcal{O}$, $P - P$ and $[2]\mathcal{O}$ separately. We have:

$$\begin{aligned} -\mathcal{O} &= \mathcal{O} \\ P \oplus \mathcal{O} &= P \\ P - P &= \mathcal{O} \\ [2]\mathcal{O} &= \mathcal{O}. \end{aligned}$$

When implementing the group law in affine coordinates, one must choose a suitable representation of $\mathcal{O}$ and take account of the cases mentioned above.

Let $P = (x_1, y_1)$ and $Q = (x_2, y_2)$ be affine points on E with $P \neq \pm Q$. The inverse of P is $-P = (x_1, -y_1)$. Formulas for $P \oplus Q = (x_3, y_3)$ and $[2]P = (x_4, y_4)$ are:

Addition:

Set $\lambda = \frac{y_1 - y_2}{x_1 - x_2}$. Then, $x_3 = \lambda^2 - x_1 - x_2$ and $y_3 = \lambda(x_1 - x_3) - y_1$.

Doubling:

Set $\lambda = \frac{3x_1^2 + a}{2y_1}$. Then, $x_4 = \lambda^2 - 2x_1$ and $y_4 = \lambda(x_1 - x_4) - y_1$.

Jacobian coordinates: The equation for E is

$$E : Y^2 = X^3 + aXZ^4 + bZ^6.$$

The group is $(E(\mathbf{L}), \oplus)$ with neutral element $(1 : 1 : 0)$. Let $P, Q \in E(\mathbf{L})$ with $P = (\xi_1 : \eta_1 : \zeta_1)$ and $Q = (\xi_2 : \eta_2 : \zeta_2)$. Assume that $P \neq Q$. The inverse of P is

$-P = (\xi_1 : -\eta_1 : \zeta_1)$. Formulas for $P \oplus Q = (\xi_3 : \eta_3 : \zeta_3)$ and $[2]P = (\xi_4 : \eta_4 : \zeta_4)$ are:

Addition:

Set $A = \xi_1\zeta_2^2$, $B = \xi_2\zeta_1^2$, $C = \eta_1\zeta_2^3$, $D = \eta_2\zeta_1^3$, $E = B - A$ and $F = D - C$.

Then, $\xi_3 = -E^3 - 2AE^2 + F^2$, $\eta_3 = -CE^3 + F(AE^2 - \xi_3)$ and $\zeta_3 = \zeta_1\zeta_2 E$.

Doubling:

Set $A = 4\xi_1\eta_1^2$ and $B = 3\xi_1^2 + a\zeta_1^4$.

Then, $\xi_4 = -2A + B^2$, $\eta_4 = -8\eta_1^4 + B(A - \xi_4)$ and $\zeta_4 = 2\eta_1\zeta_1$.

One can use these formulas to implement addition on elliptic curves given in short Weierstraß form, as long as an implementation of the operations in the ground field is available.

Chapter 2

Elliptic Curves in Cryptography

This chapter contains a brief description of how elliptic curves are used in cryptography. As described in [BSS99], not all elliptic curves are equally secure for cryptographic purposes. We present a selection of secure curves used in real-life cryptographic applications.

2.1 Cryptographic Protocols

This section presents the elliptic curve analogs of the ElGamal cryptosystem and the digital signature algorithm (DSA). Descriptions of these can be found in [Kob94]. In the setting of an ECC-system, the latter is standardized as the Elliptic Curve Digital Signature Algorithm (ECDSA) and is specified in [X9.98]. As is common, when describing cryptographic protocols, we assume that Alice wants to send a message P to Bob, while the eavesdropper Eve is able to intercept any information exchanged by Alice and Bob. Let $p > 3$ be a prime number and let E be an elliptic curve over $\mathbb{F}_p$. We assume that P is represented as an element of $E(\mathbb{F}_p)$.

2.1.1 Elliptic Curve ElGamal Cryptosystem

Initially, Alice and Bob fix a publicly known *base element* $Q \in E(\mathbb{F}_p)$ of prime order n.

(i) Bob chooses a random positive integer $k_B \in [1, n-1]$. He publishes the *public key* $[k_B]Q$ and keeps secret the *private key* k_B.

(ii) Alice chooses a secret, random positive integer $k \in [1, n-1]$ and sends $([k]Q, P \oplus [k]([k_B]Q)$ to Bob.

(iii) Bob recovers P as $P \oplus [k]([k_B]Q) \oplus (-[k_B]([k]Q)) = P$.

Eve may intercept $([k]Q, P \oplus [k]([k_B]Q)$, but she needs to solve the ECDLP in order to find k_B or k.

2.1.2 ECDSA

Let $n = |Q|$ be the (prime) order of a publicly known base point $Q \in E(\mathbb{F}_p)$. The ECDSA uses a *cryptographic hash function*[1] $h : E(\mathbb{F}_p) \to \mathbb{Z}/n\mathbb{Z}$. Let k_A and $[k_A]Q$ be Alice's private and public key respectively. The keys are chosen by Alice in a way similar to the one described in the ElGamal cryptosystem. Alice generates a signature for the message P in the following way:

Signature generation

(i) Alice computes $e = h(P)$.

(ii) She selects a random $k \in [1, n-1]$ and computes $(x_1, y_1) = [k]Q$. If $x_1 \equiv 0 \mod n$, she repeats this step.

(iii) She sets $r := x_1 \mod n$.

(iv) She sets $s := k^{-1}(e + k_A r) \mod n$. If $s = 0$, she goes to step (i).

(v) Along with the message, she sends the signature (r, s) to Bob.

Bob wants to verify that Alice sent the message P signed with (r, s). To do this, he performs the following steps:

Signature verification

(i) If r or s is not in $[1, n-1]$, the signature is rejected.

(ii) Bob computes $e = h(P)$.

(iii) He sets $c := s^{-1} \mod n$, $u_1 := ec \mod n$ and $u_2 := rc \mod n$.

(iv) He computes $(x_1, y_1) = [u_1]Q \oplus [u_2]([k_A]Q)$. If the resulting point is not affine, the signature is rejected.

(v) He sets $\nu := x_1 \mod n$. If $r = \nu$, the signature is verified. If $r \neq \nu$, the signature is rejected.

As one can see, both encryption/decryption and signature generation/verification requires scalar multiplication, and it turns out that scalar multiplication on the elliptic curve is actually the most time consuming operation involved in the protocols. In Chapters 3 and 4 we examine different ways of making scalar multiplication as efficient as possible. The scalar multiplication performed in step (iv) of the signature verification is a special case for which one can use a technique known as "Straus' algorithm" or "Shamir's trick". The reader is referred to [ACD+05] for details on this subject.

[1] Standards for hash functions can be found in [X9.98].

2.2 Elliptic Curves Recommended by NIST

In January 2000, FIPS PUB[2] 186-2 was published. This is a digital signature standard, which includes the ECDSA and is the result of a revision of FIPS PUB 186-1 performed by NIST. For elliptic curves, FIPS PUB 186-2 recommends five prime fields and five binary fields. In this examination we only consider prime fields.

The prime fields are $\mathbb{F}_{p_{192}}$, $\mathbb{F}_{p_{224}}$, $\mathbb{F}_{p_{256}}$, $\mathbb{F}_{p_{384}}$ and $\mathbb{F}_{p_{521}}$, where

$$p_{192} = 2^{192} - 2^{64} - 1,$$

$$p_{224} = 2^{224} - 2^{96} + 1,$$

$$p_{256} = 2^{256} - 2^{224} + 2^{192} + 2^{96} - 1,$$

$$p_{384} = 2^{384} - 2^{128} - 2^{96} + 2^{32} - 1,$$

$$p_{521} = 2^{521} - 1.$$

The form of the primes allows for very efficient modular reduction (see [Sol99]). For each of the five fields an elliptic curve was selected. As we saw in Chapter 1, an elliptic curve over $\mathbb{F}_p$ can be defined by an equation of the form $y^2 = x^3+ax+b$, where $a, b \in \mathbb{F}_p$. The NIST curves all have $a \equiv -3 \mod p$ which, as we shall see in Chapter 4, is an advantage when performing certain elliptic curve operations. The value of b was chosen pseudo-randomly, via the SHA-1 based method described in [X9.98] and [P1300], such that the group $(E(\mathbb{F}_p), \oplus)$ of rational points is of prime order for all five curves. The base point $P \in E(\mathbb{F}_p)$ was chosen to be a generator of the group. The NIST curves over $\mathbb{F}_{p_{192}}$, $\mathbb{F}_{p_{224}}$, $\mathbb{F}_{p_{256}}$, $\mathbb{F}_{p_{384}}$ and $\mathbb{F}_{p_{521}}$ with these properties are denoted P-192, P-224, P-256, P-384 and P-521 respectively. The value of b and the group order n corresponding to each of the five curves are shown in Table 2.1.

We will consider only the five NIST curves over prime fields. Curves over binary fields are described in detail in [ACD$^+$05].

[2]Federal Information Processing Standards Publication

P-192:
$p = 2^{192} - 2^{64} - 1$
$a =$ -3
$b =$ 0x 64210519 E59C80E7 0FA7E9AB 72243049 FEB8DEEC C146B9B1
$n =$ 0x FFFFFFFF FFFFFFFF FFFFFFFF 99DEF836 146BC9B1 B4D22831
P-224:
$p = 2^{224} - 2^{96} + 1$
$a =$ -3
$b =$ 0x B4050A85 0C04B3AB F5413256 5044B0B7 D7BFD8BA 270B3943 2355FFB4
$n =$ 0x FFFFFFFF FFFFFFFF FFFFFFFF FFFF16A2 E0B8F03E 13DD2945 5C5C2A3D
P-256:
$p = 2^{256} - 2^{224} + 2^{192} + 2^{96} - 1$
$a =$ -3
$b =$ 0x 5AC635D8 AA3A93E7 B3EBBD55 769886BC 651D06B0 CC53B0F6 3BCE3C3E 27D2604B
$n =$ 0x FFFFFFFF 00000000 FFFFFFFF FFFFFFFF BCE6FAAD A7179E84 F3B9CAC2 FC632551
P-384:
$p = 2^{384} - 2^{128} - 2^{96} + 2^{32} - 1$
$a =$ -3
$b =$ 0x B3312FA7 E23EE7E4 988E056B E3F82D19 181D9C6E FE814112 0314088F 5013875A C656398D 8A2ED19D 2A85C8ED D3EC2AEF
$n =$ 0x FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF C7634D81 F4372DDF 581A0DB2 48B0A77A ECEC196A CCC52973
P-521:
$p = 2^{521} - 1$
$a =$ -3
$b =$ 0x 00000051 953EB961 8E1C9A1F 929A21A0 B68540EE A2DA725B 99B315F3 B8B48991 8EF109E1 56193951 EC7E937B 1652C0BD 3BB1BF07 3573DF88 3D2C34F1 EF451FD4 6B503F00
$n =$ 0x 000001FF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFA 51868783 BF2F966B 7FCC0148 F709A5D0 3BB5C9B8 899C47AE BB6FB71E 91386409

Table 2.1: The table shows the five NIST curves over prime fields.

Part II

Efficient Scalar Multiplication

Chapter 3

Scalar Multiplication Methods

As mentioned in Section 2.1, the most time-consuming operation performed in an ECC-system is scalar multiplication, i.e. determining $[k]P$ for a positive integer k and $P \in E(\mathbb{F}_p)$. Scalar multiplication in $E(\mathbb{F}_p)$ consists of a sequence of elliptic curve doublings (`ECDBL`) and elliptic curve additions (`ECADD`) which, in turn, consist of a number of operations in the ground field $\mathbb{F}_p$. In this chapter we evaluate and compare a number of known scalar multiplication algorithms based on the number of `ECDBL` and `ECADD` required by the algorithm.

We point out that our evaluation and comparison is more detailed than previously published surveys of scalar multiplication methods, and we include many proofs of correctness of the presented algorithms[1] Hopefully, the degree of detail presented here will be helpful to anyone implementing a scalar multiplication method.

In this chapter, t denotes a function measuring the requirements of a given algorithm. For instance, if $\mathfrak{A}$ is an algorithm for performing scalar multiplication, one has $t(\mathfrak{A}) = u \cdot \texttt{ECDBL} + v \cdot \texttt{ECADD}$ for some non-negative numbers u and v. The goal of this chapter is to choose an algorithm $\mathfrak{A}$, for which $t(\mathfrak{A})$ is minimal under some conditions. The setup is as follows:

Setup: Let k be a positive integer with binary representation

$$k = (k_{l-1} \cdots k_0)_2,$$

where $k_{l-1} = 1$. Let E be an elliptic curve over $\mathbb{F}_p$ given by the equation $y^2 = x^3 - 3x + b$. Let $P = (x, y)$ be an affine point in $E(\mathbb{F}_p)$. We wish to determine the point

$$[k]P \in E(\mathbb{F}_p).$$

[1] An introduction to proving correctness of algorithms can be found in [CLRS01].

We assume that k is positive. If k is negative, the scalar multiplication algorithms in this section will produce the expected output for input $k' = -k$ and $P' = -P = (x, -y)$. The naive way of determining $[k]P$ is to compute $[2]P, [3]P, \ldots, [k-1]P, [k]P$, which requires $\texttt{ECDBL} + (k-2) \cdot \texttt{ECADD}$. This is not feasible when k is large, so we will aim at reducing the requirement.

3.1 Binary Methods

This section presents three algorithms for performing scalar multiplication on an elliptic curve. The algorithms all use a binary representation of the scalar – hence the name *binary method*.

3.1.1 The Double-and-add Method

The *double-and-add* method is one of the oldest methods for performing scalar multiplication[2]. It is based on the observation that $[2^n]P$ can be computed as

$$[2]P, [4]P, \ldots, [2^n]P$$

in n operations. The method is shown in Algorithm 1.

Algorithm 1 Double and add

Input: An affine point $P \in E(\mathbb{F}_p)$ and $k = (k_{l-1} \cdots k_0)_2$.
Output: $[k]P \in E(\mathbb{F}_p)$.

1: $Q \leftarrow P$; $i \leftarrow l - 2$;
2: **while** $i \geq 0$ **do**
3: $\quad Q \leftarrow [2]Q$;
4: $\quad$ **if** $k_i = 1$ **then**
5: $\quad\quad Q \leftarrow P \oplus Q$;
6: $\quad$ **end if**
7: $\quad i \leftarrow i - 1$;
8: **end while**
9: **return** Q

Proof of correctness: Notice that i is decremented in line 7, so eventually the algorithm terminates due to the condition in line 2. Algorithm 1 maintains the loop invariant

$\mathcal{L}$: *At the start of each iteration of the while-loop in lines 2-8,*

$$Q = \left[\sum_{j=i+1}^{l-1} k_j 2^{j-i-1}\right] P.$$

[2] In a general (multiplicatively written) group the algorithm is known as the *square-and-multiply* method and performs exponentiation.

As $k_{l-1} = 1$, the statement is true prior to the first iteration. Furthermore, we have for all $i < l$ that

$$[(k_{l-1} \cdots k_{i+1} k_i)_2]P = [2]([(k_{l-1} \cdots k_{i+1})_2]P) + [k_i]P, \tag{3.1}$$

since $(k_{l-1} \cdots k_{i+1} k_i)_2 = 2(k_{l-1} \cdots k_{i+1})_2 + k_i$. Therefore, when $i = -1$, the algorithm terminates and returns $Q = \sum_{j=0}^{l-1} k_j 2^j = [k]P$.

■

The number of additions required by Algorithm 1 depends on the *Hamming weight* (the number of non-zero bits) $\nu(k)$ of k, as an addition is performed if, and only if, $k_i = 1$. We have $\nu(k) = \frac{1}{2}l$ on average, so on average the algorithm executes $\frac{1}{2}(l-1) \cdot \texttt{ECADD}$. One $\texttt{ECDBL}$ per bit is always performed, so we get the following result:

Proposition 3.1 (Requirement of the double-and-add method)**.** *On average,*

$$t(\textit{Algorithm 1}) = (l-1) \cdot \texttt{ECDBL} + \frac{l-1}{2} \cdot \texttt{ECADD}.$$

Example 3.1. The smallest field recommended by NIST is $\mathbb{F}_{p_{192}}$ (see Section 2.2). If we assume that k is a 192-bit integer, the average cost of Algorithm 1 is

$$191 \cdot \texttt{ECDBL} + 96 \cdot \texttt{ECADD}.$$

3.1.2 The 2^w-ary Method

An obvious modification of Algorithm 1 is to use a larger base for representing k. The base could be any number m, but we will focus on the special case $m = 2^w$ for a positive integer $w \geq 1$. This is equivalent to partitioning the binary representation of k into windows of length w and process these windows one by one. For instance, if $k = (398)_{10} = (110001110)_2$ and $w = 3$, we get the partitioning

$$k = (\underline{110}\ \underline{001}\ \underline{110})_2.$$

This corresponds to the equality $k = (616)_{2^3}$.

If one can afford to use storage for precomputed values, Algorithm 2, originally proposed by Brauer in his paper *On addition chains* from 1939, is an improvement of Algorithm 1. The algorithm uses the function $\sigma : \mathbb{N}_0 \to \mathbb{N} \times \mathbb{N}_0$ defined by

$$\sigma(m) = \begin{cases} (w, 0), & m = 0 \\ (s, u), & m \neq 0, \text{ where } m = 2^s u \text{ with } u \text{ odd.} \end{cases}$$

Algorithm 2 2^w-ary scalar multiplication

Input: An affine point $P \in E(\mathbb{F}_p)$, $w \geq 1$ and $k = (e_{n-1} \cdots e_0)_{2^w}$.
Output: $[k]P \in E(\mathbb{F}_p)$.

1: Compute the odd multiples $[3]P, [5]P, \ldots, [2^w - 1]P$.
2: $Q \leftarrow \mathcal{O}$;
3: $i \leftarrow n - 1$;
4: $(s, u) \leftarrow \sigma(e_i)$;
5: **while** $i \geq 0$ **do**
6:     **for** $j = 1$ to $w - s$ **do**
7:         $Q \leftarrow [2]Q$;
8:     **end for**
9:     **if** $e_i \neq 0$ **then**
10:         $Q \leftarrow Q \oplus [u]P$; //As u is odd, $[u]P$ has been precomputed in line 1.
11:     **end if**
12:     **for** $j = 1$ to s **do**
13:         $Q \leftarrow [2]Q$;
14:     **end for**
15:     $i \leftarrow i - 1$;
16: **end while**
17: **return** Q

Proof of correctness: The proof is almost completely identical to the proof of correctness of Algorithm 1. The loop invariant is in this case

$$\mathcal{L} : \textit{At the start of the while-loop in lines 3-12,}$$
$$Q = \left[\sum_{j=i+1}^{n-1} e_j 2^{w(j-i-1)}\right] P.$$

■

We assume that the `ECDBL` in line 7 is not carried out when $Q = \mathcal{O}$. This is reasonable, as $[2]\mathcal{O} = \mathcal{O}$. Similarly, we assume that the very first addition in line 6 is not performed, as $Q \oplus [u]P = [u]P$. Algorithm 2 executes $(l-1) \cdot$ `ECDBL` in lines 2-16 due to the splitting of doubles into a part before and a part after the `ECADD` in line 10. An `ECADD` is performed for each $e_i \neq 0$. On average, $\frac{2^w-1}{2^w}$ of the e_i's are non-zero, so the main loop performs

$$(n-1)\frac{2^w-1}{2^w} \cdot \texttt{ECADD} = \left(\left\lceil \frac{l}{w} \right\rceil - 1\right) \cdot \frac{(2^w-1)}{2^w} \cdot \texttt{ECADD}$$

on average. The precomputations require one `ECDBL` and $(2^{w-1}-1)$`ECADD`, so the average requirement of Algorithm 2 is:

Proposition 3.2 (Requirement of the 2^w-ary method)**.** *One has*

$$t(\textit{Algorithm 2}) = l \cdot \texttt{ECDBL} + \left(\left\lceil \frac{l}{w} \right\rceil \cdot \frac{2^w - 1}{2^w} + 2^{w-1} - 2 \right) \cdot \texttt{ECADD}$$

on average. Algorithm 2 requires storage for $2^{w-1} - 1$ *precomputed points.*

One needs to choose an optimal value of w. To minimize the number of `ECADD` on the right hand side of the equation in Proposition 3.2, one has to minimize the value of

$$\phi(w) = \left\lceil \frac{l}{w} \right\rceil \cdot \frac{2^w - 1}{2^w} + 2^{w-1} - 1$$

for a fixed l. For instance, one gets the values of $\phi(w)$ shown in Figure 3.1 for $l = 192$ and $l = 521$, when $w \in [1, 10]$.

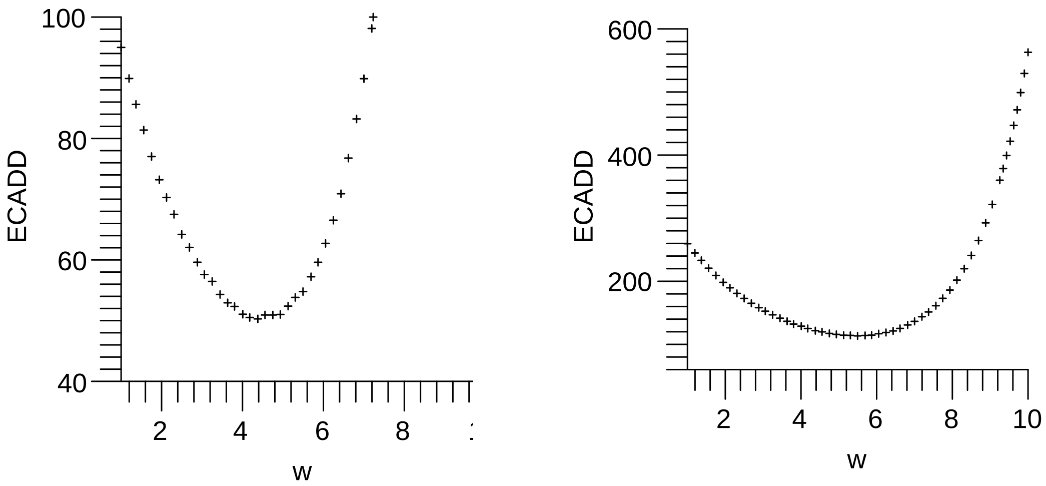


Figure 3.1: The plots show the value of $\phi(w)$ for $l = 192$ and $l = 521$ respectively, when $w \in [1, 10]$.

In the cases in Figure 3.1, $w = 4$ and $w = 6$ respectively are optimal. Similar considerations lead to the optimal values of w for various values of l shown in Table 3.1.

l	$[70, 196]$	$[197, 520]$	$[521, 1452]$
w	4	5	6

Table 3.1: The table shows a selection of optimal values of w for the 2^w-ary method.

Example 3.2. In the example where k is a 192-bit integer, we get the optimal value $w = 4$ from Table 3.1. This results in an average requirement of

$$192 \cdot \texttt{ECDBL} + 51 \cdot \texttt{ECADD}.$$

Compared to the double-and-add method, the 2^w-ary method saves 45 `ECADD` on average, while it uses an extra `ECDBL` and storage for precomputed values.

3.1.3 Sliding-window Method

If we return to the situation $k = (398)_{10} = (110001110)_2$ from Section 3.1.2, we see that Algorithm 2 computes $[k]P$ from the following intermediate values of Q:

$$\mathcal{O}, [3]P, [6]P, [12]P, [24]P, [48]P, [49]P, [98]P, [196]P, [199]P, [398]P.$$

Alternatively, one could compute

$$\mathcal{O}, [3]P, [6]P, [12]P, [24]P, [48]P, [96]P, [192]P, [199]P, [398]P,$$

whereby one `ECADD` is saved. The latter sequence of calculations corresponds to allowing the "windows" in the representation of k to be separated by one or more consecutive zeroes:

$$(398)_{10} = (\underline{11}\,000\,\underline{111}\,0)_2.$$

Skipping a zero can then be done by performing an `ECDBL`. Algorithm 3 shows the method in general.

Remark 3.1. In lines 4-7, Algorithm 3 performs `ECDBL` until a k_i with $k_i \neq 0$ is found. In lines 8-9, for fixed $k_i = 1$, the longest subsequence of bits $(k_i \cdots k_t)$ of length less than or equal to w such that $k_t = 1$ is found. As $k_t = 1$, we have that $(k_i \cdots k_t)_2$ is odd, so $[(k_i \cdots k_t)_2]P$ has been precomputed.

∘

Algorithm 3 Sliding-window scalar multiplication

Input: An affine point $P \in E(\mathbb{F}_p)$, $w \geq 1$ and $k = (k_{l-1} \cdots k_0)_2$.
Output: $[k]P \in E(\mathbb{F}_p)$.

1: Compute the odd multiples $[3]P, [5]P, \ldots, [2^w - 1]P$.
2: $Q \leftarrow P$; and $i \leftarrow l - 2$;.
3: **while** $i \geq 0$ **do**
4:    **if** $k_i = 0$ **then**
5:       $Q \leftarrow [2]Q$;
6:       $i \leftarrow i - 1$;
7:    **else**
8:       $s \leftarrow \max\{i - w + 1, 0\}$;
9:       $t \leftarrow \min\{j \in \mathbb{Z} \mid j \geq s \wedge k_j = 1\}$;
10:       **for** $h = 1$ to $i - t + 1$ **do**
11:          $Q \leftarrow [2]Q$;
12:       **end for**
13:       $u \leftarrow (k_i \cdots k_t)_2$;
14:       $Q \leftarrow Q \oplus [u]P$;
15:       **for** $h = 1$ to $t - s$ **do**
16:          $Q \leftarrow [2]Q$;
17:       **end for**
18:       $i \leftarrow s - 1$;
19:    **end if**
20: **end while**
21: **return** Q

Proof of correctness: Algorithm 3 assigns the value $\max\{i - w + 1, 0\}$ to s in line 8. After this assignment, $s \leq i$. When i becomes $s - 1$ in line 18, the value of i is decremented, so the algorithm eventually terminates. Algorithm 3 maintains the loop invariant

$\mathcal{L}$: *At the start of the while-loop in lines 3-20,*
$$Q = \left[\sum_{j=i+1}^{l-1} k_j 2^{j-i-1}\right] P.$$

The statement $\mathcal{L}$ is true prior to the first iteration, as $k_{l-1} = 1$. Let $i < l - 2$, and assume that $\mathcal{L}$ holds prior to the $(l - i - 2)$'th iteration. We aim at proving that $\mathcal{L}$ holds prior to the $(l - i - 1)$'th iteration. If $k_i = 0$, we have

$$Q = \left[\sum_{j=i+1}^{l-1} k_j 2^{j-i}\right] P \tag{3.2}$$

after the assignment in line 5. When the value of i is decremented in line 6, equation (3.2) says that $Q = \left[\sum_{j=i+1}^{l-1} k_j 2^{j-i-1}\right] P$ (keeping in mind that k_{i+1} – the former k_i – is zero).

If $k_i \neq 0$, we have

$$\begin{aligned}
Q &= \left[\left(\sum_{j=i+1}^{l-1} k_j 2^{j-s}\right) + 2^{t-s}u\right] P \\
&= \left[\left(\sum_{j=i+1}^{l-1} k_j 2^{j-s}\right) + 2^{t-s}(k_i 2^{i-j} + \cdots + k_t)\right] P \\
&= \left[\left(\sum_{j=i+1}^{l-1} k_j 2^{j-s}\right) + \sum_{j=t}^{i} k_j 2^{j-s}\right] P \\
&= \left[\left(\sum_{j=t}^{l-1} k_j 2^{j-s}\right)\right] P \\
&= \left[\left(\sum_{j=s}^{l-1} k_j 2^{j-s}\right)\right] P
\end{aligned}$$

after the execution of lines 13-17. Here, the last equation is valid as $k_j = 0$ for $s < j < t$. When i is assigned a new value in line 18, the loop invariant is reestablished, so $\mathcal{L}$ is maintained. At the end of the algorithm $i = -1$, and the loop invariant ensures that $Q = [k]P$.

■

Notice that Algorithm 3 performs one `ECDBL` for each bit in the binary representation of k and that an `ECADD` is performed only in the case where a window is created (in lines 7-19). Assume that k is unbounded. Let (X_n) be a random process (cf. Appendix A) given by

$$X_i = \begin{cases} 1, & k_i = 0 \\ w, & k_i \neq 0 \end{cases} .$$

We interpret the output of X_i as the length of the window created by Algorithm 3 in the i'th iteration of the main loop. For each X_i we have the distribution

$$P(X_i = 1) = P(X_i = w) = \frac{1}{2}.$$

For every i this gives an expectation of $EX_i = \frac{w+1}{2}$, so the expected number of bits of k being processed per iteration of the main loop is $\frac{w+1}{2}$. Divide the binary representation of k into pieces of length $\frac{w+1}{2}$, and recall that half of these pieces will imply an `ECADD` on average. We now see that Algorithm 3 requires

$$\frac{1}{2} \cdot \frac{2(l-1)}{w+1} \cdot \texttt{ECADD} = \frac{l-1}{w+1} \cdot \texttt{ECADD}$$

on average. Also counting the operations from the precomputations, one gets:

Proposition 3.3 (Requirement of the sliding-window method)**.** *One has*

$$t(\textit{Algorithm 3}) = l \cdot \texttt{ECDBL} + \left(\frac{l-1}{w+1} + 2^{w-1} - 1\right) \cdot \texttt{ECADD}$$

on average. Algorithm 3 requires storage for $2^{w-1} - 1$ *precomputed points.*

Figure 3.2 shows the number of ECADD required on average by Algorithm 3 for $l = 192$ and $l = 521$ as a function of w. Table 3.2 shows optimal values of w for selected values of l.

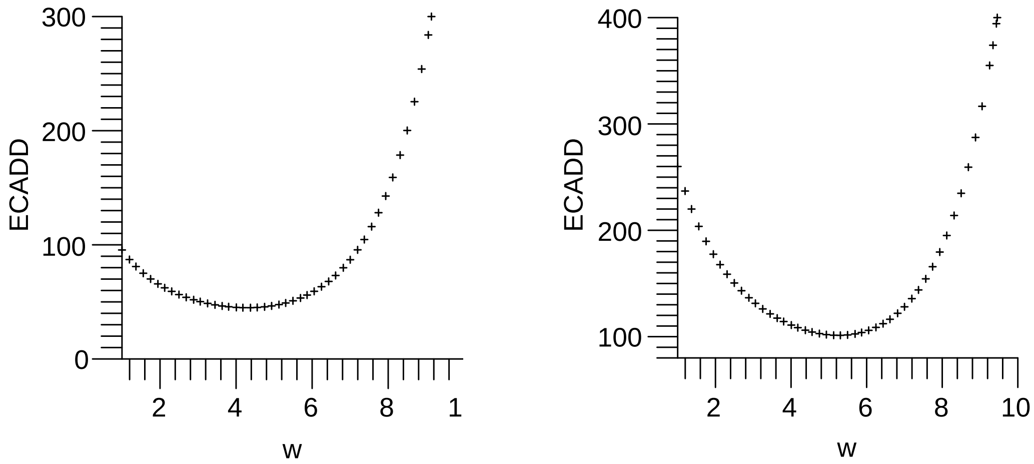


Figure 3.2: The plots show the total number of ECADD required by Algorithm 3 for $l = 192$ and $l = 521$ respectively, when $w \in [1, 10]$.

l	$[25, 80]$	$[81, 240]$	$[241, 672]$
w	3	4	5

Table 3.2: The table shows a selection of optimal values of w for the sliding-window method.

Example 3.3. In the example $l = 192$ and $w = 4$, Proposition 3.3 gives that Algorithm 3 requires

$$192 \cdot \texttt{ECDBL} + 45 \cdot \texttt{ECADD}$$

on average. Compared to the 2^w-ary method, the sliding-window method saves 6 ECADD on average and uses the same amount of storage for precomputed values.

3.2 Methods using Signed Representations

In this section we analyze a selection of scalar multiplication methods which use a *signed-digit representation* (defined below) of the scalar k. In $E(\mathbb{F}_p)$ one has the advantage that inversion can de done very efficiently.
Indeed, if $P = (x, y) \in E(\mathbb{F}_p)$, we have $-P = (x, -y)$, so inverting a point is computationally equivalent to performing a negation modulo p – the cost of which is negligible in efficient field implementations (cf. Section 5.2.1). By allowing negative coefficients in the representation of k and using the fast inversion in $E(\mathbb{F}_p)$, one can achieve faster scalar multiplication than what we have seen among the binary methods in Section 3.1.

Example 3.4. We wish to compute $[2^s - 1]P$ for some $s > 1$. Doing this using Algorithm 1 requires $(s-1) \cdot$ `ECDBL` and $(s-1) \cdot$ `ECADD`. If one computes $[2^s - 1]P$ as $[2^s]P \oplus (-P)$, the calculation only requires $s \cdot$ `ECDBL` and one `ECADD`.

◦

From Example 3.4 we see that it can be advantageous to have a representation of the scalar at hand which allows negative digits. This leads to the following definition:

Definition 3.1 (Signed digit representation)**.** A *signed digit representation* of an integer k to the base b is an ordered sequence of integers $d_0, \ldots, d_{m-1}$ with $|d_i| < b$ for $i = 0, \ldots, m-1$ such that

$$k = \sum_{i=0}^{m-1} d_i b^i.$$

◦

Signed digit representations are not unique. For instance,

$$23 = (1100\overline{1})_2 = (11\overline{1}\,\overline{1}1)_2,$$

where $\overline{1} = -1$. To get a unique representation one has to introduce some additional conditions on the representation:

Definition 3.2 (Non-adjacent form)**.** A *non-adjacent form* (NAF) of an integer k is a signed-binary representation of k to the base $b = 2$ such that $d_i d_{i+1} = 0$ for $i \geq 0$. The NAF is written $(d_{m-1} \cdots d_0)_{\mathrm{NAF}}$.

◦

Proofs of existence and uniqueness of the NAF of k can be found in [MS04] by Muir & Stinson. They also prove that the Hamming weight of the NAF of an integer k is minimal among all signed digit representations of k and that the number of bits in the NAF of k is at most one more than the number of bits in the binary representation of k. Several other results applying to the NAF of integers are also proven in [MS04]. Algorithm 4 computes the NAF of an integer. In line 4 of Algorithm 4, mods denotes the signed residue with minimal absolute

---

Algorithm 4 Generation of the non-adjacent form (right-to-left version)

Input: An integer $k = (k_{l-1} \cdots k_0)_2$.
Output: The NAF $k = (d_l \cdots d_0)_{\mathrm{NAF}}$.

1: $i \leftarrow 0$; $d \leftarrow k$;
2: **while** $d > 0$ **do**
3:     **if** d is odd **then**
4:         $d_i \leftarrow d$ mods 4;
5:         $d \leftarrow d - d_i$;
6:     **else**
7:         $d_i \leftarrow 0$;
8:     **end if**
9:     $d \leftarrow \frac{d}{2}$;
10:     $i \leftarrow i + 1$;
11: **end while**
12: **return** $(d_l \cdots d_0)_{NAF}$;

---

value. When d is odd, we have either $d \equiv 1 \equiv -3 \mod 4$ or $d \equiv 3 \equiv -1 \mod 4$. In the former case, d mods $4 = 1$, and in the latter case, d mods $4 = -1$, so the operation is well-defined, when d is odd. A proof of correctness of Algorithm 4 can be found in [MS04].

If d is odd in line 3, the bit d_i is assigned the value 1 or $\overline{1}$, depending on whether the two least significant bits of d are 01 or 11 respectively. In both cases, the value of d is decremented in line 5 such that d becomes divisible by four and d is even at the end of the iteration in line 11. If, on the other hand, d is even in line 3, the bit d_i is assigned the value 0 in line 7. One can see that the name "non-adjacent form" is justified, as two non-zero digits cannot be adjacent in the output.

Assume that k is random and unbounded, and that the k_i are uniformly distributed and independently drawn. The process of generating a NAF can be interpreted as a random process $M = (X_n)_{\mathbb{N}_0}$ with state space $\mathcal{S} = \{0, *\}$, where $*$ symbolizes 1 or $\overline{1}$. The conditional distribution of X_{n+1} is

$$
\begin{aligned}
P(X_{n+1} = 0 \mid X_n = 0) &= \frac{1}{2} \\
P(X_{n+1} = * \mid X_n = 0) &= \frac{1}{2} \\
P(X_{n+1} = 0 \mid X_n = *) &= 1 \\
P(X_{n+1} = * \mid X_n = *) &= 0.
\end{aligned}
$$

As these probabilities are valid for any $n > 0$, the process M is a homogeneous random process. Furthermore, we have that, for any $n > 0$, the value of X_{n+1} only depends on the value of X_n, so M is a Markov chain (cf. Appendix A). The transition matrix is

$$T = \begin{bmatrix} \frac{1}{2} & \frac{1}{2} \\ 1 & 0 \end{bmatrix},$$

and the transition graph is shown in Figure 3.2.

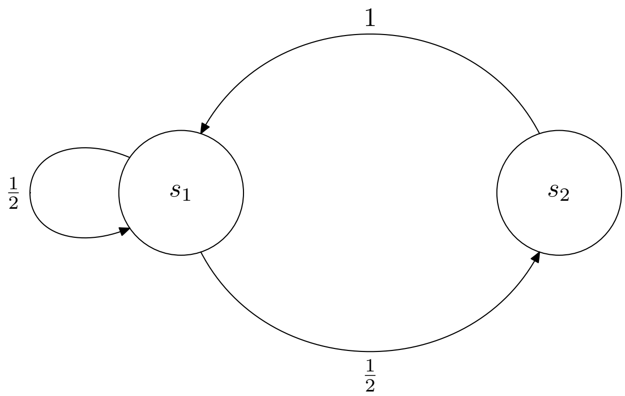


Figure 3.3: The figure shows the transition graph for the Markov chain corresponding to the process of generating a NAF.

The initial distribution is $\mu^{(0)} = (0, 1)$, and a stationary distribution for M is $\pi = (\frac{2}{3}, \frac{1}{3})$. As M is irreducible and aperiodic, Theorem A.3 in Appendix A implies that $\mu^{(n)}$ converges to π in total variation. This means that, for sufficiently large k, we can assume the Hamming weight of k in NAF representation to be $\frac{1}{3}$ of the number of bits on average. We sum up these considerations in the following proposition:

Proposition 3.4 (Hamming weight of the NAF)**.** *Let ν be the average number of non-zero bits in the NAF of a random positive integer $k = (k_{m-1} \cdots k_0)_{NAF}$. Then,*

$$\nu \approx \frac{m}{3}.$$

In the sequel we will assume that the NAF of an integer k is *always* one bit longer than the binary representation. Therefore, we have that

$$k = (k_{l-1} \cdots k_0)_2 = (d_l \cdots d_0)_{NAF}$$

for suitable $d_0, \ldots, d_l$, where d_l might be zero.

3.2.1 The Addition-subtraction Method

Algorithm 4 processes the bits of k from right to left. There exists a left-to-right variant of the algorithm for computing the NAF of an integer. The left-to-right version is used in the *addition-subtraction method* for performing scalar multiplication in $E(\mathbb{F}_p)$. The Addition-subtraction method is recommended in [P1304] and is shown in Algorithm 5. As the scope of this text does not encompass

Algorithm 5 Addition-subtraction method (including integer recoding)

Input: An affine point $P \in E(\mathbb{F}_p)$ and $k = (k_{l-1} \cdots k_0)_2$.
Output: $[k]P \in E(\mathbb{F}_p)$.

$(h_l h_{l-1} \cdots h_0) \leftarrow 3k$; $//h_l = 1$
$(k_l k_{l-1} \cdots k_0) \leftarrow k$; $//k_l = 0$
$Q \leftarrow P$;
$i \leftarrow l - 2$;
while $i \geq 1$ **do**
  $Q \leftarrow [2]Q$;
  if $h_i = 1$ and $k_i = 0$ **then**
    $Q \leftarrow Q \oplus P$;
  end if
  if $h_i = 0$ and $k_i = 1$ **then**
    $Q \leftarrow Q \oplus (-P)$;
  end if
  $i \leftarrow i - 1$;
end while
return Q

optimization of integer recoding[3], the addition-subtraction method is rewritten to exclude determining the NAF of k. The result is shown in Algorithm 6. Proofs of correctness of these algorithms are analogous to the proof that Algorithm 1 is correct.

[3]Integer recoding is the process of converting integers from one representation to another.

Algorithm 6 Addition-subtraction method

Input: An affine point $P \in E(\mathbb{F}_p)$ and $k = (d_l \cdots d_0)_{\mathrm{NAF}}$.
Output: $[k]P \in E(\mathbb{F}_p)$.

```
1: Q ← O;
2: i ← l;
3: while i ≥ 0 do
4:     Q ← [2]Q;
5:     if d_i ≠ 0 then
6:         Q ← Q ⊕ [d_i]P;
7:     end if
8:     i ← i − 1;
9: end while
10: return Q
```

We assume that the very first doubling in line 4 is not performed, as $Q = \mathcal{O}$. Similarly, we assume that the very first addition in line 6 is not performed, as $\mathcal{O} \oplus [d_i]P = [d_i]P$. As in Algorithm 1, we see that an addition is performed if, and only if, $d_i \neq 0$. Using Proposition 3.4, we get:

Proposition 3.5 (Complexity of the addition-subtraction method)**.** *For a sufficiently large scalar $k = (k_{l-1} \cdots k_0)_2 = (d_l \cdots d_0)_{NAF}$, one has on average*

$$t(\textit{Algorithm 6}) = l\,\texttt{ECDBL} + \frac{l}{3}\,\texttt{ECADD}.$$

The addition-subtraction method is the scalar multiplication method used in the test implementation developed by IBM (source code is enclosed in Appendix C.3.1).

Example 3.5. If we return to our example of $l = 192$, we see that Algorithm 6 requires

$$192 \cdot \texttt{ECDBL} + 64 \cdot \texttt{ECADD}$$

on average.

$\circ$

The average number of operations in Example 3.5 is not impressively low compared to the sliding-window method (Algorithm 3). However, the addition-subtraction method has the advantage of needing no precomputations. Furthermore, the approach can be generalized to give a substantial reduction in the number of operations.

3.2.2 The Width-w NAF Method

This section presents a scalar multiplication method which has a lower requirement than any of the methods discussed in Sections 3.1.1-3.2.1. The method can be seen as a combination of the sliding-window method and the addition-subtraction method. It relies on a generalized NAF representation of the scalar.

Definition 3.3 (Width-w non-adjacent form)**.** Let $w > 1$, and let k be a positive integer. Let k be written as

$$k = \sum_{i=0}^{m-1} d_i 2^i, \tag{3.3}$$

where

(i) $d_i = 0$ or d_i is odd for $i = 0, \ldots, m-1$.

(ii) $|d_i| < 2^{w-1}$ for $i = 0, \ldots, m-1$.

(iii) Among any sequence of w consecutive coefficients at most one is non-zero.

The representation in equation (3.3) is called a *width-w non-adjacent form* (NAF_w), and we write

$$k = (d_{m-1} \cdots d_0)_{\text{NAF}_w}.$$

$\circ$

Remark 3.2. The representation in Definition 3.3 can also be described in another way. If we write k as

$$k = 2^{\kappa_0}(2^{\kappa_1}(\cdots 2^{\kappa_{\nu-1}}(2^{\kappa_\nu} W_\nu + W_{\nu-1}) \cdots + W_1) + W_0)$$

with $W_\nu > 0$, conditions (i) and (ii) in definition 3.3 correspond to W_i being odd and $-2^{w-1} + 1 \leq W_i \leq 2^{w-1} - 1$ for all i. Condition (iii) corresponds to $\kappa_0 \geq 0$ and $\kappa_i \geq w$ for all $i \geq 1$. For instance, if $k = (700000\bar{3}0001)_{\text{NAF}_w}$ with $w = 4$, we have $k = 2^0(2^4(2^6 \cdot 7 - 3) + 1)$.

$\circ$

For $w = 2$, the NAF_w is simply the ordinary NAF discussed earlier. For any integer $w > 1$, the NAF_w shares the following properties with the NAF:

- ◇ Every integer has a unique NAF_w.
- ◇ The NAF_w of an integer k is at most one bit longer than the binary representation of k.

For proofs that these properties hold see [MS04]. Additionally, Avanzi [Ava05] shows that the NAF_w representation is a recoding of smallest Hamming weight among all recodings with coefficients smaller than 2^{w-1} in absolute value.

Like in the case of the NAF, we assume that the NAF_w of a positive integer k is *always* one bit longer than the binary representation of k. Therefore, the most significant bit d_l of

$$k = (d_l \cdots d_0)_{\mathrm{NAF}_w}$$

might be zero. A method for generating the NAF_w is shown in Algorithm 7.

Algorithm 7 Generation of the width-w non-adjacent form.

Input: Integers $k = (k_{l-1} \cdots k_0)_2$ and $w > 1$.
Output: $k = (d_l \cdots d_0)_{\mathrm{NAF}_w}$.

1: $i \leftarrow 0$; $d \leftarrow k$;
2: **while** $d > 0$ **do**
3:   **if** d is odd **then**
4:     $d_i \leftarrow d$ mods 2^w; //d is odd, so mods is well-defined.
5:     $d \leftarrow d - d_i$;
6:   **else**
7:     $d_i \leftarrow 0$;
8:   **end if**
9:   $d \leftarrow \frac{d}{2}$;
10:   $i \leftarrow i + 1$;
11: **end while**
12: **return** $(d_l \cdots d_0)_{NAF}$

Proof of correctness: Lines 4, 5 and 9 ensure that d is reduced in each iteration, so the algorithm eventually terminates. We now verify that the output of Algorithm 7 satisfies the conditions in Definition 3.3. The assignment $d_i \leftarrow d$ mods 2^w in line 4 (where d is odd) ensures that every non-zero d_i is odd and less than 2^{w-1} in absolute value.

In the i'th iteration of the main loop in lines 2-11, we assume that d_i is assigned a non-zero value. Subsequently, d becomes a multiple of 2^w in line 5 and now has the form

$$d = (\cdots \overbrace{0 \cdots 0}^{w})_2.$$

The assignment $d \leftarrow \frac{d}{2}$ gives

$$d = (\cdots \overbrace{0 \cdots 0}^{w-1})_2.$$

If $d \neq 0$, the main loop will execute $w - 1$ times and output $w - 1$ zero-valued bits. If $d = 0$, d_i is the most significant digit, and the algorithm terminates. These considerations ensure that condition (iii) in Definition 3.3 is satisfied.

All that remains is to verify that Algorithm 7 actually outputs a value which equals k. To see that it is so, notice that the algorithm maintains the loop invariant:

$\mathcal{L}$: *At line 2 of algorithm 7,*
$k = 2^i d + \sum_{j=0}^{i-1} d_j 2^j$.

As $i = 0$ and $d = k$ prior to the first iteration, the statement $\mathcal{L}$ holds at this point. Assume that $\mathcal{L}$ holds for some $i > 0$. We want to show that $\mathcal{L}$ holds for $i + 1$. If d is even, $d_{i-1} = 0$ after the incrementation of i, and d is assigned the value $\frac{d}{2}$, so $\mathcal{L}$ holds prior to the next iteration. Assume that d is odd. We know that

$$\begin{aligned} k &= 2^i d + \sum_{j=0}^{i-1} d_j 2^j \\ &= 2\left(\frac{2^i(d - (d \text{ mods } 2^w))}{2}\right) + \sum_{j=0}^{i-1} d_j 2^j + 2^i(d \text{ mods } 2). \end{aligned} \tag{3.4}$$

After the assignments in lines 5 and 9, equation (3.4) becomes

$$k = 2^i d + \sum_{j=0}^{i} d_j 2^j,$$

so the invariant is restored, when i is incremented.

At the end of the algorithm $d = 0$, so the invariant ensures that $k = \sum_{j=0}^{l} d_j 2^j$, with the convention that $d_j = 0$ for j greater than or equal to the final value of i.

■

We define the *density* of a representation of k to be the Hamming weight of the representation divided by the number of bits in the representation. The average density of a binary representation is $\frac{1}{2}$, and it turns out that the average density of a NAF_w is less than $\frac{1}{2}$. In fact, the following result holds:

Proposition 3.6. *Let k be a positive integer. The density of the width-w NAF representation of k is $\frac{1}{w+1}$ on average.*

Proof: We know that Algorithm 7 computes the unique NAF_w representation of k. Algorithm 7 can be viewed as a homogeneous random process $(X_n)_{\mathbb{N}_0}$ with state space $\mathcal{S} = \{s_1, s_2\}$, where

$$s_1 = 0 \text{ (a single bit) and } s_2 = \overbrace{0 \cdots 0*}^{w}.$$

Here, we denote by $*$ a non-zero number with absolute value less than 2^{w-1}. Adapting this view, one should keep in mind that for any k with a finite number of bits the random process is finite, and the last state does not necessarily have to be either s_1 or s_2. We assume that k is unbounded.

The event $X_n = s_2$ corresponds to d being odd in line 3 of Algorithm 7. The probability of this to occur equals the probability of the least significant bit of d being equal to one, so $P(X_n = s_1) = \frac{1}{2}$. Therefore, $P(X_n = s_2) = \frac{1}{2}$, and we get a density of

$$\frac{P(X_n = s_2)}{P(X_n = s_1) \cdot 1 + P(X_n = s_2) \cdot w} = \frac{1}{w+1}.$$

■

In later sections we will be interested in knowing:

(a) The average length of the first sequence of zeroes produced by Algorithm 7.

(b) The average length of sequences of consecutive zeroes produced by Algorithm 7. These sequences are also known as *zero-runs*.

Assume that k is unbounded. To find the length in (a), let X be a random variable describing the length of the first (possibly empty) sequence of consecutive zeroes produced by Algorithm 7. This means that $X \in \mathbb{N}_0$. The event $X = 0$ corresponds to k being odd, so $P(X = 0) = \frac{1}{2}$. The event $X = 1$ corresponds to k having the form $k = (\cdots 10)_2$, so $P(X = 1) = \frac{1}{4}$. Similarly, one can see that for all $j \geq 0$ we have $P(X = j) = \frac{1}{2^{j+1}}$. This gives an expectation of

$$EX = \sum_{j=0}^{\infty} \frac{j}{2^{j+1}} = 1,$$

so on average we expect Algorithm 7 to output one zero to begin with.

To find the length in (b), we let Y be a random variable describing the number of zeroes in a zero-run (apart from the $w-1$ zeros we know for sure to be in there), so $Y \in \mathbb{N}_0$. The event $Y = 0$ corresponds to d having the form

$$d = (\cdots 1 \overbrace{0 \cdots 0}^{w-1})$$

after the assignment in line 5. As we know that the $w-1$ least significant bits of d are zero, we have $P(Y = 0) = \frac{1}{2}$. Similarly, the event $Y = 1$ corresponds to d having the form

$$d = (\cdots 10 \overbrace{0 \cdots 0}^{w-1})$$

after the assignment in line 5. Therefore, $P(Y = 1) = \frac{1}{4}$. In general, Y has the same distribution as X, so $EY = 1$. Therefore, we will expect a zero-run (apart from the first one) to have length w on average. We summarize these observations in the following proposition:

Proposition 3.7 (Length of zero-runs)**.** *For large k, one has on average:*

(i) The length of the first (possibly empty) zero-run produced by Algorithm 7 is 1.

(ii) The length of zero-runs other than the first one produced by Algorithm 7 is w.

As the number of `ECADD` performed in the scalar multiplication algorithms we have considered so far depends on the Hamming weight (and, thereby, on the density) of the scalar, a scalar in NAF_w can be used to reduce the number of elliptic curve operations involved in scalar multiplication. Algorithm 8 shows the details of the method.

Algorithm 8 Width-w NAF scalar multiplication.

Input: An affine point $P \in E(\mathbb{F}_p)$ and $k = (d_l \cdots d_0)_{\text{NAF}_w}$.
Output: $[k]P \in E(\mathbb{F}_p)$.

1: Compute the odd multiples $[\pm 3]P, [\pm 5]P, \ldots, [\pm(2^{w-1} - 1)]P$.
2: $Q \leftarrow \mathcal{O}$; $i \leftarrow l$;
3: **while** $i \geq 0$ **do**
4: $\quad Q \leftarrow [2]Q$;
5: $\quad$ **if** $d_i \neq 0$ **then**
6: $\quad\quad Q \leftarrow Q \oplus [d_i]P$; //If $d_i \neq 0$, it is odd, and $[d_i]P$ has been precomputed.
7: $\quad$ **end if**
8: $\quad i \leftarrow i - 1$;
9: **end while**
10: **return** Q

Proof of correctness: In line 8 the value of i is decremented, and when $i = 0$ the algorithm terminates. Algorithm 8 maintains the loop invariant

$$\mathcal{L}\text{: In line 3, we have } Q = \left[\sum_{j=i+1}^{l} d_j 2^{j-i-1}\right] P.$$

The rest of the proof is identical to the proof of correctness of Algorithm 1 – except for the use of the identity

$$[(d_l \cdots d_{i+1} d_i)_{NAF_w}]P = [2]([(d_l \cdots d_{i+1})_{NAF_w}]P) + [d_i]P$$

instead of equation (3.1).

■

Algorithm 7 performs one `ECDBL` for each bit in the representation of k. An `ECADD` is performed for each non-zero bit in the representation. We assume that the first `ECDBL` in line 4 and the first `ECADD` in line 6 are not performed, as $Q = \mathcal{O}$. The precomputations require one ECDBL and $(2^{w-2} - 1) \cdot$ ECADD. Assuming that the NAF_w representation of k is always one bit longer than the binary representation, Proposition 3.6 gives

Proposition 3.8 (Requirement of the width-w NAF scalar multiplication)**.** *For input $k = (k_{l-1} \cdots k_0)_2$, we have*

$$t(\mathit{Algorithm\ 8}) = (l+1) \cdot \mathtt{ECDBL} + \left(2^{w-2} - 1 + \frac{l}{w+1}\right) \cdot \mathtt{ECADD}$$

on average. Algorithm 8 requires storage for $2^{w-2} - 1$ precomputed points.

As was the case with the 2^w-ary method and the sliding-window method, the number of operations performed in the NAF_w method depends on the value of w. Figure 3.4 shows the number of ECADD in the cases $l = 192$ and $l = 521$.

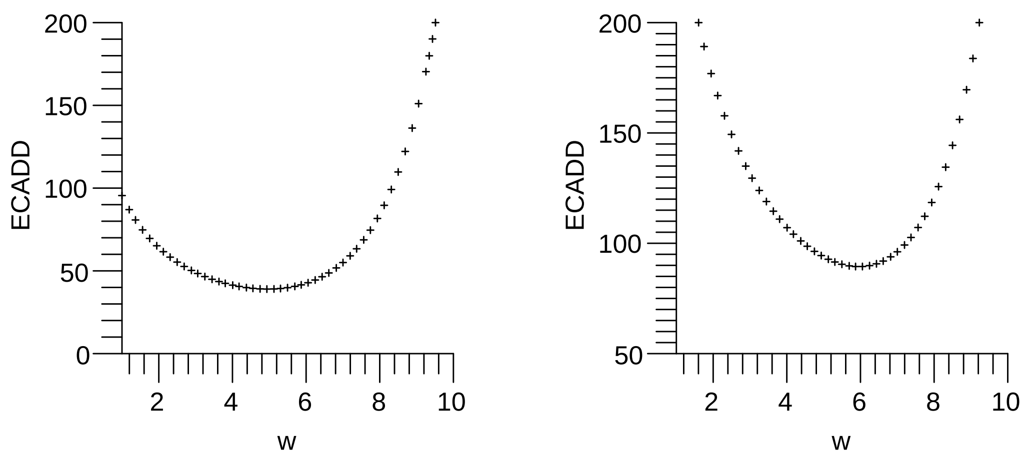


Figure 3.4: The plots show the number of ECADD performed by the NAF_w method for $l = 192$ and $l = 521$ respectively when $w \in [1, 10]$.

The expressions plotted in Figure 3.4 are minimized for $w = 5$ and $w = 6$ respectively. Table 3.3 shows a selection of optimal values of w.

l	$[41, 119]$	$[120, 335]$	$[336, 895]$
w	4	5	6

Table 3.3: The table shows a selection of optimal values of w corresponding to different values of l.

Example 3.6. In the case $l = 192$, the value $w = 5$ is optimal, so Algorithm 7 requires

$$193 \cdot \texttt{ECDBL} + 39 \cdot \texttt{ECADD}$$

on average. Compared to the sliding-window method, the NAF_w method saves 6 `ECADD`, while it introduces an extra `ECDBL`. The NAF_w method needs to store only 7 precomputed points (one only needs to store the even multiples) instead of the 15 precomputed points required by the sliding-window method. As long as the cost of a `ECDBL` is strictly less than that of 6 `ECADD`, Algorithm 8 is the better choice.

3.3 Comparison and Conclusion

In order to be able to compare the different scalar multiplication algorithms, we count the total number of `ECADD` required on average by the methods. We assume that all computations are done in affine coordinates. In Chapter 4 we will see that it is reasonable to assume that an `ECDBL` corresponds to 1.05 `ECADD`. Figure 3.5 shows the average number of `ECADD` required by the different methods as functions of l (the length of the binary representation of k), assuming that $t(\texttt{ECDBL}) = 1.05 \cdot t(\texttt{ECADD})$. The plots in Figure 3.5 correspond to an optimal choice of w where this value is used (in the 2^w-ary, sliding-window and NAF_w methods, cf. Tables 3.1, 3.2 and 3.3).

The plots in Figure 3.5 show that the NAF_w method is the better choice among the methods presented in Sections 3.1 and 3.2. Also, Algorithm 8 requires less storage for precomputed values than Algorithms 2 and 3 do.

Remark 3.3. There is a generalization of the sliding-window method to a NAF representation of the scalar. This generalization offers no improvement over the NAF_w method and is not as easily implemented. An analysis can be found in [Sem04].

∘

We now draw conclusions based on the observations made in this chapter. In Sections 3.1 and 3.2 we have presented and analyzed a selection of algorithms for performing scalar multiplication on an elliptic curve. The sliding-window method (Algorithm 3) was superior among the methods using an unsigned representation of the scalar, while the NAF_w method (Algorithm 8) was the better choice among the methods using a signed representation. The NAF_w method was even better than the sliding-window method in the case of k being a 192-bit integer (cf. Example 3.1, 3.2, 3.3, 3.5 and 3.6). We have seen that the NAF_w method is actually superior for all applied values of l. The algorithm uses storage for precomputed

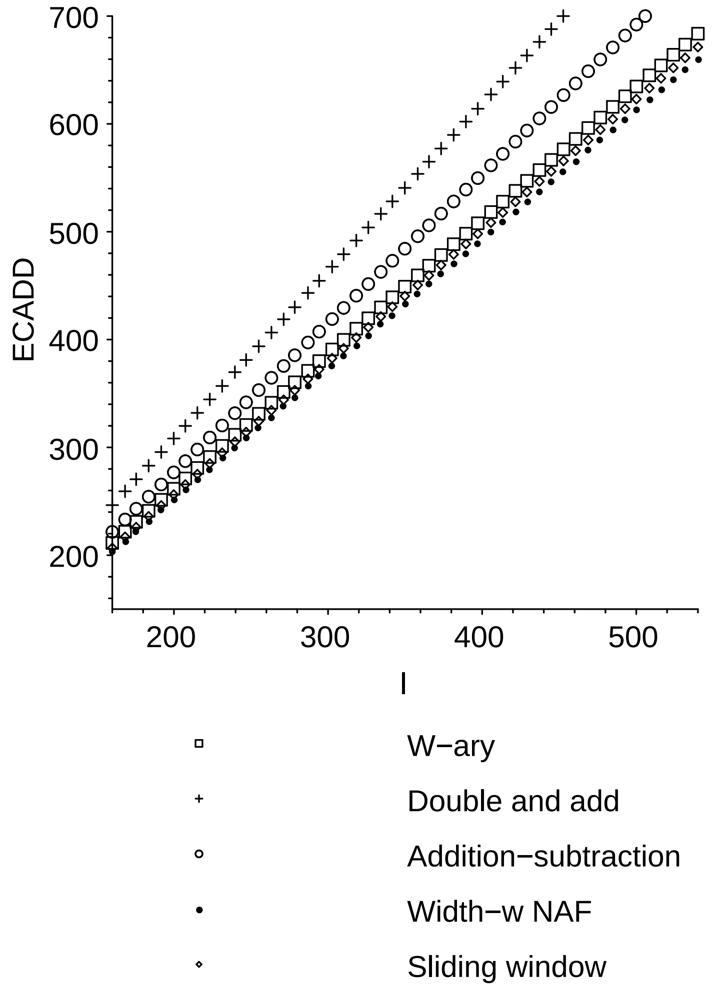


Figure 3.5: The plot shows the number of ECADD required by the different scalar multiplication methods, assuming that $t(\texttt{ECDBL}) = 1.05 \cdot t(\texttt{ECADD})$.

values, but the storage requirement is less than what is the case for other methods which uses precomputation, and we conclude that the storage requirement is acceptable. Therefore, Algorithm 8 should be used for scalar multiplication.

Chapter 4

Coordinate Representations

In Chapter 3 we dealt with the task of minimizing the number of elliptic curve additions/doublings performed during scalar multiplication. This chapter deals with minimizing the number of field operations involved in the individual additions and doublings. Doing so requires some knowledge of coordinate representations of elliptic curves. This text covers five representations: Projective, affine and Jacobian coordinates (see Section 1.1 for details on these representations), and the Jacobian variants *modified Jacobian coordinates* and *Chudnovsky-Jacobian coordinates*. We present formulas for addition and doubling on the NIST curves in all five representations. In the cases of projective, affine and Jacobian coordinates, the formulas are almost identical to the general formulas from Section 1.2.1. However, as the NIST curves have $a = -3$, there are differences affecting the number of required field operations. Furthermore, we examine the advantages of using a mixture of the aforementioned representations during scalar multiplication.

When evaluating the formulas for addition and doubling in different coordinates, we let M, S and I denote multiplication, squaring and inversion modulo p respectively. We assume that the time required to perform an addition, subtraction, comparison or negation in $\mathbb{F}_p$ is negligible (this assumption is discussed in Section 5.2.1).

In the sequel we assume that the NAF_w of a positive integer k is always one bit longer than the binary representation $k = (k_{l-1} \cdots k_0)_2$ and that the most significant bit of $k = (d_l \cdots d_0)_{\mathrm{NAF}_w}$ is positive.

4.1 Fixed Representations

In this section we present formulas for addition and doubling on the NIST curves using a fixed coordinate representation. For each operation we count the number of required field operations.

4.1.1 Projective Coordinates

The equation for E is

$$E : Y^2Z = X^3 - 3XZ^2 + bZ^3.$$

The group of rational points is $(E(\mathbb{F}_p), \oplus)$ with neutral element $(0:1:0)$. Let $P, Q \in E(\mathbb{F}_p)$ with $P = (X_1 : Y_1 : Z_1)$ and $Q = (X_2 : Y_2 : Z_2)$ with $P \neq Q$. The inverse of P is $-P = (X_1 : -Y_1 : Z_1)$. Formulas for $P \oplus Q = (X_3 : Y_3 : Z_3)$ and $[2]P = (X_4 : Y_4 : Z_4)$ are:

Addition:

Set $A = Y_2Z_1 - Y_1Z_2$, $B = X_2Z_1 - X_1Z_2$ and $C = A^2Z_1Z_2 - B^3 - 2B^2X_1Z_2$.

Then, $X_3 = BC$, $Y_3 = A(B^2X_1Z_2 - C) - B^3Y_1Z_2$ and $Z_3 = B^3Z_1Z_2$.

Doubling:

Set $A = 3(X_1^2 - Z_1^2)$, $B = Y_1Z_1$, $C = X_1Y_1B$ and $D = A^2 - 8C$.

Then, $X_4 = 2BD$, $Y_4 = A(4C - D) - 8Y_1^2B^2$ and $Z_4 = 8B^3$.

As one can check in the formulas, an addition requires 12 multiplications and 2 squarings, written as $12M + 2S$, while a doubling requires $7M + 5S$.

Remark 4.1. If $Z_1 = 1$, the requirement reduces to $9M + 2S$ for addition and $5M + 4S$ for doubling. If $Z_1 = Z_2 = 1$, addition drops to $5M + 2S$. These special cases will be of interest later, when we discuss the use of mixed coordinates. For now, the reader should simply note their existence.

4.1.2 Affine Coordinates

The equation for E is

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

The group of rational points is $(E(\mathbb{F}_p, \oplus))$ with neutral element $\mathcal{O}$.
Let $P = (x_1, y_1)$ and $Q = (x_2, y_2)$ be affine points on E with $P \neq \pm Q$. The inverse of P is $-P = (x_1, -y_1)$. Formulas for $P \oplus Q = (x_3, y_3)$ and $[2]P = (x_4, y_4)$ are:

Addition:

Set $\lambda = \frac{y_1 - y_2}{x_1 - x_2}$.

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

Doubling:

Set $\lambda = \frac{3x_1^2 - 3}{2y_1}$.

Then, $x_4 = \lambda^2 - 2x_1$ and $y_4 = \lambda(x_1 - x_4) - y_1$.

An addition requires $I + 2M + S$, while a doubling requires $I + 2M + 2S$.

4.1.3 Jacobian Coordinates

The equation for E is

$$E : Y^2 = X^3 - 3XZ^4 + bZ^6.$$

The group of rational points is $(E(\mathbb{F}_p), \oplus)$ with neutral element $(1 : 1 : 0)$. Let $P = (\xi_1 : \eta_1 : \zeta_1)$ and $Q = (\xi_2 : \eta_2 : \zeta_2)$ be $\mathbb{F}_p$-rational points on E and assume that $P \neq Q$. The inverse of P is $-P = (\xi_1 : -\eta_1 : \zeta_1)$. Formulas for $P \oplus Q = (\xi_3 : \eta_3 : \zeta_3)$ and $[2]P = (\xi_4 : \eta_4 : \zeta_4)$ are:

Addition:

Set $A = \xi_1\zeta_2^2$, $B = \xi_2\zeta_1^2$, $C = \eta_1\zeta_2^3$, $D = \eta_2\zeta_1^3$, $E = B - A$ and $F = D - C$.

Then, $\xi_3 = -E^3 - 2AE^2 + F^2$, $\eta_3 = -CE^3 + F(AE^2 - \xi_3)$ and $\zeta_3 = \zeta_1\zeta_2 E$.

Doubling:

Set $A = 4\xi_1\eta_1^2$ and $B = 3(\xi_1 - \zeta_1^2)(\xi_1 + \zeta_1^2)$.

Then, $\xi_4 = -2A + B^2$, $\eta_4 = -8\eta_1^4 + B(A - \xi_4)$ and $\zeta_4 = 2\eta_1\zeta_1$.

An addition requires $12M + 4S$, and a doubling requires $4M + 4S$.

Remark 4.2. If $\xi_1 = 1$, the cost of an addition and a doubling reduces to $8M+3S$ and $2M + 4S$ respectively.

4.1.3.1 Chudnovsky-Jacobian Coordinates

At this point we have seen that Jacobian coordinates provide faster doublings, but slower additions, than projective coordinates. Addition in Jacobian coordinates can be sped up by changing the internal representation of a point P from $P = (\xi : \eta : \zeta)$ to $P = (\xi : \eta : \zeta : \zeta^2 : \zeta^3)$. The latter representation is called the *Chudnovsky-Jacobian* coordinates of P. More storage is required, but by using Chudnovsky-Jacobian coordinates one achieves a cost of $11M + 3S$ for addition, while the cost of a doubling increases to $7M + 3S$.

4.1.3.2 Modified Jacobian Coordinates

Assume that the coefficient a can be any element of $\mathbb{F}_p$ and internally represent a Jacobian point $P = (\xi : \eta : \zeta)$ as a quadruple $(\xi : \eta : \zeta : a\zeta^4)$. This quadruple is called the *modified Jacobian* coordinates of P. For $P = (\xi_1 : \eta_1 : \zeta_1 : a\zeta_1^4)$ and $Q = (\xi_2 : \eta_2 : \zeta_2 : a\zeta_2^4)$ with $P \neq Q$, this gives the following formulas for $P \oplus Q = (\xi_3 : \eta_3 : \zeta_3 : a\zeta_3^4)$ and $[2]P = (\xi_4 : \eta_4 : \zeta_4 : a\zeta_4^4)$:

Addition:

Set $A = \xi_1\zeta_2^2$, $B = \xi_2\zeta_1^2$, $C = \eta_1\zeta_2^3$, $D = \eta_2\zeta_1^3$, $E = B - A$ and $F = D - C$.

Then, $\xi_3 = -E^3 - 2AE^2 + F^2$, $\eta_3 = -CE^3 + F(AE^2 - \xi_3)$ and $\zeta_3 = \zeta_1\zeta_2 E$.

Doubling:

Set $A = 4\xi_1\eta_1^2$, $B = 3\xi_1^2 + a\zeta_1^4$ and $C = 8\eta_1^4$.

Then, $\xi_3 = -2A + B^2$, $\eta_3 = B(A - \xi_3) - C$, $\zeta_3 = 2\eta_1\zeta_1$ and $-3\zeta_3^4 = 2C(-3\zeta_1^4)$.

The formula for addition is identical to the one in Section 4.1.3, but calculating the element $a\zeta_3^4$ requires $1M + 2S$ ($2S$ for the NIST curves). Thus, the total cost of addition is $13M + 6S$ ($12M + 6S$ for the NIST curves). Doubling requires $4M + 4S$, regardless of the value of a, making modified Jacobian coordinates the better choice for doublings unless $a = -3$, in which case Jacobian coordinates and modified Jacobian coordinates are equally good.

4.2 Mixed Representations

Let $\mathcal{A}$, $\mathcal{P}$, $\mathcal{J}$, $\mathcal{J}^c$ and $\mathcal{J}^m$ symbolize affine, projective, Jacobian, Chudnovsky-Jacobian and modified Jacobian coordinates respectively. We have seen, in Sections 4.1.1, 4.1.2 and 4.1.3, that the choice of coordinate representation affects the number of field operations involved in scalar multiplication. Therefore, it is natural to ask which coordinate system minimizes the number of field operations. Unfortunately, the question is not as easy to answer as it is to ask. One coordinate representation may be superior when performing doublings, but not when performing additions (e.g. $\mathcal{J}$) or vice versa (e.g. $\mathcal{P}$).

Instead of trying to select one fixed representation among the available ones, we will aim at *combining* the representations. As suggested in [CMO98], one can use the individual strengths of the different representations in a combined manner. The idea is to perform each type of operation (`ECADD` or `ECDBL`) in the optimal representation for that particular operation. The goal is to have a strategy for the process of scalar multiplication defining exactly which coordinate representation should be used at a given stage of the process.

Changing between representation is done during execution of the elliptic curve operations. Let "→" symbolize any action which modifies a point on an elliptic curve (for instance performing a doubling or disregarding one or more coordinates of the point). If we wish to double a point (x, y) in $\mathcal{A}$ and express the result in $\mathcal{J}$, we do as follows:

$$\overset{\mathcal{A}}{(x,y)} \longrightarrow (x : \overset{\mathcal{J}}{y} : 1) \longrightarrow [2](x : \overset{\mathcal{J}}{y} : 1) = (\xi : \overset{\mathcal{J}}{\eta} : \zeta) \ . \tag{4.1}$$

The doubling on the left hand side of the equation in (4.1) is performed in $\mathcal{J}$. Similarly, we can add an affine point (x, y) to a Jacobian point $(\xi : \eta : \zeta)$ and express the result in $\mathcal{J}^c$ by performing the following steps:

$$\begin{aligned} \overset{\mathcal{A}}{(x,y)} &\longrightarrow (x : y : \overset{\mathcal{J}^c}{1} : 1 : 1) \\ &\longrightarrow (x : y : \overset{\mathcal{J}^c}{1} : 1 : 1) \oplus (\xi : \overset{\mathcal{J}}{\eta} : \zeta) \\ &\longrightarrow (x : y : \overset{\mathcal{J}^c}{1} : 1 : 1) \oplus (\xi : \eta : \overset{\mathcal{J}^c}{\zeta} : \zeta^2 : \zeta^3) \\ &= (\xi' : \eta' : \overset{\mathcal{J}^c}{\zeta'} : \zeta'^2 : \zeta'^3) \ . \end{aligned} \tag{4.2}$$

In both cases the technique is the same: We represent all points in the coordinates of the "target system" and perform the operation in that system. However, not all conversions between systems are equally simple. While conversions from $\mathcal{A}$ to $\mathcal{P}$ and from $\mathcal{A}$ to $\mathcal{J}$ are done by performing $(x, y) \to (x, y, 1)$, and conversions from $\mathcal{J}^c$ or $\mathcal{J}^m$ to $\mathcal{J}$ are done by disregarding one or more coordinates, conversions between $\mathcal{P}$ and $\mathcal{J}$ require inverting and multiplying elements of $\mathbb{F}_p$. Because of the overhead involved in the latter type of conversions, operations using a mixture of projective and Jacobian coordinates are not suitable for efficient implementations. Table 4.1 shows the cost of doubling and addition for the selection of combinations of coordinate systems upon which our remaining analysis is based. In Table 4.1, the notation

$$t(\mathcal{C}^1 + \mathcal{C}^2 = \mathcal{C}^3)$$

represents the field operations involved in adding a point in representation $\mathcal{C}^1$ to a point in representation $\mathcal{C}^2$ and expressing the result in representation $\mathcal{C}^3$. Similarly, the notation

$$t(2\mathcal{C}^1 = \mathcal{C}^2)$$

represents the field operations involved in doubling a point represented in $\mathcal{C}^1$ and expressing the result in the representation $\mathcal{C}^2$. The notations $t(2\mathcal{C})$ and $t(\mathcal{C}+\mathcal{C})$ denote the number of operations involved in doubling and addition respectively in a fixed representation $\mathcal{C}$.

Doubling			**Addition**		
Fixed:			Fixed:		
$t(2\mathcal{A})$	$=$	$I+2M+2S$	$t(\mathcal{A}+\mathcal{A})$	$=$	$I+2M+S$
$t(2\mathcal{P})$	$=$	$7M+5S$	$t(\mathcal{J}^m+\mathcal{J}^m)$	$=$	$12M+6S$
$t(2\mathcal{J}^c)$	$=$	$7M+3S$	$t(\mathcal{J}+\mathcal{J})$	$=$	$12M+4S$
$t(2\mathcal{J}^m)$	$=$	$4M+4S$	$t(\mathcal{P}+\mathcal{P})$	$=$	$12M+2S$
$t(2\mathcal{J})$	$=$	$4M+4S$	$t(\mathcal{J}^c+\mathcal{J}^c)$	$=$	$11M+3S$
Mixed:			Mixed:		
$t(2\mathcal{J}^m=\mathcal{J}^c)$	$=$	$4M+5S$	$t(\mathcal{J}^m+\mathcal{J}^c=\mathcal{J}^m)$	$=$	$11M+5S$
$t(2\mathcal{A}=\mathcal{P})$	$=$	$4M+4S$	$t(\mathcal{J}+\mathcal{J}^c=\mathcal{J}^m)$	$=$	$11M+5S$
$t(2\mathcal{A}=\mathcal{J}^c)$	$=$	$4M+3S$	$t(\mathcal{J}^c+\mathcal{J}^c=\mathcal{J}^m)$	$=$	$10M+4S$
$t(2\mathcal{J}^c=\mathcal{J})$	$=$	$4M+3S$	$t(\mathcal{J}^c+\mathcal{J}=\mathcal{J})$	$=$	$11M+3S$
$t(2\mathcal{J}^m=\mathcal{J})$	$=$	$3M+4S$	$t(\mathcal{J}+\mathcal{A}=\mathcal{J}^m)$	$=$	$8M+5S$
$t(2\mathcal{J}^m=\mathcal{J})$	$=$	$3M+4S$	$t(\mathcal{J}^m+\mathcal{A}=\mathcal{J}^m)$	$=$	$10M+3S$
$t(2\mathcal{A}=\mathcal{J})$	$=$	$2M+4S$	$t(\mathcal{J}^c+\mathcal{J}^c=\mathcal{J})$	$=$	$10M+2S$
			$t(\mathcal{J}^c+\mathcal{A}=\mathcal{J}^c)$	$=$	$8M+3S$
			$t(\mathcal{J}+\mathcal{A}=\mathcal{J})$	$=$	$8M+3S$
			$t(\mathcal{J}^m+\mathcal{A}=\mathcal{J})$	$=$	$8M+3S$
			$t(\mathcal{J}^c+\mathcal{A}=\mathcal{J}^m)$	$=$	$7M+4S$
			$t(\mathcal{A}+\mathcal{J}^c=\mathcal{J})$	$=$	$7M+2S$
			$t(\mathcal{A}+\mathcal{A}=\mathcal{J}^m)$	$=$	$4M+4S$
			$t(\mathcal{A}+\mathcal{A}=\mathcal{J}^c)$	$=$	$4M+2S$
			$t(\mathcal{A}+\mathcal{A}=\mathcal{J})$	$=$	$4M+2S$

Table 4.1: Number of field operations involved in `ECDBL` and `ECADD` using mixed coordinates.

Example 4.1. Assume that we are given points P, Q on E with $[4]P \neq Q, \mathcal{O}$. We wish to perform the following sequence of operations:

1) $P' := [2]P$.

2) $P'' := [2]P'$.

3) $P''' := P'' \oplus Q$.

Assume that $I = 16M$ and $M = S$, that Q is given in $\mathcal{A}$ and that P and P''' must be in the same representation. Which representations should we choose for P, P' and P'' in order to minimize the number of field operations? Choosing $\mathcal{A}$ as the representation for all points results in a cost of

$$\begin{aligned} 2t(2\mathcal{A}) + t(\mathcal{A} + \mathcal{A}) &= 2(I + 2M + 2S) + I + 2M + S \\ &= 3I + 4M + 5S \\ &= 57M. \end{aligned}$$

The question is: Can we do better? To answer this, we need to find coordinate systems $\mathcal{C}^1$, $\mathcal{C}^2$ and $\mathcal{C}^3$ such that

$$\begin{aligned} &t(2\mathcal{C}^1 = \mathcal{C}^2) + t(2\mathcal{C}^2 = \mathcal{C}^3) + t(\mathcal{C}^3 + \mathcal{A} = \mathcal{C}^1) = \\ &\min_{\substack{\mathcal{C}^i, \mathcal{C}^j, \\ \mathcal{C}^k \in C}} \left(t(2\mathcal{C}^i = \mathcal{C}^j) + t(2\mathcal{C}^j = \mathcal{C}^k) + t(\mathcal{C}^k + \mathcal{A} = \mathcal{C}^i) \right), \end{aligned} \tag{4.3}$$

where $C = \{\mathcal{A}, \mathcal{P}, \mathcal{J}, \mathcal{J}^c, \mathcal{J}^m\}$. From Table 4.1 we see that $(\mathcal{C}^1, \mathcal{C}^2, \mathcal{C}^3) = (\mathcal{J}, \mathcal{J}, \mathcal{J})$ satisfies equation (4.3), and we get a total cost of

$$\begin{aligned} 2t(2\mathcal{J}) + t(\mathcal{J} + \mathcal{A} = \mathcal{J}) &= 2(4M + 4S) + 8M + 3S \\ &= 16M + 11S \\ &= 27M. \end{aligned}$$

This is a $30M$ reduction compared to the version using an exclusively affine representation.

○

We will use the idea from Example 4.1 to optimize the efficiency our method of scalar multiplication. Recall, from Section 3.3, that the NAF_w method (Algorithm 8) was chosen as our method for scalar multiplication. Algorithm 8 uses a NAF_w representation of k. As mentioned in Remark 3.2, this means that k is written as

$$k = 2^{\kappa_0}(2^{\kappa_1}(\cdots 2^{\kappa_{\nu-1}}(2^{\kappa_\nu} W_\nu + W_{\nu-1}) \cdots + W_1) + W_0),$$

where

◇ W_i is odd and $-2^{w-1} + 1 \leq W_i \leq 2^{w-1} - 1$ for all i.

◇ $W_\nu > 0$, $\kappa_0 \geq 0$ and $\kappa_i \geq w$ for all $i \geq 1$.

We assume that the points $[\pm(2i+1)]P$, $1 \leq i \leq 2^{w-2}-1$, have been precomputed. Algorithm 8 works by repeating

$$Q := [2^{\kappa_i}]Q + [W_{i-1}]P,$$

i.e.

$$Q := [2(2^{\kappa_i - 1})]Q + [W_{i-1}]P. \tag{4.4}$$

As $\kappa_i = w+1$ on average according to Proposition 3.7, the cost of the right hand side of assignment (4.4) is

$$w \cdot t(2\mathcal{C}^1) + t(2\mathcal{C}^1 = \mathcal{C}^2) + t(\mathcal{C}^2 + \mathcal{C}^3 = \mathcal{C}^1)$$

for coordinate representations $(\mathcal{C}^1, \mathcal{C}^2, \mathcal{C}^3)$ on average (notice that the result of the addition is expressed in $\mathcal{C}^1$ such that the calculation of $[2^{k_{i+1}} - 1]Q$ can take place in $\mathcal{C}^1$). From Proposition 3.6 we get that the average density of a NAF_w representation is $\frac{1}{w+1}$. If $k = (d_l \cdots d_0)_{\mathrm{NAF}_w}$, we have $1 + \frac{l}{w+1}$ non-zero bits on average. Hence, Algorithm 8 requires

$$T_w(\mathcal{C}^1, \mathcal{C}^2, \mathcal{C}^3) = \frac{lw}{w+1} \cdot t(2\mathcal{C}^1) + \frac{l}{w+1}\big(t(2\mathcal{C}^1 = \mathcal{C}^2) + t(\mathcal{C}^2 + \mathcal{C}^3 = \mathcal{C}^1)\big)$$

on average (excluding the cost for the precomputations). The most frequently occurring value in $T_w(\mathcal{C}^1, \mathcal{C}^2, \mathcal{C}^3)$ is $t(2\mathcal{C}^1)$. From the values in Table 4.1 we see that we should choose either $\mathcal{C}^1 = \mathcal{J}^m$ or $\mathcal{C}^1 = \mathcal{J}$.

The system $\mathcal{C}^3$ is the one used for representing the precomputed points. As addition is the dominating operation involved in the precomputations, one should choose $\mathcal{C}^3$ in a way such that $t(\mathcal{C}^3 + \mathcal{C}^3)$ is as small as possible. From Table 4.1 one sees that both $\mathcal{A}$ and $\mathcal{J}^c$ are good candidates. Which system is the better is determined by considering

1) The ratios I/M and S/M.

2) The possible values of $t(2\mathcal{C}^1 = \mathcal{C}^2) + t(\mathcal{C}^2 + \mathcal{C}^3 = \mathcal{C}^1)$.

As we shall see in Section 5.2.1, it is reasonable to assume that $I/M = 16$ and $S/M = 1$. For $\mathcal{C}^1 = \mathcal{J}$ and $\mathcal{C}^3 = \mathcal{A}$ as well as $\mathcal{C}^3 = \mathcal{J}^c$ we get the lowest cost of the right hand side in equation (4.4) by choosing $\mathcal{C}^2 = \mathcal{J}$. The costs, denoted t_1 and t_2 when using $\mathcal{C}^3 = \mathcal{A}$ and $\mathcal{C}^3 = \mathcal{J}^c$ respectively, are:

$$\begin{aligned} t_1 &= (w+1)t(2\mathcal{J}) + t(\mathcal{J} + \mathcal{A} = \mathcal{J}) \\ &= (4w+12)M + (4w+7)S, \\ t_2 &= (w+1)t(2\mathcal{J}) + t(\mathcal{J} + \mathcal{J}^c = \mathcal{J}) \\ &= (4w+15)M + (4w+7)S. \end{aligned}$$

For $\mathcal{C}^1 = \mathcal{J}^m$ we also get the lowest cost of (4.4), denoted t_3 and t_4 corresponding to $\mathcal{C}^3 = \mathcal{A}$ and $\mathcal{C}^3 = \mathcal{J}^c$ respectively, by choosing $\mathcal{C}^2 = \mathcal{J}$:

$$\begin{aligned} t_3 &= w \cdot t(2\mathcal{J}^m) + t(2\mathcal{J}^m = \mathcal{J}) + t(\mathcal{J} + \mathcal{A} = \mathcal{J}^m) \\ &= (4w + 11)M + (4w + 9)S, \\ t_4 &= w \cdot t(2\mathcal{J}^m) + t(2\mathcal{J}^m = \mathcal{J}) + t(\mathcal{J} + \mathcal{J}^c = \mathcal{J}^m) \\ &= (4w + 14)M + (4w + 9)S. \end{aligned}$$

As both $t_3 > t_1$ and $t_4 > t_2$ (recall that $S = M$), we set $\mathcal{C}^1 := \mathcal{J}$ and proceed with this choice.

4.2.1 Efficient Precomputations

When constructing the table of precomputed points in Algorithm 8, one would normally calculate

$$[2]P, [3]P, [5]P, \ldots, [2^{w-1} - 1]P,$$

which requires one `ECDBL` and $(2^{w-2} - 1)$`ECADD`. Doing this in affine coordinates requires

$$2^{w-2}(I + 2M + S) + S$$

according to Table 4.1.

As this section will show, it is possible to reduce the number of inversions involved in the precomputations by using a method due to Montgomery, known as *simultaneous inversion* in $\mathbb{F}_p$. The method is shown in Algorithm 9.

Algorithm 9 Simultaneous inversion in $\mathbb{F}_p$

Input: $a_1, \ldots, a_j \in \mathbb{F}_p$ with $a_i \neq 0$ for $i = 1, \ldots, j$.
Output: $b_1, \ldots, b_j \in \mathbb{F}_p$ with $a_i b_i = 1$ for $i = 1, \ldots, j$

1: $c_1 \leftarrow a_1$;
2: $i \leftarrow 2$;
3: **while** $i \leq j$ **do**
4: $\quad c_i \leftarrow a_i c_{i-1}$;
5: $\quad i \leftarrow i + 1$;
6: **end while**
7: $u \leftarrow c_j^{-1}$;
8: $i \leftarrow j$;
9: **while** $i \geq 2$ **do**
10: $\quad b_i \leftarrow u c_{i-1}$;
11: $\quad u \leftarrow u a_i$;
12: $\quad i \leftarrow i - 1$;
13: **end while**
14: $b_1 \leftarrow u$;
15: **return** $(b_1, \ldots, b_j)$

Proof of correctness: The loops in lines 3-6 and 9-13 terminate due to the assignments in lines 5 and 12 respectively, so Algorithm 9 terminates. The algorithm maintains the loop invariant

$\mathcal{L}$: *At the beginning of the loop in lines 9-13 of algorithm 9,*
$$u = a_i^{-1} \cdots a_1^{-1}.$$

To see this, notice that the loop in lines 3-6 ensures that $c_i = a_i \cdots a_1$ for $i = 1, \ldots, j$, so $\mathcal{L}$ holds prior to the first iteration in line 9, due to the assignment in line 7.

Assume that $\mathcal{L}$ holds prior to the k'th iteration with $k < j - 2$. After the assignment in line 11, we have $u = a_{i-1}^{-1} \cdots a_1^{-1}$, so, when i is decremented, the invariant is restored.

When the loop in lines 9-13 terminates, we have $i = 1$ and $u = a_1^{-1}$. Therefore, the assignments in lines 10 and 15 ensure that the correct values are returned.

■

Algorithm 9 requires $I + (3j - 3)M$. Cohen [CMO98] shows, that simultaneous inversions can be used to reduce the number of inversions involved in precomputations but does not give a specific algorithm. To the author's knowledge, no such algorithm has been published. Therefore, we construct the algorithm, which is shown in full detail in Algorithm 10. The algorithm makes use of the routines `ECADD_NI` and `ECDBL_NI`. These are elliptic curve addition and doubling respectively in affine coordinates which do not perform any inversions. The inverted values are provided as input to the routines. Source code for Java implementations of Algorithm 10, `ECADD_NI` and `ECDBL_NI` are enclosed in Appendix C.7 and C.2.

Algorithm 10 Precomputations in $\mathcal{A}$ using simultaneous inversion.

Input: $P \in E(\mathbb{F}_p)$ given in $\mathcal{A}$, $w > 1$.
Output: $P, [3]P, \ldots, [2^{w-1}-1]P \in E(\mathbb{F}_p)$.

1: $(x_1, y_1) \leftarrow P$;
2: $(x_2, y_2) \leftarrow \texttt{ECDBL}(P)$;
3: $i \leftarrow 1$;
4: **while** $i \leq w-2$ **do**
5:   **if** $i < w-2$ **then**
6:     $m \leftarrow 2^{i-1}+1$;
7:     $(e_1, \ldots, e_m) \leftarrow (x_{2^i} - x_1, x_{2^i} - x_3, \ldots, x_{2^i} - x_{2^i-1}, 2y_{2^i})$;
8:   **else**
9:     $m \leftarrow 2^{i-1}$;
10:     $(e_1, \ldots, e_m) \leftarrow (x_{2^i} - x_1, x_{2^i} - x_3, \ldots, x_{2^i} - x_{2^i-1})$;
11:   **end if**
12:   $(\delta_{2^i+1}, \delta_{2^i+3}, \ldots, \delta_{2^{i+1}-1}, \delta_{2^{i+1}}) \leftarrow \texttt{SIMINV}(e_1, \ldots, e_m)$; //SIMINV is an implementation of Algorithm 9.
13:   $j \leftarrow 2^i + 1$;
14:   **while** $j \leq 2^{i+1}-1$ **do**
15:     $(x_j, y_j) \leftarrow \texttt{ECADD\_NI}((x_{j-2^i}, y_{j-2^i}), (x_{2^i}, y_{2^i}), \delta_j)$;
16:     $j \leftarrow j+2$;
17:   **end while**
18:   **if** $i < w-2$ **then**
19:     $(x_{2^{i+1}}, y_{2^{i+1}}) \leftarrow \texttt{ECDBL\_NI}((x_{2^i}, y_{2^i}), \delta_{2^{i+1}})$;
20:   **end if**
21:   $i \leftarrow i+1$;
22: **end while**
23: **return** $((x_1, y_1), (x_3, y_3), \ldots, (x_{2^i-1}, y_{2^i-1}))$

Proof of correctness: The incrementations in lines 16 and 21 ensure that the inner loop in lines 14-17 and the outer loop in lines 4-22 both terminate, so the algorithm terminates.

Algorithm 10 maintains the loop invariant

$\mathcal{L}$: *At the beginning of the loop in lines 4-22 of Algorithm 10,*

$$(x_1, y_1), (x_3, y_3), \ldots, (x_{2^i-1}, y_{2^i-1})$$

are the coordinates of $P, [3]P, \ldots, [2^i - 1]P$ *respectively.*

This is true prior to the first iteration due to the assignments in lines 1 and 3. Assume that $\mathcal{L}$ holds prior to the i'th iteration for $1 < i < w-2$. Using the inverted elements from line 12, lines 14-20 calculate

$$((x_{2^i+1}, y_{2^i+1}), (x_{2^i+3}, y_{2^i+3}), \ldots, (x_{2^{i+1}-1}, y_{2^{i+1}-1}), (x_{2^{i+1}}, y_{2^{i+1}})) =$$
$$([2^i+1]P, [2^i+3]P, \ldots, [2^{i+1}-1]P, [2^{i+1}]P).$$

When i is incremented in line 21, the invariant is restored. When $i = w-1$, the algorithm terminates, and we have

$$((x_1, y_1), (x_3, y_3), \ldots, (x_{2^i-1}, y_{2^i-1})) = (P, [3]P, \ldots, [2^{w-1}-1]P).$$

■

The `ECDBL` in line 2 requires $I + 2M + 2S$. Of the $w-2$ iterations of the main loop in lines 4-22, the first $w-3$ iterations each require

- ◇ Simultaneous inversion of $2^{i-1}+1$ elements.
- ◇ 2^{i-1} `ECADD_NI`.
- ◇ One `ECDBL_NI`.

The last iteration requires

- ◇ Simultaneous inversion of 2^{w-3} elements.
- ◇ 2^{w-3} `ECADD_NI`.

The cost of the first $w-3$ iterations is

$$\sum_{i=1}^{w-3} \bigl(I + 3 \cdot 2^{i-1}M + 2^{i-1}(2M+S) + 2M + 2S\bigr) =$$
$$(w-3)I + (5 \cdot 2^{w-3} + 2w - 11)M + (2^{w-3} + 2w - 7)S,$$

while the cost of the last iteration is

$$I + (3 \cdot 2^{w-3} - 3)M + 2^{w-3}(2M+S).$$

The total cost of Algorithm 10 is

$$(w-1)I + (5 \cdot 2^{w-2} + 2w - 12)M + (2^{w-2} + 2w - 5)S. \qquad (4.5)$$

For $w = 4, 5, 6$ (the values of w which we are using) this amounts to

$$\begin{aligned} w = 4 &: \quad 3I + 16M + 7S = 71M \\ w = 5 &: \quad 4I + 38M + 13S = 115M \\ w = 6 &: \quad 5I + 80M + 23S = 183M, \end{aligned}$$

when we assume that $I/M = 16$ and $M = S$.

Other possible schemes for precomputations are:

(a) One doubling in $\mathcal{A}$ and $2^{w-2} - 1$ additions in $\mathcal{A}$.

(b) One doubling from $\mathcal{A}$ to $\mathcal{P}$, one mixed addition $\mathcal{A} + \mathcal{P} = \mathcal{P}$ and $2^{w-2} - 2$ additions in $\mathcal{P}$. To get an affine representation of the precomputed points, one needs an inversion of $2^{w-2} - 1$ elements using simultaneous inversions and $(2^{w-2} - 1) \cdot 2M$.

(c) One doubling from $\mathcal{A}$ to $\mathcal{J}$, one mixed addition $\mathcal{A} + \mathcal{J} = \mathcal{J}$ and $2^{w-2} - 2$ additions in $\mathcal{J}$. To get an affine representation, one needs an inversion of $2^{w-2} - 1$ elements using simultaneous inversions and $(2^{w-2} - 1) \cdot (3M + S)$.

(d) One doubling in $\mathcal{A}$, one addition $\mathcal{A} + \mathcal{A} = \mathcal{P}$ and $2^{w-2} - 2$ mixed additions $\mathcal{A}+\mathcal{P} = \mathcal{P}$. To get an affine representation, one needs an inversion of $2^{w-2}-1$ elements using simultaneous inversions and $(2^{w-2} - 1) \cdot 2M$.

(e) One doubling in $\mathcal{A}$, one addition $\mathcal{A} + \mathcal{A} = \mathcal{J}$ and $2^{w-2} - 2$ mixed additions $\mathcal{A}+\mathcal{J} = \mathcal{J}$. To get an affine representation, one needs an inversion of $2^{w-2}-1$ elements using simultaneous inversions and $(2^{w-2} - 1) \cdot (3M + S)$.

Table 4.2 shows the field operations required by these precomputation schemes and Algorithm 10 for $w = 4, 5, 6$. Table 4.3 shows the total number of field

	Algorithm 10	Scheme (a)	Scheme (b)
$w = 4$	$3I + 16M + 7S$	$4I + 8M + 5S$	$I + 49M + 10S$
$w = 5$	$4I + 38M + 13S$	$8I + 16M + 9S$	$I + 117M + 18S$
$w = 6$	$5I + 80M + 23S$	$16I + 32M + 17S$	$I + 253M + 34S$

	Scheme (c)	Scheme (d)	Scheme (e)
$w = 4$	$I + 49M + 18S$	$2I + 37M + 8S$	$2I + 37M + 13S$
$w = 5$	$I + 121M + 38S$	$2I + 93M + 16S$	$2I + 93M + 29S$
$w = 6$	$I + 265M + 78S$	$2I + 205M + 32S$	$2I + 205M + 61S$

Table 4.2: The tables show the field operations required by different precomputation schemes.

multiplications required by the same precomputation schemes and Algorithm 10 for $w = 4, 5, 6$, assuming that $I/M = 16$ and $S = M$.

As one can see, Algorithm 10 is the most efficient method for doing precomputations. Also, it uses the same amount of storage as the other schemes. Therefore, Algorithm 10 should be used, when precomputations are done in affine coordinates.

	Algorithm 10	Scheme (a)	Scheme (b)	Scheme (c)	Scheme (d)	Scheme (e)
$w = 4$	$71M$	$77M$	$75M$	$83M$	$77M$	$82M$
$w = 5$	$115M$	$153M$	$151M$	$175M$	$141M$	$154M$
$w = 6$	$183M$	$305M$	$303M$	$359M$	$269M$	$299M$

Table 4.3: The table shows the number of field multiplications required by different precomputation schemes.

4.2.2 Initial Doublings during Scalar Multiplication

Regardless of the coordinate representations used, we perform the calculation $[2^{\kappa_\nu} \cdot W_\nu]P$ in Algorithm 8 immediately after doing the precomputations (cf. the description on page 48). Cohen [CMO98] notices and uses that one can reduce the number of elliptic curve operations involved in this calculation, but no general formula is given. We construct such a general formula. The idea is to reduce the number of `ECDBL` by accepting an additional `ECADD`. When $W_\nu = 1$, this is done by noticing that

$$[2^{\kappa_\nu}]P = [2^{\kappa_\nu - w+1}]([2^{w-1} - 1]P + P).$$

This reduces $\kappa_\nu \cdot$ `ECDBL` to $(\kappa_\nu - w - 1) \cdot$ `ECDBL` and one `ECADD`. In general, one has, for W_ν with $1 \leq W_\nu \leq 2^{w-1} - 1$, that $W_\nu = (a_{l-1} \cdots a_0)_2$ with $l \leq w - 1$ due to the definition of the NAF_w. Assuming that $a_{l-1} = 1$, we have

$$[2^{\kappa_\nu} \cdot W_\nu]P = [2^{\kappa_\nu - w+l}]([2^{w-1} - 1]P + [(W_\nu - 2^{l-1}) \cdot 2^{w-l} + 1]P). \qquad (4.6)$$

To see this, notice that

$$\begin{aligned} 2^{\kappa_\nu - w+l}(2^{w-1} + (W_\nu - 2^{l-1}) \cdot 2^{w-l}) &= 2^{\kappa_\nu + l-1} + 2^{\kappa_\nu}(a_{l-2} \cdot 2^{l-2} + \cdots + 1) \\ &= 2^{\kappa_\nu} \cdot (2^{l-1} + a_{l-2} \cdot 2^{l-2} + \cdots 1) \\ &= 2^{\kappa_\nu} \cdot W_\nu. \end{aligned}$$

For $W_\nu \leq 15$ we have

$$\begin{array}{ll} W_\nu = 1: & [2^{\kappa_\nu}]P = [2^{\kappa_\nu - w+1}]([2^{w-1} - 1]P + P) \\ W_\nu = 3: & [2^{\kappa_\nu} \cdot 3]P = [2^{\kappa_\nu - w+2}]([2^{w-1} - 1]P + [2^{w-2} + 1]P) \\ W_\nu = 5: & [2^{\kappa_\nu} \cdot 5]P = [2^{\kappa_\nu - w+3}]([2^{w-1} - 1]P + [2^{w-3} + 1]P) \\ W_\nu = 7: & [2^{\kappa_\nu} \cdot 7]P = [2^{\kappa_\nu - w+3}]([2^{w-1} - 1]P + [3 \cdot 2^{w-3} + 1]P) \quad * \\ W_\nu = 9: & [2^{\kappa_\nu} \cdot 9]P = [2^{\kappa_\nu - w+4}]([2^{w-1} - 1]P + [2^{w-4} + 1]P) \\ W_\nu = 11: & [2^{\kappa_\nu} \cdot 11]P = [2^{\kappa_\nu - w+4}]([2^{w-1} - 1]P + [3 \cdot 2^{w-4} + 1]P) \\ W_\nu = 13: & [2^{\kappa_\nu} \cdot 13]P = [2^{\kappa_\nu - w+4}]([2^{w-1} - 1]P + [5 \cdot 2^{w-4} + 1]P) \\ W_\nu = 15: & [2^{\kappa_\nu} \cdot 15]P = [2^{\kappa_\nu - w+4}]([2^{w-1} - 1]P + [7 \cdot 2^{w-4} + 1]P) \quad * \end{array}$$

The equations marked with $*$ are "critical", in the sense that the addition involved is actually a doubling for $w = 4$ and $w = 5$ respectively. In these cases, an approach using equation (4.6) offers no improvement. For every w such a "critical" case is found for $W_\nu = 2^{w-1} - 1$.

Assuming that the most significant bit is one, there is one positive odd number with binary length one, one with length two and 2^{l-2} with length l for $l \geq 3$. Each of the aforementioned modifications saves $(\kappa_\nu - w + l) \cdot \texttt{ECDBL}$ and introduces an additional ECADD. From Proposition 3.7 we know that $\kappa_\nu = w + 1$ on average. Therefore, one saves

$$\frac{1 \cdot 2 + 1 \cdot 3 + 2 \cdot 4 + 4 \cdot 5 + \cdots + (2^{w-3} - 1)w}{2^{w-2}} \cdot \texttt{ECDBL} =$$
$$\frac{2 - w + \sum_{i=0}^{w-3} 2^i(i+3)}{2^{w-2}} \cdot \texttt{ECDBL} =$$
$$(2^{-w}(-4w+4) + w - 1) \cdot \texttt{ECDBL}$$

on average by using equation (4.6). An extra

$$\frac{\overbrace{1 + 1 + \cdots + 1 + 0}^{2^{w-2}}}{2^{w-2}} \cdot \texttt{ECADD} =$$
$$\frac{2^{w-2} - 1}{2^{w-2}} \cdot \texttt{ECADD} =$$
$$(1 - 2^{2-w}) \cdot \texttt{ECADD}$$

is needed on average.

Remark 4.3. Notice that equation (4.6) also holds when W_i is even. Assuming that the most significant bit is one, there are 2^{l-1} numbers with l bits, so if W_i is any positive number, one saves

$$\frac{\sum_{i=0}^{w-2} 2^i(i+2) - w}{2^{w-1} - 1} \cdot \texttt{ECDBL} =$$
$$\frac{(w-1) \cdot 2^{w-1} - w}{2^{w-1} - 1} \cdot \texttt{ECDBL},$$

and introduces an extra

$$\frac{\overbrace{1 + 1 + \cdots + 1 + 0}^{2^{w-1}-1}}{2^{w-1} - 1} \cdot \texttt{ECADD} =$$
$$\frac{2^{w-1} - 2}{2^{w-1} - 1} \cdot \texttt{ECADD},$$

on average.

4.2.3 Double in $\mathcal{J}$, Precomputed Points in $\mathcal{A}$

We assume that $\mathcal{C}^1 = \mathcal{J}$ and $\mathcal{C}^3 = \mathcal{A}$. We look for $\mathcal{C}^2$ such that

$$t(2\mathcal{J} = \mathcal{C}^2) + t(\mathcal{C}^2 + \mathcal{A} = \mathcal{J})$$

is minimized. From Table 4.1 we see that $\mathcal{J}$ is the better choice. Therefore, we choose $(\mathcal{C}^1, \mathcal{C}^2, \mathcal{C}^3) = (\mathcal{J}, \mathcal{J}, \mathcal{A})$. As precomputations are done in $\mathcal{A}$, we use the technique from Section 4.2.1 to get a cost of

$$PRE_w = (w-1)I + (5 \cdot 2^{w-2} + 2w - 12)M + (2^{w-2} + 2w - 5)S$$

for the precomputations.

When performing the first stage (FS) of doublings, we use the method from Section 4.2.2. If $1 \leq W_\nu < 2^{w-1} - 1$, this requires

$$\begin{aligned} FS^1(s) &= t(\mathcal{A} + \mathcal{A} = \mathcal{J}) + (s+1)t(2\mathcal{J}) \\ &= (4s+8)M + (4s+6)S, \end{aligned}$$

where s is the binary length of W_ν. If $W_\nu = 2^{w-1} - 1$, we get a cost of

$$\begin{aligned} FS^2_w &= t(2\mathcal{A} = \mathcal{J}) + w \cdot t(2\mathcal{J}) \\ &= (4w+2)M + 4(w+1)S \end{aligned}$$

on average. The total cost for the first stage of doublings is

$$\begin{aligned} FS_w &= \frac{FS^1(1) + FS^2_w - FS^1(w-1) + \sum_{i=0}^{w-3} 2^i \cdot FS^1(i+2)}{2^{w-2}} \\ &= (2^{3-w} + 4w)M + (3 \cdot 2^{3-w} + 4w - 2)S \end{aligned}$$

on average.

For the last stage (LS) of doublings $(2^{\kappa_0}(Q + [W_0]P)$, where Q is the intermediate point, we observe that we need $\kappa_0 \cdot t(2\mathcal{J})$. From Proposition 3.7 we know, that the expected value of κ_0 is one. Therefore, the last stage of doublings requires

$$LS = t(2\mathcal{J}) = 4M + 4S$$

on average.

After having taken into account the requirements of the first and last stage of doublings, we need only to be concerned with the subsequence of bits of k marked with † in equation (4.7) below.

$$k = \underbrace{(\overbrace{W_\nu 0 \cdots 0}^{\kappa_\nu \geq w} \underbrace{\overbrace{W_{\nu-1} 0 \cdots 0}^{\kappa_{\nu-1} \geq w} \cdots \overbrace{W_1 0 \cdots 0}^{\kappa_1 \geq w} W_0}_{\dagger} \overbrace{0 \cdots 0}^{\kappa_0 \geq 0})}_{l+1}{}_{NAF_w}. \tag{4.7}$$

Assuming that $\kappa_0 = 1$ and $\kappa_\nu = w + 1$, there are $m := l - w - 1$ bits marked with †. Recall, from Proposition 3.6, that the density of an integer in NAF_w is $\frac{1}{w+1}$ on average. Each of the $\frac{m}{w+1}$ non-zero bits of † corresponds to an addition. Therefore, the average number of field operations is

$$\begin{aligned} T_w(\mathcal{J}, \mathcal{J}, \mathcal{A}) = {} & PRE_w + FS_w + LS + \\ & m \cdot t(2\mathcal{J}) + \frac{m}{w+1} t(\mathcal{A} + \mathcal{J} = \mathcal{J}) + C, \end{aligned}$$

when using Algorithm 8 with $(\mathcal{C}^1, \mathcal{C}^2, \mathcal{C}^3) = (\mathcal{J}, \mathcal{J}, \mathcal{A})$. Here, C denotes the cost of converting the final point from $\mathcal{J}$ to $\mathcal{A}$. This conversion requires $I + 3M + S$, i.e.

$$\begin{aligned} & T_w(\mathcal{J}, \mathcal{J}, \mathcal{A}) = \\ & w \cdot I + \left(5 \cdot 2^{w-2} + 2^{3-w} + \frac{8m}{w+1} + 4l + 2w - 13\right) M + \\ & \left(2^{w-2} + 3 \cdot 2^{3-w} + \frac{3m}{w+1} + 4l + 2w - 10\right) S. \end{aligned} \tag{4.8}$$

4.2.4 Double in $\mathcal{J}$, Precomputed Points in $\mathcal{J}^c$

We assume that $\mathcal{C}^1 = \mathcal{J}$ and $\mathcal{C}^3 = \mathcal{J}^c$. We look for $\mathcal{C}^2$ such that

$$t(2\mathcal{J} = \mathcal{C}^2) + t(\mathcal{C}^2 + \mathcal{J}^c = \mathcal{J})$$

is as small as possible. From Table 4.1 we see that $\mathcal{J}$ is the better choice, so we choose $(\mathcal{C}^1, \mathcal{C}^2, \mathcal{C}^3) = (\mathcal{J}, \mathcal{J}, \mathcal{J}^c)$.

When using non-affine (inversion-free) coordinates for the precomputed points, there is nothing to gain from the modification discussed in Section 4.2.1. The total requirement for the precomputations is

$$\begin{aligned} PRE_w &= t(2\mathcal{A} = \mathcal{J}^c) + t(\mathcal{J}^c + \mathcal{A} = \mathcal{J}^c) + (2^{w-2} - 2) t(\mathcal{J}^c + \mathcal{J}^c) \\ &= (11 \cdot 2^{w-2} - 10)M + 3 \cdot 2^{w-2} S. \end{aligned}$$

For the first stage of doublings we, once again, use the approach from Section 4.2.2. If $W_\nu = 1$, we take advantage of P being in affine coordinates to get

$$\begin{aligned} FS^1 &= t(\mathcal{A} + \mathcal{J}^c = \mathcal{J}) + 2 \cdot t(2\mathcal{J}) \\ &= 15M + 10S. \end{aligned}$$

If $1 < W_\nu < 2^{w-1} - 1$, we get

$$\begin{aligned} FS^2(s) &= t(\mathcal{J}^c + \mathcal{J}^c = \mathcal{J}) + (s+1) t(2\mathcal{J}) \\ &= (4s + 14)M + (4s + 6)S, \end{aligned}$$

where s is the binary length of W_ν. In the case of $W_\nu = 2^{w-1}-1$, the requirement is

$$\begin{aligned} FS_w^3 &= t(2\mathcal{J}^c = \mathcal{J}) + w \cdot t(2\mathcal{J}) \\ &= 4(w+1)M + (4w+3)S \end{aligned}$$

on average. This makes an average cost of

$$\begin{aligned} FS_w &= \frac{FS^1 + FS_w^3 - FS^2(w-1) + \sum_{i=0}^{w-3} 2^i \cdot FS^2(i+2)}{2^{w-2}} \\ &= (-5 \cdot 2^{2-w} + 4w + 6)M + (5 \cdot 2^{2-w} + 4w - 2)S \end{aligned}$$

for the first stage of doublings.

With the same reasoning as in Section 4.2.3, the last stage of doublings requires

$$LS = t(2\mathcal{J}) = 4M + 4S$$

on average.

Even though we have chosen $\mathcal{J}^c$ for the precomputations, the point P (and $-P$) are still available in affine representation as input to the algorithm. As mixed addition with affine points is faster than mixed addition with points in $\mathcal{J}^c$, we use the affine representation of P when $W_i = \pm 1$. The probability of the event $W_i = \pm 1$ to occur is $\frac{1}{2^{w-2}}$. We define the map ψ by

$$\psi(w) = \frac{1}{2^{w-2}} \cdot t(\mathcal{A} + \mathcal{J} = \mathcal{J}) + \left(1 - \frac{1}{2^{w-2}}\right) \cdot t(\mathcal{J}^c + \mathcal{J} = \mathcal{J}).$$

With $m = l - w - 1$, the total cost is

$$T_w(\mathcal{J}, \mathcal{J}, \mathcal{J}^c) = PRE_w + FS_w + m \cdot t(2\mathcal{J}) + \frac{m \cdot \psi(w)}{w+1} + C,$$

where, once again, $C = I + 3M + S$. Thus,

$$\begin{aligned} &T_w(\mathcal{J}, \mathcal{J}, \mathcal{J}^c) = \\ &I + \left(11 \cdot 2^{w-2} - 5 \cdot 2^{2-w} + \frac{(11 - 3 \cdot 2^{2-w})m}{w+1} + 4l - 5\right) M + \\ &\left(3 \cdot 2^{w-2} + 5 \cdot 2^{2-w} + \frac{3m}{w+1} + 4l - 5\right) S. \end{aligned} \tag{4.9}$$

4.3 Comparison and Conclusion

As mentioned in Section 4.2, the choice between $\mathcal{C}^3 = \mathcal{A}$ and $\mathcal{C}^3 = \mathcal{J}^c$ depends on the ratio I/M, which in our case is assumed to be 16. In this section we analyze the situation for a selection of values of l and determine when to use the different representations.

The interesting cases are $l = 192$, 224, 256, 384, 521 (cf. Section 2.2). Therefore, the values of $T_w(\mathcal{J}, \mathcal{J}, \mathcal{A})$ and $T_w(\mathcal{J}, \mathcal{J}, \mathcal{J}^c)$ are determined for these values of l. When performing these calculations, w should be chosen optimally. Figure 4.1 shows $T_w(\mathcal{J}, \mathcal{J}, \mathcal{J}^c)$ for $w \in [1, 10]$. One might suspect that $w = 5$ and $w = 6$

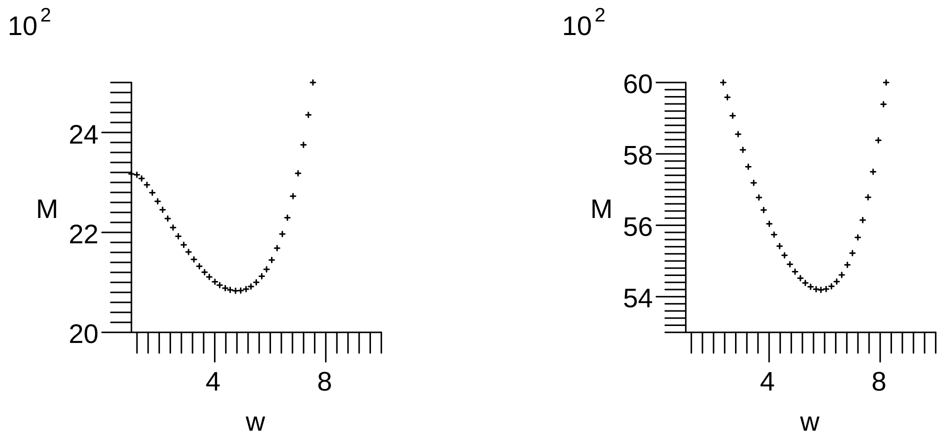


Figure 4.1: The plots show the number of field multiplications in $T_w(\mathcal{J}, \mathcal{J}, \mathcal{J}^c)$ for $l = 192$ (left) and $l = 521$ (right) respectively, when $w \in [1, 10]$.

are optimal values in the two cases, and, indeed, one finds that for $l = 192$:

$$T_4(\mathcal{J}, \mathcal{J}, \mathcal{J}^c) = 2120M,$$
$$T_5(\mathcal{J}, \mathcal{J}, \mathcal{J}^c) = 2084M,$$
$$T_6(\mathcal{J}, \mathcal{J}, \mathcal{J}^c) = 2139M.$$

Similarly, for $l = 521$:

$$T_5(\mathcal{J}, \mathcal{J}, \mathcal{J}^c) = 5463M,$$
$$T_6(\mathcal{J}, \mathcal{J}, \mathcal{J}^c) = 5420M,$$
$$T_7(\mathcal{J}, \mathcal{J}, \mathcal{J}^c) = 5522M.$$

Let $T_w^l(\mathcal{C}^1, \mathcal{C}^2, \mathcal{C}^3)$ denote the value of $T_w(\mathcal{C}^1, \mathcal{C}^2, \mathcal{C}^3)$ for a fixed value of l. Determining when to use $\mathcal{A}$ instead of $\mathcal{J}^c$ for the precomputed points boils down to solving the inequalities

$$\min_w(T^l_w(\mathcal{J},\mathcal{J},\mathcal{A})) < \min_w(T^l_w(\mathcal{J},\mathcal{J},\mathcal{J}^c)), \quad l \in \{192, 224, 256, 384, 521\}$$

with respect to I/M.

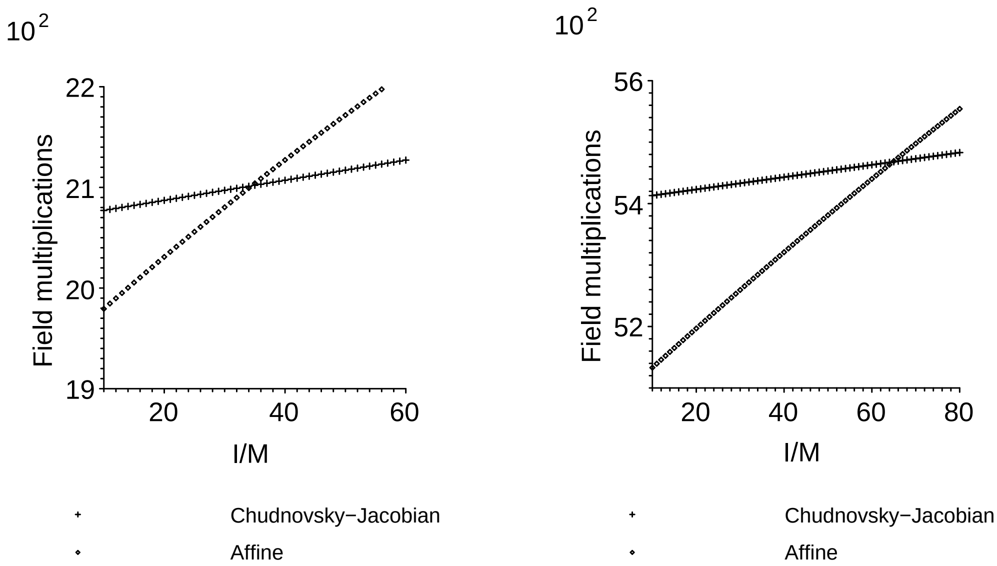


Figure 4.2: The plots show the value of $\min_w(T^l_w(\mathcal{J},\mathcal{J},\mathcal{A}))$ and $\min_w(T^l_w(\mathcal{J},\mathcal{J},\mathcal{J}^c))$ for $l = 192$ (left) and $l = 521$ (right)

	$l = 192$	$l = 224$	$l = 256$	$l = 384$	$l = 521$
$I/M \in$	$[0, 34]$	$[0, 37]$	$[0, 41]$	$[0, 53]$	$[0, 64]$

Table 4.4: The values of I/M for which precomputations should be done in $\mathcal{A}$.

Figure 4.2 shows the values of $\min_w(T^l_w(\mathcal{J},\mathcal{J},\mathcal{A}))$ and $\min_w(T^l_w(\mathcal{J},\mathcal{J},\mathcal{J}^c))$ for $l = 192$ and $l = 521$. Table 4.4 shows the values of I/M for which the optimal choice is $\mathcal{C}^3 = \mathcal{A}$. As we are working with $I/M = 16$, we should choose $(\mathcal{C}^1, \mathcal{C}^2, \mathcal{C}^3) = (\mathcal{J},\mathcal{J},\mathcal{A})$.

We now draw conclusions based on our observations. In Sections 4.1.1-4.1.3, formulas for the operations `ECDBL` and `ECADD` using the coordinate representations $\mathcal{P}, \mathcal{A}, \mathcal{J}, \mathcal{J}^c$ and $\mathcal{J}^m$ where presented. In Section 4.2 we showed that the total number of field multiplications involved in scalar multiplication on an elliptic curve can be reduced by using a mixture of the coordinate representations. Jacobian coordinates were chosen for performing sequences of doublings during scalar multiplication. In Sections 4.2.3 and 4.2.4 we analyzed the cases where precomputations are done in affine coordinates and Chudnovsky-Jacobian coordinates respectively. We compared the two choices of representations for the precomputations. The conclusion was that using affine coordinates is the more efficient choice, when $S = M$ and $I/M = 16$. Therefore, we should represent the precomputed points in affine coordinates, perform doublings in Jacobian coordinates and perform additions in mixed affine/Jacobian coordinates. We also notice that the optimizations from Sections 4.2.1 and 4.2.2 should be used, as they reduce the average number of required field operations.

Chapter 5

Implementations

In this chapter we analyze the test implementation developed by IBM and compare it to our implementation of the scheme constructed in Chapters 3 and 4. The goal is to determine how much is saved, measured in field multiplications and execution time, by using our implementation in preference to the IBM test implementation.

5.1 Setup for Time Measurements

All implementations are executed on a Lenovo ThinkPad T60P with Intel Core Duo 2.16 GHz processor and 1GB DDRII SDRAM using Sun Java SDK version 1.5.0. Test vectors used for the timings are enclosed in Appendix B.

When measuring the execution time of an implementation `IMPL`, the straightforward way is to execute a program similar to the one shown in pseudo code below, where we assume that `System.time` returns the current time in milliseconds:

```
t := System.time();
IMPL();
t := System.time()-t;
return t;
```

However, some implementations require so little CPU time per execution that this strategy results in the value $t = 0$. Instead, we execute the implementation `IMPL` as many times as possible within a fixed time period. Subsequently, we determine the average execution time for the implementation. We select a time period of two seconds and get the strategy shown in pseudo code below:

```
n := 0;
limit := 2000 + (start := System.time());
while (end := System.time()) < limit do
  IMPL();
  n := n+1;
end while
t := (end-start)/n;
return t;
```

This ensures that we always get a non-zero result when measuring the implementations.

When performing tests on a Java Virtual Machine (JVM), one must keep in mind that the JVM makes use of a Just-in-time (JIT) compiler[1] to convert parts of the Java bytecode, which are identified to be frequently occurring, into native (assembler) code in order to improve execution speed. If no native code is produced, the implementation will be executed using the bytecode-interpreter. This does not give a clear picture of the time required to perform modular arithmetic on large integers, as we cannot assume that the bytecode-interpreter is optimized for these operations. In order to ensure that native code is produced, one must execute the time measuring routine a number of times successively, as this forces the JIT compiler into producing native code. We find that two successive executions of the routine is sufficient.

5.2 IBM Test Implementation

The original source code from IBM is enclosed in Appendix C.3.1. The implementation is one of Algorithm 5, which is based on the recommendations in [P1300]. It contains no separate field implementation and performs integer recoding during scalar multiplication. In order to attain more clarity and better grounds of comparison, the original IBM version is modified slightly. The modifications encompass

(i) Creating a separate field implementation.

(ii) Performing integer recoding prior to scalar multiplication. This results in Algorithm 6.

Source code for the modified implementation is enclosed in Appendix C.3.2. Source code for implementations of integer recoding routines are enclosed in Appendix C.7.

[1] In our case the JIT compiler is part of the the Sun Hotspot JVM.

5.2.1 Field Implementations

We construct an implementation of each of the fields $\mathbb{F}_{p192}$, $\mathbb{F}_{p224}$, $\mathbb{F}_{p256}$, $\mathbb{F}_{p384}$ and $\mathbb{F}_{p521}$. Source code for the field implementations is enclosed in Appendix C.1. The implementations are based on Java's `BigInteger` class, which is capable of performing modular arithmetic on large integers. Only modular addition and subtraction are implemented differently in order to reduce the execution time for these operations. Timings of a selection field operations are shown in Table 5.1. With these implementations one can reasonably assume that $S = M$

	$\mathbb{F}_{p192}$	$\mathbb{F}_{p224}$	$\mathbb{F}_{p256}$	$\mathbb{F}_{p382}$	$\mathbb{F}_{p521}$
Operation	Time	Time	Time	Time	Time
Addition	$266\,ns$	$276\,ns$	$291\,ns$	$338\,ns$	$400\,ns$
Subtraction	$248\,ns$	$439\,ns$	$273\,ns$	$323\,ns$	$686\,ns$
Multiplication (M)	$3184\,ns$	$3856\,ns$	$4746\,ns$	$8\,\mu s$	$16\,\mu s$
Squaring (S)	$3318\,ns$	$4149\,ns$	$4950\,ns$	$8\,\mu s$	$15\,\mu s$
Inversion (I)	$51\,ms$	$59\,ms$	$80\,ms$	$136\,ms$	$223\,ms$

Table 5.1: The table shows timings of a selection of field operations.

and $I/M = 16$. We have a multiplication-to-addition ratio of approximately 21 on average, and will assume that the time required to perform an addition or subtraction is negligible compared to the time required to perform a multiplication. When comparing our field implementations to field implementations such as the one described in [BHLM01], which has a multiplication-to-addition ratio of approximately 15 on average and in which additions and subtractions are assumed to be negligible, it seems valid to make this assumption. Furthermore, optimization of field operations is not a subject of this examination. Therefore, no further steps will be taken to reduce the execution times of modular addition and subtraction. However, as addition and subtraction does require *some* execution time, we must be prepared that our assumption will result in discrepancies between theoretical estimates and experimental values later on.

The time required to perform negation and comparison in the fields is even less than that required to perform addition and subtraction. Negations and comparisons are, therefore, also assumed to be negligible.

5.2.2 Scalar Multiplication

The method used for scalar multiplication in the IBM test implementation is the addition-subtraction method (Algorithm 6) using exclusively affine coordinates. From Section 3.2.1 we know that this scheme requires

$$l \cdot t(2\mathcal{A}) + \frac{l}{3} \cdot t(\mathcal{A} + \mathcal{A}),$$

where l is the number of bits in the scalar. Using the values from Table 4.1, we see that the average requirement is

$$\begin{aligned} T_{IBM} &:= l \cdot t(2\mathcal{A}) + \frac{l}{3} \cdot t(\mathcal{A} + \mathcal{A}) \\ &= \frac{4l}{3} I + \frac{8l}{3} M + \frac{7l}{3} S \\ &= \frac{79l}{3} M. \end{aligned}$$

The average number of field multiplications and timings of the implementation are shown in Table 5.2.

	$l = 192$	$l = 224$	$l = 256$	$l = 382$	$l = 521$
T_{IBM}	$5056M$	$5899M$	$6741M$	$10112M$	$13720M$
Time	$15625\,\mu s$	$22222\,\mu s$	$30769\,\mu s$	$80\,ms$	$175\,ms$

Table 5.2: The table shows the average number of field multiplications required by the scheme implemented by IBM and timings of the implementation.

5.3 An Efficient Scheme

Using the same field implementations as the ones described in Section 5.2.1, we implement Algorithm 8 with $(\mathcal{C}^1, \mathcal{C}^2, \mathcal{C}^3) = (\mathcal{J}, \mathcal{J}, \mathcal{A})$ (in the notation of Section 4.2). Also, we use the modifications described in Sections 4.2.1 and 4.2.2. Source code for the implementation is enclosed in Appendix C.3.3.

With $T_{Optimized} := \min_w(T_w(\mathcal{J}, \mathcal{J}, \mathcal{A}))$ (the average number of field multiplications required by our efficient scheme in the notation of Section 4.2.3), we get the values in Table 5.3. The table also shows the average reduction compared to the IBM test implementation. As Table 5.3 shows, we get an approximate reduction of 61% on average. Timings of the implementation of our scheme are shown in Table 5.4. The reduction in execution time compared to the IBM implementation is approximately 55% on average. The main reason for the discrepancy

	$l = 192$	$l = 224$	$l = 256$	$l = 382$	$l = 521$
$T_{Optimized}$	2011M	2326M	2640M	3866M	5177M
Reduction	60.0%	60.6%	60.8%	61.8%	62.3%.

Table 5.3: The table shows the average number of field multiplications required by our efficient scheme and reduction compared to the scheme implemented by IBM.

	$l = 192$	$l = 224$	$l = 256$	$l = 382$	$l = 521$
Time	$7352\,\mu s$	$10050\,\mu s$	$13698\,\mu s$	$34482\,\mu s$	$81\,ms$

Table 5.4: The table shows timings of the optimized implementation.

between the theoretical and the empirically observed reduction is that we do not take into account the number of modular additions/subtraction required by the scalar multiplication schemes (cf. our discussion in Section 5.2.1). However, the timings support our conclusions in that our scheme remains advantageous in the experiments, and we conclude that the discrepancy is acceptable.

The relative improvement gained by using our implementation in preference to the IBM test implementation could potentially be even greater. The NIST primes allow for very fast modular reduction compared to the speed of modular inversion, as shown by Solinas in [Sol99]. This makes it even more beneficial to move from a system based on affine coordinates to a system using coordinates which allows for elliptic curve operations without inversions.

5.4 Conclusion

The implementation developed by IBM is an addition-subtraction method based on the standards in [P1300], developed for test purposes. It uses exclusively affine coordinates. The IBM test implementation contains no separate field implementation, and integer recoding is performed during scalar multiplication. Therefore, the implementation is modified slightly, in order to be able to compare the implementation with one based on the construction in Chapters 3 and 4.

Our field implementations are based on Java's `BigInteger` class with a custom implementation of modular addition and subtraction. Our implementations can all be assumed to have $S = M$ and $I/M = 16$. Also, we assume negligible costs for addition and subtraction. The field implementations are used in the modified IBM implementation as well as in our implementation of an efficient scalar multiplication scheme.

Our scalar multiplication scheme is a NAF_w method with precomputations in affine coordinates, doublings in Jacobian coordinates and addition in mixed affine/Jacobian coordinates. We also implement the modifications discussed in Sections 4.2.1 and 4.2.2. This gives a 61% reduction in the average number of required field multiplications, while timings show a 55% reduction on average. We claim that the reason for the discrepancy between the theoretical reduction and the empirically observed reduction is that our theoretical examination does not take into account the number of modular additions and subtractions performed. The experiments do support our conclusions, and we disregard the discrepancy.

Part III

Countermeasures against Power Analysis

Chapter 6

Power Analysis

One of the major threats against ECC-systems is the use of *side channel analysis* to break the systems, i.e. gain knowledge of sensitive information (most commonly the secret key of the system). A *side channel* is a source of information about the system which is available to anyone having access to measurements of the hardware executing the algorithms of the system, e.g. timing information or power consumption measurements. A *side channel attack* is an attack based on side channel analysis (more details can be found in [Joy05]).

Side channel attacks can be *invasive* or *non-invasive*. Invasive attacks partially or fully destroys the chip executing the system; therefore, they are likely to be detected. Furthermore, these kinds of attacks require use of laboratory stations and are time-consuming. Countermeasures against invasive attacks are usually implemented in hardware.

Non-invasive side channel attacks leave the physical system (chip, patching etc.) undamaged; therefore, they are difficult to detect. Performing non-invasive attacks is also relatively inexpensive compared to performing invasive attacks. Countermeasures against non-invasive attacks are usually implemented in software and are based on a mathematical foundation. In this chapter we will focus on non-invasive side channel attacks and the mathematical countermeasures against them.

So far, no comparison between the efficiencies of these countermeasures has been published. We perform such a comparison in the following sections.

We assume that the hardware executing the system is located on a smart card or a similar, easily accessible, device (see [ACD$^+$05] for a detailed introduction to smart cards). The operation performed by the hardware is $[k]P$, where k is the secret key. The purpose of an attack is to learn the value of k. Notice that the attack only applies to protocols using long-term keys, e.g. the ElGamal cryptosystem (cf. Section 2.1.1). For protocols using ephemeral keys, e.g. the ECDSA, the attacks described in this chapter are not useful.

The most commonly known side channel attacks are *timing attacks*, attacks based on *simple power analysis* (SPA) and attacks based on *differential power* *analysis* (DPA). By implementing countermeasures against SPA and DPA one also thwarts timings attacks. Therefore, we will not consider countermeasures against timing attacks and only focus on SPA attacks and DPA attacks. Successful attacks based on power analysis have been documented (for instance in [KJJ99] and [AO00]). Both types of attacks use information about power consumption as a side channel. What makes SPA and DPA possible is that the power consumption in the hardware executing an ECC-system depends on the data being manipulated in the system. Under some circumstances, measurements of the power consumption reveal information about the secret key k.

Almost all chip design today is based on CMOS (Complementary Metal Oxide Semiconductor) technology. A change of state in a CMOS logical gate results in a change in power consumption. This change can be detected by using an oscilloscope. Any electronic device (PC, smart card etc.) performs calculations by switching a number of logical gates (in the CPU, buses, memory etc.). The total number of gates used in a computation depends on the values (the data involved in the computation) in the registers of the device. As power consumption depends on the number of gates switching, different data inputs for the same operation will result in different *power consumption traces* (measurements of power consumption during the time of execution). By monitoring (and possibly performing a statistical evaluation of) the traces, an attacker can sometimes attain knowledge about sensitive information in the system. In our case, the sensitive information is the value of the integer k being used in scalar multiplication.

6.1 Simple Power Analysis

SPA is based on a single power consumption trace from the chip. As Chapter 4 shows, the number and composition of field operations involved in an `ECADD` differs from the number and composition of field operations involved in an `ECDBL`. Each type of field operation has its own unique power consumption trace. Therefore, an `ECADD` and an `ECDBL` have different power consumption traces in general.

If the double-and-add algorithm (Algorithm 1) is used for scalar multiplication, an attacker will see a power consumption trace consisting of a mixture of two distinguishable sub-traces corresponding to `ECDBL` and `ECADD` respectively (see Figure 6.1).

As doublings occur more frequently than addition on average, the attacker can identify the most frequently occurring sub-trace as an `ECDBL` and the other sub-trace as an `ECADD`. Knowing that an `ECDBL` corresponds to a zero-bit in the scalar k and that an `ECDBL` followed by an `ECADD` corresponds to a one-bit in the scalar, an attacker will be able to deduce all the bits of k by observing the power consumption trace from a single execution of the algorithm. In Figure 6.1 the observed sequence of bits is 001.

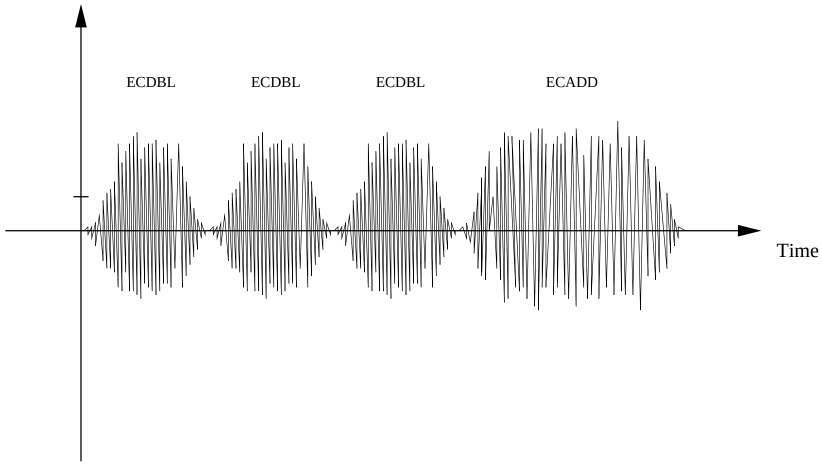


Figure 6.1: Schematic SPA trace for the double-and-add algorithm

A straightforward way of securing the algorithm is to *always* perform an `ECADD` and an `ECDBL`, regardless of the value of the current bit. Subsequently, a superfluous `ECADD` is disregarded. This approach is known as the *double-and-add always* method. The method is shown in Algorithm 11.

Algorithm 11 Double-and-add-always

Input: $P \in E(\mathbb{F}_p)$ and $k = (k_{l-1} \cdots k_0)_2$
Output: $[k]P \in E(\mathbb{F}_p)$.

$Q_0 \leftarrow P$; $Q_1 \leftarrow \mathcal{O}$; $i \leftarrow l - 2$;
while $i \geq 0$ **do**
  $Q_0 \leftarrow [2]Q_0$;
  $Q_{1-k_i} \leftarrow Q_{1-k_i} \oplus P$;
  $i \leftarrow i - 1$;
end while
return Q_0

Notice that Algorithm 11 performs one `ECDBL` and one `ECADD` for each bit of k. A power consumption trace from the execution of the algorithm will, therefore, be useless in an SPA attack. Algorithm 11 requires $(l-1)(\texttt{ECDBL} + \texttt{ECADD})$.

In general, a scalar multiplication algorithm is vulnerable to SPA if it behaves differently according to the values of the individual bits of the scalar. On the other hand, it is impossible to mount a successful SPA attack if the algorithm behaves exactly the same regardless of the values of the bits of k. Because of this, all countermeasures against SPA attacks modify the algorithm to get a uniform power consumption trace. The countermeasures can be split into three categories:

1) Algorithms with uniform behaviour.

2) Algorithms with unified addition and doubling.

3) Algorithms with dummy field operations.

When evaluating countermeasures against SPA attacks, one should consider security against *fault injection attacks* (FI attacks). These attacks are based on the idea that one can deduce information about k by forcing the system to perform erroneous instructions during scalar multiplication. The first published FI attack was the "Bellcore attack" on an RSA implementation (see [BDL97]). In ECC-systems, FI attacks can be used to disclose dummy operations in SPA countermeasures. They are carried out by injecting power into, or emissioning light onto, the chip executing the scalar multiplication. This will perturb the components of the chip and alter the value of one or more bits in the representation of the point being multiplied. When the scalar multiplication algorithm terminates, one can compare the result with the correct value of $[k]P$. If the result of the scalar multiplication is correct, regardless of the fault injection, one can deduce that the operation being performed at the time of the injection was a dummy. Consider Algorithm 11 as an example. If $k_i = 0$, the operation in line 4 is a dummy. If the calculation $Q_{1-k_i} \leftarrow Q_{1-k_i} \oplus P$ is perturbed, it will not influence the return value of the algorithm.

A successful FI attack requires the ability to execute the algorithm a number of times with a fixed k as well as access to a correct result of the calculation $[k]P$ for comparison. Therefore, if the value of k is changed every time the algorithm is executed, FI attacks are not possible. Algorithms without dummy operations are also secure against FI attacks. Conversely, algorithms which use dummy operations are a priori vulnerable to FI attacks, unless the value of k is randomized in some way.

6.1.1 Algorithms with Uniform Behaviour

The simplest example of a scalar multiplication algorithm with uniform behaviour is the double-and-add always method (Algorithm 11). With an optimal choice of coordinate representations, the algorithm requires

$$\begin{aligned} T_{\text{Alg.11}} &= (l-1)(t(2\mathcal{J}) + t(\mathcal{J} + \mathcal{A} = \mathcal{J})) + C \\ &= I + (12l - 9)M + (7l - 6)S. \end{aligned}$$

Here, $C = I + 3M + S$ is the cost of conversion from $\mathcal{J}$ to $\mathcal{A}$. Table 6.1 shows the number of field multiplications performed by Algorithm 11 and the overhead compared to the efficient, non-secure scheme described in Section 5.3.
Algorithm 11 uses no extra storage for precomputation. As it introduces dummy operations, it is, however, vulnerable to FI attacks. One cannot hope for improvements by adapting Algorithm 11 to a scalar in NAF_w (cf. Section 3.2.2). Indeed, the whole point of the NAF_w method is to reduce the number of `ECADD`

	$l = 192$	$l = 224$	$l = 256$	$l = 384$	$l = 521$
$T_{\text{Alg.11}}$	$3649M$	$4257M$	$4865M$	$7297M$	$9900M$
Overhead	81.5%	83.0%	84.2%	88.7%	91.2%

Table 6.1: The table shows the number of field multiplications required by Algorithm 11 and overhead compared to the efficient, non-secure scheme.

involved in scalar multiplication by lowering the Hamming weight of the scalar. As Algorithm 11 executes both an `ECDBL` and an `ECADD` for every bit of the scalar, the NAF_w method does not apply in any sensible way.

In [OT03] Okeya and Takagi show that it is possible to construct a more efficient scheme, which uses a different representation of the scalar. The representation is constructed using Algorithm 12. The algorithm returns the non-zero bits in a representation on the form

$$k = (U_d \overbrace{0 \cdots 0}^{w} U_{d-1} \overbrace{0 \cdots 0}^{w} \cdots U_0 \overbrace{0 \cdots 0}^{w})_2, \tag{6.1}$$

written $(U_d \cdots U_0)_{\text{NAF}_w^*}$. This representation satisfies Definition 3.3 except for the fact that the U_i's are allowed to be even.

Algorithm 12 Okeya & Takagi recoding

Input: $k = (k_{l-1} \cdots k_0)_2, w > 1$.
Output: $U_{\lceil \frac{l}{w} \rceil}, \ldots, U_0$ such that $k = (U_{\lceil \frac{l}{w} \rceil} \cdots U_0)_{NAF_w^*}$.

1: $d \leftarrow \lceil \frac{l}{w} \rceil$;
2: $i \leftarrow 0$;
3: **while** $i \leq d$ **do**
4: $\quad U_i \leftarrow k \text{ mods } 2^w$;
5: $\quad k \leftarrow k - U_i$;
6: $\quad k \leftarrow \frac{k}{2^w}$;
7: $\quad i \leftarrow i + 1$;
8: **end while**
9: **return** $U_d, \ldots, U_0$

If k is even in line 4 of Algorithm 12, we ensure that mods is well-defined by always choosing the positive residue in situations where the absolute values of the positive and negative residue are equal.

The advantage of the NAF_w^* representation in equation (6.1) is that it consists of repetitions of a single, fixed pattern. This is used in Algorithm 13, which thwarts SPA.

Algorithm 13 W-double-one-add always

Input: $P \in E(\mathbb{F}_p)$, $w > 1$ and $k = (U_d \cdots U_0)_{NAF_w^*}$.
Output: $[k]P \in E(\mathbb{F}_p)$.

1: Compute $[\pm 2]P$, $[\pm 3]P$, $[\pm 4]P, \ldots, [\pm(2^{w-1}-1)]P$.
2: $Q_0 \leftarrow [U_d]P$;
3: $i \leftarrow d-1$;
4: **while** $i \geq 0$ **do**
5:     $j \leftarrow w$;
6:     **while** $j \geq 1$ **do**
7:         $Q_0 \leftarrow [2]Q_0$;
8:         $j \leftarrow j-1$;
9:     **end while**
10:     $Q_1 \leftarrow Q_0 \oplus [U_i]P$;
11:     $Q_0 \leftarrow Q_{\delta(U_i)}$;
12:     $i \leftarrow i-1$;
13: **end while**
14: **return** Q_0

If $U_i = 0$, the addition in line 10 is a dummy operation, as the result of the addition is never used. In this case, any point can be used for $[U_i]P$ (if addition with the point at infinity $\mathcal{O}$ is faster than addition with other points, one must use a point different from $\mathcal{O}$ for $[0]P$). Algorithm 13 has the fixed pattern

$$\overbrace{\texttt{ECDBL}, \ldots, \texttt{ECDBL}}^{w}, \texttt{ECADD}, \overbrace{\texttt{ECDBL}, \ldots, \texttt{ECDBL}}^{w}, \texttt{ECADD}, \cdots$$

This makes it impossible to deduce any information about k by using SPA. For proofs of correctness of Algorithms 12 and 13 see [OT03].

In line 1, Algorithm 13 precomputes $2^{w-1} - 2$ points. If the precomputed points are represented in $\mathcal{A}$, we can extend the use of simultaneous inversions discussed in Section 4.2.1. Algorithm 14 computes $[2]P, [3]P, [4]P, \ldots, [2^{w-1}-1]P$ using Algorithm 9 for inversions. As was the case with Algorithm 10, Algorithm 14 is based on the idea of Cohen ([CMO98]) but has been constructed for this examination. It has, to the author's knowledge, not been published previously.

Algorithm 14 Precomputations in $\mathcal{A}$ using simultaneous inversion.

Input: $P \in E(\mathbb{F}_p)$ given in $\mathcal{A}$, $w > 1$.
Output: P, $[2]P$, $[3]P, \ldots, [2^{w-1}-1]P \in E(\mathbb{F}_p)$.

$(x_1, y_1) \leftarrow P$;
$(x_2, y_2) \leftarrow$ `ECDBL`(P);
$i \leftarrow 1$;
while $i \leq w-2$ **do**
  $(d_1, \ldots, d_{2^i}) \leftarrow (x_{2^i} - x_1, x_{2^i} - x_2, \ldots, x_{2^i} - x_{2^i-1}, 2y_{2^i})$;
  $(\delta_{2^i+1}, \delta_{2^i+2}, \ldots, \delta_{2^{i+1}-1}, \delta_{2^{i+1}}) \leftarrow$ `SIMINV`$(d_1, \ldots, d_{2^i})$; //`SIMINV` is an implementation of Algorithm 9.
  $j \leftarrow 2^i + 1$;
  while $j \leq 2^{i+1} - 1$ **do**
    $(x_j, y_j) \leftarrow$ `ECADD_NI`$((x_{j-2^i}, y_{j-2^i}), (x_{2^i}, y_{2^i}), \delta_j)$;
    $j \leftarrow j + 1$;
  end while
  $(x_{2^{i+1}}, y_{2^{i+1}}) \leftarrow$ `ECDBL_NI`$((x_{2^i}, y_{2^i}), \delta_{2^{i+1}})$;
  $i \leftarrow i + 1$;
end while
return $((x_1, y_1), (x_2, y_2), \ldots, (x_{2^i-1}, y_{2^i-1}))$

The `ECDBL` requires $I + 2M + 2S$. The $w-2$ iterations of the main loop each requires

- One simultaneous inversion of 2^i elements.
- $(2^i - 1)$`ECADD_NI`.
- One `ECDBL_NI`.

This makes a total requirement of

$$PRE_w^{\mathcal{A}} := I + 2M + 2S + \sum_{i=1}^{w-2}(I + 2^i(5M + S) - 3M + S) =$$
$$(w-1)I + (5 \cdot 2^{w-1} - 3w - 2)M + (2^{w-1} + w - 2)S$$

for the precomputations. If $\mathcal{J}^c$ are used for the precomputations, the cost is

$$\begin{aligned} PRE_w^{\mathcal{J}^c} &= t(2\mathcal{A} = \mathcal{J}^c) + t(\mathcal{A} + \mathcal{J}^c = \mathcal{J}^c) + (2^{w-1} - 4) \cdot t(\mathcal{J}^c + \mathcal{J}^c) \\ &= (11 \cdot 2^{w-1} - 32)M + (3 \cdot 2^{w-1} - 6)S. \end{aligned}$$

For the fist stage of doublings ($[2^w \cdot U_d]P$) we could potentially use the modification discussed in Section 4.2.2. As mentioned in Remark 4.3, the modification is valid for any $U_d > 0$. Therefore, with appropriate precautions, we could use the modification for $U_d < 0$ by setting $[2^w \cdot U_d]P = [-1 \cdot 2^w \cdot (-U_d)]P$. This would, however, corrupt the idea of having a uniform behaviour, as the power consumption corresponding to the first stage of doublings would wary according to the value of U_d. Because of this, we refrain from using the modification.

Algorithm 13 requires $w\lceil \frac{l}{w} \rceil \cdot$ ECDBL and $(\lceil \frac{l}{w} \rceil - 1) \cdot$ ECADD so, by using $\mathcal{A}$ for precomputations, we get a cost of

$$T_w^{\mathcal{A}} := PRE_w^{\mathcal{A}} + t(2\mathcal{A} = \mathcal{J}) + \left(w \cdot \left\lceil \frac{l}{w} \right\rceil - 1\right) \cdot t(2\mathcal{J}) + \left(\left\lceil \frac{l}{w} \right\rceil - 1\right) \cdot t(\mathcal{A} + \mathcal{J} = \mathcal{J}) + C,$$

where $C = I + 3M + S$ is the cost of converting the result from $\mathcal{J}$ to $\mathcal{A}$. By using $\mathcal{J}^c$ for precomputations, we get a cost of

$$T_w^{\mathcal{J}^c} := PRE_w^{\mathcal{J}} + t(2\mathcal{J}^c = \mathcal{J}) + \left(w \cdot \left\lceil \frac{l}{w} \right\rceil - 1\right) \cdot t(2\mathcal{J}) + \left(\left\lceil \frac{l}{w} \right\rceil - 1\right) \cdot t(\mathcal{J}^c + \mathcal{J} = \mathcal{J}) + C.$$

Using the values from Table 4.1, we get

$$T_w^{\mathcal{A}} = wI + \left(5 \cdot 2^{w-1} + \left\lceil \frac{l}{w} \right\rceil (4w + 8) - 3w - 9\right) M + \left(2^{w-1} + \left\lceil \frac{l}{w} \right\rceil (4w + 3) + w - 4\right) S$$

and

$$T_w^{\mathcal{J}^c} = I + \left(11 \cdot 2^{w-1} + \left\lceil \frac{l}{w} \right\rceil (4w + 11) - 3w - 40\right) M + \left(3 \cdot 2^{w-1} + \left\lceil \frac{l}{w} \right\rceil (4w + 3) - 9\right) S.$$

The values of $\min_w(T_w^{\mathcal{A}})$ and $\min_w(T_w^{\mathcal{J}^c})$ are shown in Figure 6.2 for $l = 192$ and $l = 521$ respectively.

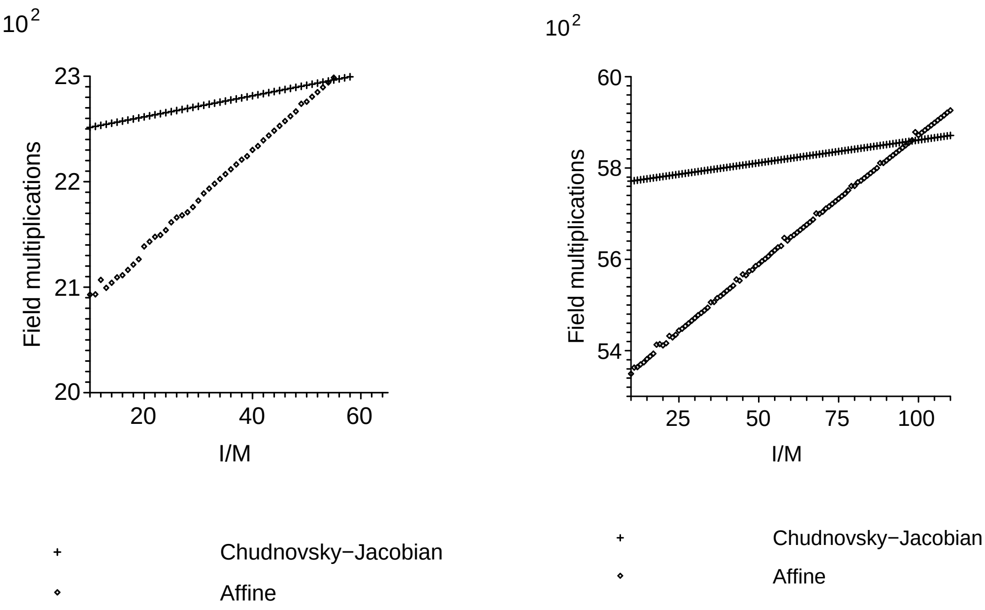


Figure 6.2: The plots show the number of field multiplications in $\min_w(T_w^{\mathcal{A}})$ and $\min_w(T_w^{\mathcal{J}^c})$ for $l = 192$ (left) and $l = 521$ (right).

As the figure indicates, we should choose $\mathcal{A}$ for precomputations, when I/M is less than some value depending on l. This value is shown in Table 6.2 for the applied values of l (see Section 2.2). As we have $I/M = 16$, we choose $\mathcal{A}$ for

	$l = 192$	$l = 224$	$l = 256$	$l = 384$	$l = 521$
v	55	61	67	82	98

Table 6.2: For different values of l, the table shows a value v which satisfies that $\mathcal{A}$ is the better choice for precomputations if $I/M < v$.

precomputations and get the average number of field multiplications shown in Table 6.3. The table also shows the overhead we introduce by using Algorithm 13 in preference to the efficient, non-secure scheme described in Section 5.3. Algorithm 13 needs to store twice as many precomputed points as the non-secure scheme. Additionally, it introduces dummy operations, so the algorithm is vulnerable to FI attacks.

In 1987, Montgomery proposed Algorithm 15 for scalar multiplication. The algorithm performs both an addition and a doubling for each bit of the scalar. Thereby, it makes SPA impossible. It does not introduce dummy operations, as every operation is used. The requirement of Algorithm 15 is

$$l \cdot \texttt{ECDBL} + (l - 1) \cdot \texttt{ECADD}.$$

	$l = 192$	$l = 224$	$l = 256$	$l = 384$	$l = 521$
$T_w^{\mathcal{A}}$	$2142M$ $w = 5$	$2448M$ $w = 5$	$2800M$ $w = 6$	$4039M$ $w = 6$	$5396M$ $w = 6$
Overhead	6.5%	5.2%	6.1%	4.5%	4.2%

Table 6.3: The table shows the average number of field multiplications required by Algorithm 13 and overhead compared to the efficient, non-secure scheme.

Algorithm 15 Montgomery's ladder algorithm

Input: $P \in E(\mathbb{F}_p)$ and $k = (k_{l-1} \cdots k_0)_2$.
Output: $[k]P \in E(\mathbb{F}_p)$.

```
1: P_1 ← P; P_2 ← [2]P;
2: i ← l − 2;
3: while i ≥ 0 do
4:    if k_i = 0 then
5:       P_2 ← P_1 ⊕ P_2; P_1 ← [2]P_1;
6:    else
7:       P_1 ← P_1 ⊕ P_2; P_2 ← [2]P_2;
8:    end if
9:    i ← i − 1;
10: end while
11: return P_1
```

What makes the Montgomery ladder algorithm interesting is that the structure of the algorithm allows for the use of efficient formulas for addition and doubling. Notice that, throughout Algorithm 15, the difference $P_2 - P_1$ is always equal to P at the beginning of the main loop in lines 3-10. Montgomery showed that for curves on the form

$$By^2 = x^3 + Ax^2 + x, \quad B \neq 0$$

(Montgomery form) in large characteristic, an `ECADD` can be performed in $4M+2S$ – provided that the difference between the addends is a known point. An `ECDBL` requires $3M + 2S$.

A curve in Montgomery form can always be converted into a curve in short Weierstraß form by setting $a = \frac{1}{B^2} - \frac{A^2}{3B^3}$ and $b = \frac{-A^3}{27B^3} - a\frac{A}{3B}$, but the converse is not true.

We say that two elliptic curves E and $\widetilde{E}$ are *isomorphic* over $\mathbf{K}$ if there exists $u \in \mathbf{K}^*$ and $r, s, t \in \mathbf{K}$ such that the map

$$(x, y) \mapsto (u^2x + r, u^3y + u^2sx + t)$$

transforms the equation of E into the equation of $\widetilde{E}$. The map given above is called *an admissible change of variables.* The general result is:

Theorem 6.1. *An elliptic curve* $E : y^2 = x^3 + ax + b$ *is isomorphic to a curve in Montgomery form if, and only if,*

1) $x^3 + ax + b$ *has at least one root* α *in* $\mathbb{F}_p$.

2) $3\alpha^2 + a$ *is a quadratic residue in* $\mathbb{F}_p$.

This result tells us that we cannot, in general, expect a curve in short Weierstraß form to have a Montgomery form representation. The NIST curves over prime fields have no Montgomery form representation, as the polynomial $x^3 - 3x + b$ is irreducible over $\mathbb{F}_p$ for these curves.

In 2002, Brier and Joye [BJ02] generalized Montgomery's idea to arbitrary curves in short Weierstraß form. Their result was:

Proposition 6.2. *Let* $\boldsymbol{K}$ *be a field with* $char(\boldsymbol{K}) \neq 2, 3$, *and let* $E : y^2 = x^3 + ax + b$ *be an elliptic curve over* $\boldsymbol{K}$. *Let* $P = (x_1, y_1)$ *and* $Q = (x_2, y_2)$ *be given such that* $P, Q \in E(\boldsymbol{K}) \setminus \{\mathcal{O}\}$ *and* $P \neq \pm Q$. *Let* $P - Q = (x_3, y_3)$. *Then, the x-coordinate* $x(P \oplus Q)$ *of* $P \oplus Q$ *satisfies*

$$x(P \oplus Q) \cdot x_3 = \frac{(x_1 x_2 - a)^2 - 4b(x_1 + x_2)}{(x_1 - x_2)^2}. \tag{6.2}$$

If $y_1 \neq 0$, *the x-coordinate* $x([2]P)$ *of* $[2]P$ *satisfies*

$$x([2]P) = \frac{(x_1^2 - a)^2 - 8bx_1}{4(x_1^3 + ax_1 + b)}. \tag{6.3}$$

The y-coordinate $y(P)$ *of* P *satisfies*

$$y(P) = \frac{2b + (a + x_3 x_1)(x_3 + x_1) - x_2(x_3 - x_1)^2}{2y_3}. \tag{6.4}$$

Notice that the y-coordinate does not appear anywhere in equations (6.2) and (6.3). Equation (6.4) ensures that the y-coordinate can be recovered.

To eliminate inversions in the addition formula, we will use projective coordinates (cf. Section 1.1). As $P, Q \neq \mathcal{O}$ and $P \neq -Q$, we have $x_1 = \frac{X_1}{Z_1}$, $x_2 = \frac{X_2}{Z_2}$ and $x(P \oplus Q) = \frac{X}{Z}$ for some $X_i, Z_i, Z \in \mathbf{K}$, $i = 1, 2, 3$, with $Z_1 Z_2 Z \neq 0$. Substituting into equation (6.2) gives

$$\begin{aligned}\frac{X}{Z} &= \frac{(\frac{X_1 X_2}{Z_1 Z_2} - a)^2 - 4b(\frac{X_1}{Z_1} + \frac{X_2}{Z_2})}{x_3(\frac{X_1}{Z_1} - \frac{X_2}{Z_2})^2} \\ &= \frac{(X_1 X_2 - aZ_1 Z_2)^2 - 4bZ_1 Z_2(X_1 Z_2 + X_2 Z_1)}{x_3(X_1 Z_2 - X_2 Z_1)^2}.\end{aligned}$$

From this we see that formulas for X and Z are

$$X = (X_1X_2 - aZ_1Z_2)^2 - 4bZ_1Z_2(X_1Z_2 + X_2Z_1)$$
$$Z = x_3(X_1Z_2 - X_2Z_1)^2. \tag{6.5}$$

Similar calculations result in the following formulas for the first and third projective coordinate of $[2]P = (X : Y : Z)$:

$$X = (X_1^2 - aZ_1^2)^2 - 8bX_1Z_1^3$$
$$Z = 4Z_1(X_1^3 + aX_1Z_1^2 + bZ_1^3). \tag{6.6}$$

When $a = -3$, an addition requires $7M + 2S$, while a doubling requires $5M + 3S$.

The first doubling in Algorithm 15 requires $t(2\mathcal{A} = \mathcal{P})$. Using formula (6.6) with $Z_1 = 1$, this can be done in $2M + 2S$.

The algorithm performs $(l - 1) \cdot (\texttt{ECDBL} + \texttt{ECADD})$. As $P_2 - P_1 = P$ (which is in $\mathcal{A}$), we can use formula (6.5) and (6.6) to get a cost of

$$\begin{aligned} T_1 &:= 2M + 2S + (l-1)\cdot(12M + 5S) \\ &= (12l - 10)M + (5l - 3)S. \end{aligned}$$

Recovering the y-coordinate of $[k]P$ is done by using equation (6.4) with $x_1 = \frac{X_1}{Z_1}$ and $x_2 = \frac{X_2}{Z_2}$. The recovery requires $2I + 2M$ (for calculating x_1 and x_2) plus $I + 4M + S$. This makes a cost of

$$T_2 := 3I + 6M + S$$

for the y-recovery. The total cost of Algorithm 15 is

$$\begin{aligned} T_{Montgomery} &:= T_1 + T_2 \\ &= 3I + (12l - 4)M + (5l - 2)S. \end{aligned}$$

This gives the values in table 6.4.

	$l = 192$	$l = 224$	$l = 256$	$l = 384$	$l = 521$
$T_{Montgomery}$	$3306M$	$3850M$	$4394M$	$6570M$	$8899M$
Overhead	64.4%	65.5%	66.4%	70.0%	71.9%

Table 6.4: The table shows the number of field multiplications required by Algorithm 15 and overhead compared to the efficient, non-secure scheme.

Remark 6.1. Montgomery's ladder algorithm maintains the invariant $P_2 - P_1 = P$. When the algorithm terminates, $P_1 = [k]P$, so $P_2 = [k+1]P$. This means that choosing $k = |E(\mathbb{F}_p)| - 1$ will result in $P_2 = \mathcal{O}$, and in this case equation (6.4) does not apply. This is not an issue of concern in the established literature on the subject; nonetheless, it should be noted that Algorithm 15 (using the result in Proposition 6.2) returns the x-coordinate of $[k]P$ for any $k \in [1, |E(\mathbb{F}_p)| - 1]$, but y-recovery is *not* possible for $k = |E(\mathbb{F}_p)| - 1$. When using the protocols in Section 2.1, this must be taken into consideration.

∘

Despite the substantial overhead involved, Montgomery's ladder algorithm is a useful alternative to the double-and-add always method and the w-double-and-add always method. It uses less field multiplications than the former, and, unlike the latter, it does not need storage for precomputed points. Furthermore, Algorithm 15 uses no dummy operations. Therefore, it is secure against FI attacks.

6.1.2 Algorithms with Unified Addition

Instead of altering the scalar multiplication scheme to ensure a fixed pattern of `ECDBL` and `ECADD`, one can make the `ECDBL` indistinguishable from the `ECADD`. This is done in [BJ02]. The starting point is Proposition 6.3:

Proposition 6.3. *Let $E : y^2 = x^3 + ax + b$ be an elliptic curve over a field $\boldsymbol{K}$ with $char(\boldsymbol{K}) \neq 2, 3$. Let $P = (x_1, y_1)$ and $Q = (x_2, y_2)$ be $\boldsymbol{K}$-rational points on E with $P, Q \neq \mathcal{O}$ and $P \neq -Q$. Then, $P \oplus Q = (x_3, y_3)$ with*

$$x_3 = \lambda^2 - x_1 - x_2, \quad y_3 = \lambda(x_1 - x_3) - y_1, \tag{6.7}$$

where

$$\lambda = \frac{x_1^2 + x_1 x_2 + x_2^2 + a}{y_1 + y_2}.$$

If we recall that $x_1^2 + x_1 x_2 + x_2^2 = (x_1 + x_2)^2 - x_1 x_2$, when $char(\mathbf{K}) \neq 2$, we see that the cost of an addition (which might be a doubling) using equation (6.7) is $I + 3M + 2S$.

In order to deduce formulas for projective coordinates, one notices that

$$\lambda = \frac{(x_1 + x_2)^2 - x_1 x_2 + a}{y_1 + y_2}$$

is symmetric in P and Q. As $(E(\mathbb{F}_p), \oplus)$ is abelian, equation (6.4) says that

$$y_3 = \lambda(x_2 - x_3) - y_2,$$

so

$$2y_3 = \lambda(x_1 + x_2 - 2x_3) - (y_1 + y_2).$$

Using this observation, one gets, by setting $x_i = \frac{X_i}{Z_i}$, $y_i = \frac{Y_i}{Z_i}$ for $i = 1, 2, 3$, that

$$\begin{aligned} x_3 = \frac{X_3}{Z_3} = & \\ & \frac{\left[(X_1Z_2 + X_2Z_1)^2 - X_1Z_2X_2Z_1 + a(Z_1Z_2)^2\right]^2}{(Z_1Z_2)^2(Y_1Z_2 + Y_2Z_1)^2} \\ & - \frac{Z_1Z_2(Y_1Z_2 + Y_2Z_1)^2(X_1Z_2 + X_2Z_1)}{(Z_1Z_2)^2(Y_1Z_2 + Y_2Z_1)^2} \end{aligned}$$

and

$$\begin{aligned} 2y_3 = \frac{2Y_3}{Z_3} = & \\ & 3\left[(X_1Z_2 + X_2Z_1)^2 - X_1Z_2X_2Z_1 + a(Z_1Z_2)^2\right](X_1Z_2 + X_2Z_1)Z_1Z_2(Y_1Z_2 + Y_2Z_1)^2 \\ & - \frac{2\left[(X_1Z_2 + X_2Z_1)^2 - X_1Z_2X_2Z_1 + a(Z_1Z_2)^2\right]^3 + (Y_1Z_2 + Y_2Z_1)^4(Z_1Z_2)^2}{(Z_1Z_2)^2(Y_1Z_2 + Y_2Z_1)^2}. \end{aligned}$$

With a common denominator of $Z_3 = 2(Z_1Z_2(Y_1Z_2 + Y_2Z_1))^3$, one gets

$$X_3 = 2JM, \quad Y_3 = H(L - 2M) - K^2, \quad Z_3 = 2J^3, \text{ where}$$

$$\begin{aligned} & A = Z_1Z_2, \quad B = X_1Z_2, \quad C = X_2Z_1, \quad D = Y_1Z_2, \quad E = Y_2Z_1, \\ & F = B + C, \quad G = D + E, \quad H = F^2 - BC + aA^2, \quad J = AG, \\ & K = GJ, \quad L = FK, \quad M = H^2 - L. \end{aligned} \tag{6.8}$$

When $a = -3$, the *unified addition formula* (6.8) requires $12M + 5S$. If one point is given in affine coordinates, the requirement drops to $9M + 5S$.

As opposed to Montgomery's ladder algorithm, scalar multiplication using the unified addition formula does not exclude the possibility of precomputing points, so one can use an adapted version of Algorithm 8. In [BSS04] the authors give an analogue of the addition-subtraction method (Algorithm 6) which is adapted to the use of formula (6.8). No algorithm adapted to a scalar in NAF_w is given, so we construct one. Algorithm 16 shows the result, in which δ and φ are given by

$$\delta(k_i) = \begin{cases} 1, & k_i \neq 0 \\ 0, & k_i = 0 \end{cases}, \quad \varphi(\sigma, k_i) = \begin{cases} k_i, & \sigma = 0 \\ 0, & \sigma \neq 0 \end{cases}.$$

Algorithm 16 Scalar multiplication with unified addition formulas

Input: $P \in E(\mathbb{F}_p)$, $w > 1$ and $k = (d_l \cdots d_0)_{NAF_w}$.
Output: $[k]P \in E(\mathbb{F}_p)$.
1: Compute the odd multiples $[3]P, \dots [2^{w-1}-1]P$.
2: $(R_1, R_3, \dots, R_{2^{w-1}-1}) \leftarrow (P, [3]P, \dots [2^{w-1}-1]P)$;
3: $R_0 \leftarrow [d_l]P$;
4: $i \leftarrow l-1$; $\sigma \leftarrow 0$;
5: **while** $i \geq 0$ **do**
6: $\quad R_0 \leftarrow R_0 \oplus R_\sigma$; //Use unified addition and doubling.
7: $\quad \sigma \leftarrow \varphi(\sigma, d_i)$;
8: $\quad i \leftarrow i + \delta(d_i) - 1$;
9: **end while**
10: **return** R_0

Algorithm 16 performs the same operation for each iteration of the main loop in lines 5-9. This makes it secure against SPA. Precomputations can a priori be done in $\mathcal{A}$ or $\mathcal{P}$. While it may be tempting to precompute in $\mathcal{A}$ in order to make use of the efficient mixed addition in formula (6.8), one must bare in mind that our goal is to maintain indistinguishability of `ECDBL` and `ECADD`. If the precomputed points are represented in $\mathcal{A}$, then all points must be represented in $\mathcal{A}$. Otherwise, doublings will consume more power than additions – resulting in a power consumption trace like the one in figure 6.1 (only with `ECDBL` consuming more power). This would make the algorithm vulnerable to SPA.

If points are represented in $\mathcal{P}$, we use the idea from precomputation scheme (d) in Section 4.2.1 to get a cost of

$$\begin{aligned} PRE_w^{\mathcal{P}} &= t(2\mathcal{A} = \mathcal{P}) + t(\mathcal{A} + \mathcal{P} = \mathcal{P}) + (2^{w-2} - 2)t(\mathcal{P} + \mathcal{P}) \\ &= (3 \cdot 2^w - 11)M + (2^{w-1} + 2)S \end{aligned}$$

for the precomputations. Using formula (6.8), Algorithm 16 performs $l + \frac{l}{w+1}$ additions on average. With $C = I + 2M$ (the cost of converting $[k]P$ from $\mathcal{P}$ to $\mathcal{A}$), the total cost becomes

$$\begin{aligned} T_w^{\mathcal{P}} &= PRE_w^{\mathcal{P}} + \left(l + \frac{l}{w+1}\right) \cdot (12M + 5S) + C \\ &= I + \left(3 \cdot 2^w + 12l\left(1 + \frac{1}{w+1}\right) - 9\right) M + \\ &\quad \left(2^{w-1} + 5l\left(1 + \frac{1}{w+1}\right) + 2\right) S \end{aligned}$$

on average.

Precomputations in $\mathcal{A}$ require

$$PRE_w^{\mathcal{A}} = (w-1)I + (5 \cdot 2^{w-2} + 2w - 12)M + (5 \cdot 2^{w-2} + 2w - 5)S.$$

The total average requirement, when using $\mathcal{A}$ for precomputation, is

$$\begin{aligned}
T_w^{\mathcal{A}} &= PRE_w^{\mathcal{A}} + \left(l + \frac{l}{w+1}\right) \cdot (I + 3M + 2S) \\
&= \left(w - 1 + l + \frac{l}{w+1}\right) I + \\
&\left(5 \cdot 2^{w-2} + 2w + +3l\left(1 + \frac{1}{w+1}\right) - 12\right) M + \\
&\left(2^{w-2} + 2w + 2l\left(1 + \frac{1}{w+1}\right) - 5\right) S.
\end{aligned}$$

The values of $\min_w(T_w^{\mathcal{P}})$ and $\min_w(T_w^{\mathcal{A}})$ are shown in Figure 6.3 for $l = 192$ and $l = 521$. For $I/M \geq 13$, projective coordinates are the better choice for all values

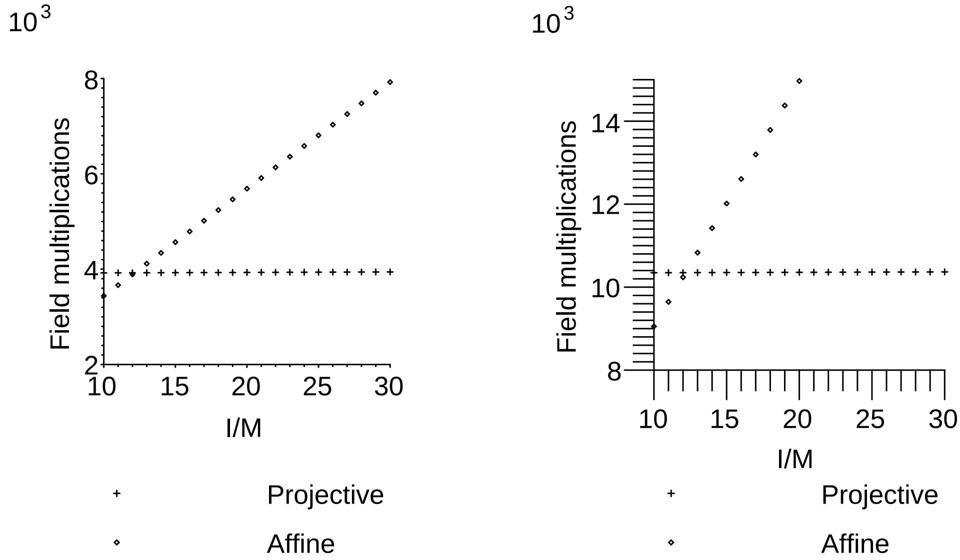


Figure 6.3: The plots show the number of field multiplications in $\min_w(T_w^{\mathcal{A}})$ and $\min_w(T_w^{\mathcal{P}})$ for $l = 192$ (left) and $l = 521$ (right) respectively.

of l. As we are working with $I/M = 16$, we choose $\mathcal{P}$ and get the values shown in Table 6.5. The use of precomputations implies the need for storing $2^{w-2} - 1$ points in memory. As the algorithm uses no dummy operations, it is secure against FI attacks. One should, however, be aware that other attacks against algorithms using unified addition has been proposed (see [SST04]).

	$l = 192$	$l = 224$	$l = 256$	$l = 384$	$l = 521$
$T_w^{\mathcal{P}}$	$3929M$ $w = 5$	$4564M$ $w = 5$	$5198M$ $w = 5$	$7694M$ $w = 6$	$10355M$ $w = 6$
Overhead	95.4%	96.2%	96.9%	99.0%	100.0%

Table 6.5: The table shows the average number of field multiplications required by Algorithm 16 and overhead compared to the efficient, non-secure scheme.

6.1.3 Algorithms with Dummy Field Operations

Section 6.1.3 is excluded from this version.

Section 6.1.3 is excluded from this version.

Section 6.1.3 is excluded from this version.

Section 6.1.3 is excluded from this version.

Section 6.1.3 is excluded from this version.

Section 6.1.3 is excluded from this version.

Section 6.1.3 is excluded from this version.

Section 6.1.3 is excluded from this version.

Section 6.1.3 is excluded from this version.

6.1.4 Comparison and Conclusion

We now compare the efficiency and security of the five SPA-secure scalar multiplication algorithms examined in Sections 6.1.1, 6.1.2 and 6.1.3. For timing purposes, the algorithms are implemented in Java (source code is enclosed in Appendix C.4). Timings are done using the same setup as the one described in Section 5.1.

Table 6.6 shows the number of field multiplications (M) required on average by the five methods, the number of points ($\sharp$) which need to be precomputed and timings of the implementations. For comparison, the same values are shown for our efficient, non-secure implementation.

From Table 6.6 one can see that an SPA countermeasure based on side channel atomicity is the better choice if speed is the primary focus. We have seen that the matrix used is small enough to make side channel atomicity more efficient with respect to storage requirements than the w-double-and-add always method (Algorithm 13), which precomputes twice as many points (even more in the case $l = 256$ when $I/M = 16$). Both methods use dummy operations and are ,therefore, vulnerable to FI-attacks. If one does not have extra storage at hand, Montgomery's ladder algorithm (Algorithm 15) should be used, as it is the fastest method among those which use the same storage as Algorithm 8 or less. Additionally, Montgomery's ladder algorithm uses no dummy operations, so it is secure against FI attacks.

Field multiplications:

Countermeasure	$l = 192$		$l = 224$		$l = 226$		$l = 384$		$l = 521$	
	M	$\sharp$	M	$\sharp$	M	$\sharp$	M	$\sharp$	M	$\sharp$
None (algorithm 8)	2011	7	2326	7	3640	7	3866	15	5177	15
Double-and-add always (algorithm 11)	3629	0	4237	0	4845	0	7277	0	9880	0
W-double-one-add always (algorithm 13)	2142	14	2448	14	2800	30	4039	30	5396	30
Montgomery's ladder algorithm (algorithm 15)⋆	3306	0	3850	0	4394	0	6570	0	8899	0
Unified addition (algorithm 16)⋆	3929	7	4564	7	5198	7	7694	15	10355	15
Side channel atomicity	2023	7	2338	7	2652	7	3878	15	5190	15
⋆ Secure against FI attacks.										

Timings:

Countermeasure	$l = 192$	$l = 224$	$l = 226$	$l = 384$	$l = 521$
	Time	Time	Time	Time	Time
None (algorithm 8)	$7352\,\mu s$	$10050\,\mu s$	$13698\,\mu s$	$34482\,\mu s$	$81\,ms$
Double-and-add always (algorithm 11)	$13513\,\mu s$	$18691\,\mu s$	$25641\,\mu s$	$68\,ms$	$166\,ms$
W-double-one-add always (algorithm 13)	$8368\,\mu s$	$11235\,\mu s$	$15503\,\mu s$	$38769\,\mu s$	$92\,ms$
Montgomery's ladder algorithm (algorithm 15)⋆	$12048\,\mu s$	$16806\,\mu s$	$22988\,\mu s$	$61\,ms$	$149\,ms$
Unified addition (algorithm 16)⋆	$12987\,\mu s$	$18018\,\mu s$	$24888\,\mu s$	$66\,ms$	$162\,ms$
Side channel atomicity	$8163\,\mu s$	$11173\,\mu s$	$15037\,\mu s$	$37735\,\mu s$	$87\,ms$
⋆ Secure against FI attacks.					

Table 6.6: The tables show the average number of field multiplications required (M), the number of precomputed points ($\sharp$) and timings of implementations of the secure algorithms presented in Sections 6.1.1, 6.1.2 and 6.1.3.

6.2 Differential Power Analysis

This section examines attacks of the kind described by Coron in [Cor99]. We assume that the scalar multiplication algorithm is secure against SPA, e.g. through implementation of one of the countermeasures discussed in Section 6.1. We examine the situation where an attacker is in possession of $n > 1$ power consumption traces corresponding to the calculation of $[k]P_1, \ldots, [k]P_n$ for known and distinct points $P_1, \ldots, P_n \in E(\mathbb{F}_p)$ (situations with $P_i = P_j$ for some i, j are more similar to SPA).

Let $\mathfrak{A}$ be a scalar multiplication algorithm, and let $G = \{g_1, \ldots, g_m\}$ be the set of logical gates in the hardware executing $\mathfrak{A}$. Let t_{max} be the maximum number of time units, e.g. ns or μs, required to execute $\mathfrak{A}$. Let

$$f(g, t), \quad g \in G,\, t \in [0, t_{max}]$$

denote the power consumption of gate g at time t. We aim at defining a function for measuring the total power consumption of the hardware at a given time during the execution of $\mathfrak{A}$. Such a function should take into account various sources of *noise* distorting the measurements. Sources include *external noise* (generated by some external object), *intrinsic noise* (generated by certain random movements within conductors in the hardware), *quantification noise* (from the quantizer in the analog-to-digital converter used to sample the power signals) and *algorithmic noise* (due to the random data being processed by the hardware). For details of noise characteristics see [MDS99].

We will take the approach of Oswald [AO00] and model the noise components as a normally distributed random variable $N(t) \in [0, \infty[$ for each $t \in [0, t_{max}]$. We define the *simple power model* F by

$$F(t) = \sum_{g \in G} f(g, t) + N(t), \quad t \in [0, t_{max}].$$

For every pair (g, t), we will view $f(g, t)$ as a random variable with unknown distribution. For every t we assume that $f(g_1, t), \ldots, f(g_m, t)$ are independent and identically distributed. The Central Limit Theorem says that $\frac{1}{m} \sum_{i=1}^{m} f(g_i, t)$ is (asymptotically) normally distributed, so for every t, $F(t)$ is normally distributed (viewing $F(t)$ as a random variable).

There are a lot of assumptions in the model described above. Not all of these assumptions can be proven valid, and one should be careful not to overestimate the scope of the simple power model. On the other hand, the simple power model is a priori the best model one can hope for, when doing cryptanalysis on a tamper-resistant device, and successful use of the model has been documented (see [MDS99] and [AO00]).

As in the previous section, the purpose of the attack is to find the value of $k = (k_{l-1} \cdots k_0)_2$. The attacker is assumed to know

(i) The points $P_1, \dots, P_n$.

(ii) The internal representation of points in the hardware.

(iii) The number of bits l in the binary representation of k.

(iv) The scalar multiplication scheme.

Assume that the s most significant bits $k_{l-1}, \dots, k_{l-s}$ of k are known to the attacker, who wants to find the value of k_{l-s-1}. The attack consists of five steps[1]:

1) The attacker makes a *guess* that $k_{l-s-1} = \kappa$, where $\kappa \in \{0, 1\}$.

2) He/she computes

$$Q_i = \left[\sum_{j=l-s-1}^{l-1} k_j \cdot 2^j \right] P_i, \quad i = 1, \dots, n.$$

These calculations can be carried out on a separate device with an implementation of the same scalar multiplication scheme as the one used by the target device (e.g. smart card).

3) Based on the knowledge of the representation of points in hardware, the attacker constructs a map

$$\Phi \,:\, E(\mathbb{F}_p) \to \{0, 1\}$$

such that $\Phi(P_i) = \Phi(P_j)$ if, and only if, the representations of P_i and P_j does not differ "significantly". This is a vague description, which must be made more precise in an concrete situation. We will assume that the Hamming weigh ν of the representation $rep(P_i)$ of a point P_i influences the power consumption in the system. We define Φ as

$$\Phi(P) = \begin{cases} 1, & \nu(rep(P)) \geq \nu_0 \\ 0, & \nu(rep(P)) < \nu_0 \end{cases}$$

for some fixed value ν_0. The map Φ is used to construct the sets

$$\mathcal{S}_0 := \{ i \,|\, \Phi(Q_i) = 0 \} \quad \text{and} \quad \mathcal{S}_1 := \{ i \,|\, \Phi(Q_i) = 1 \} .$$

This construction can be done on a separate device.

4) With a partitioning

$$0 = \Delta_1 < \Delta_2 < \cdots < \Delta_d = t_{max}$$

[1] One iteration over the five steps determines one bit of k. By repeating the five steps, one can recover all the bits of k, starting with the most significant one.

of $[0, t_{max}]$, the attacker constructs the vectors

$$(F_i(\Delta_1), \ldots, F_i(\Delta_d)), \quad i = 1, \ldots, n,$$

where $F_i(\Delta_j)$ is the value of F at time Δ_j during the calculation of $[k]P_i$. He/she sets

$$A_0(\Delta_j) := \frac{1}{|\mathcal{S}_0|} \sum_{i \in \mathcal{S}_0} F_i(\Delta_j), \quad A_1(\Delta_j) := \frac{1}{|\mathcal{S}_1|} \sum_{i \in \mathcal{S}_1} F_i(\Delta_j)$$

for $j = 1, \ldots, d$.

The collection of power measurements

$$F_i(\Delta_j), \quad i = 1, \ldots, n, \; j = 1, \ldots, d$$

requires access to the target device, while the calculation of

$$A_0(\Delta_j), \; A_1(\Delta_j), \quad j = 1, \ldots, d$$

can be done on a separate device.

5) If

$$\max_{1 \le j \le d} |A_0(\Delta_j) - A_1(\Delta_j)| \approx 0,$$

the calculation of Q_i never took place during the calculation of $[k]P_i$, i.e. the guess in step 1 was incorrect. In this case, the correct value of k_{l-s-1} is $\neg\kappa$. If

$$\max_{1 \le j \le d} |A_0(\Delta_j) - A_1(\Delta_j)| > 0,$$

the guess was correct, and the attacker proceeds to determine the next bit.

Determining whether the guess was correct or not can be done on a separate device.

Notice that the attacker only needs access to the target device during step 4. The rest of the attack can be carried out on a separate device. Figure 6.4 shows averaged traces corresponding to a wrong and correct guess. We have assumed that k is in binary representation, but the attack works for other representations of k as well.

Remark 6.2. The analysis performed in steps 1-5 above is largely a T-test for testing significant differences between two normally distributed observations, assumed to have the same variance. The method in steps 1-5 only take into account the empirical means of the two distributions, which is assumed to be sufficient (see [AO00]). Because of this, the method is also known as the *mean-method*.

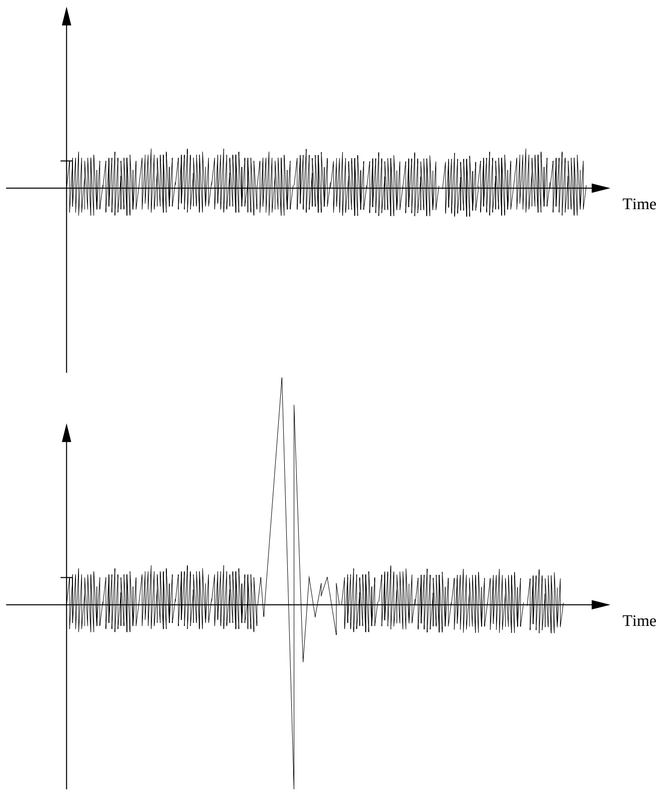


Figure 6.4: The figure shows schematic power consumption traces corresponding to a wrong (top) and a correct (bottom) DPA-guess respectively.

○

A necessary condition for being able to perform DPA is knowledge of the representation of the scalar k and the points $P_1, \ldots, P_n$. Because of this, countermeasures against DPA apply *randomness* to the scalar, the base point or the curve, making it impossible to perform the simulation in step 2 of the attack. We will consider the following randomization schemes:

- ◇ Scalar randomization by variation.
- ◇ Point randomization by blinding.
- ◇ Point randomization by redundancy.
- ◇ Curve randomization by curve isomorphisms.

Remark 6.3. Other randomization techniques are available (see for instance [ACD$^+$05] and [OA01]). The more prominent among these are scalar randomization by representation and curve randomization by field isomorphisms. The security of the former and the efficiency of the latter has, however, been questioned (see [Wal04] and [ACD$^+$05]).

6.2.1 Scalar randomization by variation

For all NIST curves, the group order $|E(\mathbb{F}_p)| = \sigma$ is a known prime number. For every $s \in \mathbb{Z}$, we have

$$[k]P = [k + s\sigma]P.$$

Hence, a randomization φ of the scalar k is given by $\varphi(k) = k + s\sigma$, where s is a random positive integer. The map φ should be applied every time a scalar multiplication is performed. Algorithm 17 shows the general method, where `ECMULT` is any scalar multiplication algorithm.

Algorithm 17 Scalar multiplication with randomized scalar

Input: $P \in E(\mathbb{F}_p)$ and $k \in \mathbb{Z}$, $k \geq 1$.
Output: $[k]P \in E(\mathbb{F}_p)$.
1: $k' \leftarrow \varphi(k)$;
2: $Q \leftarrow$ `ECMULT`(P, k');
3: **return** Q

The requirement of Algorithm 17 depends on the length of the binary representation of $k' := k + s\sigma$. We have that

$$\lceil \log_2(s) \rceil + \lceil \log_2(\sigma) \rceil \leq \lceil \log_2(k') \rceil \leq \lceil \log_2(s) \rceil + \lceil \log_2(\sigma) \rceil + 1,$$

as $k \in [1, \sigma - 1]$.

In order to thwart DPA, we want to ensure that the probability p of the same k' appearing two or more times during n independent executions of $[k']P$ is low. Assuming that the values of s are evenly distributed over $1, \ldots, 2^r - 1$ for $r = \lceil \log_2(s) \rceil$, one finds that

$$p = 1 - \frac{\prod_{i=2}^{n} (2^r - i)}{(2^r - 1)^{n-1}}.$$

An attack by Oswald & Aigner [AO00] on a DES implementation needed less than 200 samples to succeed. In light of this, we will demand that p is less than 10^{-5} for $n = 200$, i.e. we want the probability of the same k' appearing more than once during 200 independent executions to be less than 10^{-5}. A choice of $r = 32$ satisfies our demand.

We can use any algorithm as `ECMULT` in Algorithm 17, so we choose Algorithm 8 (with the modifications discussed in Sections 4.2.1 and 4.2.2). We have $\lceil \log_2(\sigma) \rceil = 192$, 224, 256, 384 and 521 for P-192,P-224,P-256,P-384 and P-521 respectively. Assume that $\lceil \log_2(k + s\sigma) \rceil = \lceil \log_2(s) \rceil + \lceil \log_2(\sigma) \rceil + 1$, and let l be the length of the binary representation of k. Table 6.7 shows the average number of field multiplications required by Algorithm 17. The overhead introduced by the countermeasure compared to the efficient, non-secure scheme is also shown.

	$l = 192$	$l = 224$	$l = 256$	$l = 384$	$l = 521$
Field multiplications	$2336M$	$2650M$	$2957M$	$4182M$	$5493M$
Optimal value of w	$w = 5$	$w = 5$	$w = 6$	$w = 6$	$w = 6$
Overhead	16.2%	14.0%	12.0%	8.2%	6.1%

Table 6.7: The table shows the average number of field multiplications required by Algorithm 17 and overhead compared to the efficient, non-secure scheme.

Not surprisingly, we see that the overhead for larger values of l reflects that the relative increase in the length of the binary representation of the scalar becomes smaller. Algorithm 17 uses no extra storage except for the case $l = 256$, which we will disregard.

A possible variant of DPA was described by Goubin. Assume that the algorithm for scalar multiplication has been secured against DPA by using a randomization scheme, such that the analysis in steps 1-5 is not possible. Also, assume that the algorithm has been secured against SPA by using the double-and-add always method in Algorithm 11. Assume that the curve contains a point P_0 with x- or y-coordinate equal to zero (this is the case with all NIST curves except for P-224, as these curves have b to be a quadratic residue modulo p). Assuming that the most significant bits $k_{l-1}, \ldots, k_{i+1}$ are known, the attacker makes a guess of $k_i = 0$ or $k_i = 1$ and defines the point

$$P_1 := \left[\left(\sum_{j=i+1}^{l-1} k_j 2^{j-i+1} + 1 + 2k_i \right)^{-1} \bmod |E(\mathbb{F}_p)| \right] P_0.$$

This is possible, as $|E(\mathbb{F}_p)|$ is a prime in the NIST recommendations.

The attacker now collects power consumption curves

$$C_j = \{(t, F_j(t)) \,|\, 0 \leq t \leq t_{max}\}, \quad j = 1, \ldots, n$$

corresponding to n executions of $[k]P_1$. Because of the randomization, the curves will all be different. However, if the randomization scheme preserves the zero-valued coordinate and the guess was correct, all curves will show the characteristics of operating on a point with a coordinate equal to zero. This will show up as peaks in the averaged curve

$$C = \left\{ \left(t, \frac{1}{n} \sum_{j=1}^{n} F_j(t) \right) \,\middle|\, 0 \leq t \leq t_{max} \right\}.$$

If C shows no peaks, the guess was incorrect. After having determined k_i, the attacker moves on to k_{i-1} and so forth. Similar attacks also exist for other SPA countermeasures.

Attacks of this type are known as *Goubin-type attacks*. Any countermeasure against DPA which leaves zero-valued coordinates unchanged is a priori vulnerable to Goubin-type attacks.

However, Although scalar randomization leaves zero-valued coordinates unchanged, the algorithm is *not* vulnerable to Goubin-type attacks, since the scalar is changed every time the algorithm is executed. The frequent changing of k also implies that implementing scalar randomization secures the algorithm against FI attacks.

Remark 6.4. If an attacker is able to mount an extremely precise FI attack, he or she may be able to perturb the calculation of $[k]P$ in such a way that P becomes a point P' on a less secure curve and that $[k]P'$ is calculated on this curve. By solving the ECDLP on the less secure curve, the attacker can determine the randomized value of k. Recovering the original k can then be done by brute force, trying the $2^{32}-1$ different values of $k - b \cdot |E(\mathbb{F}_p)|$ (notice that this requires knowledge of the point $[k]P$).

6.2.2 Point randomization by blinding

To simulate a random base point P, one can calculate

$$[k]P = [k](P \oplus Q) \oplus [k](-Q),$$

where Q is some point on the curve E being used. Finding a random point Q on E for each scalar multiplication being performed would require either

- calculating $[k]Q$ every time $[k]P$ is calculated (increasing running time by a factor two) or
- maintaining a table of pairs $(Q_i, [k]Q_i)$ containing every point Q_i on E (introducing a massive storage requirement).

Therefore, we select a set of points $\{Q_1, \ldots, Q_n\}$ on E, calculate $[k]Q_1, \ldots, [k]Q_n$ and store the pairs

$$\mathcal{R} = \{(Q_i, [k](-Q_i)) \,|\, i = 1, \ldots, n\}$$

in a table (notice that this scheme only applies to situations where a fixed scalar is used). A point Q and the corresponding $[k](-Q)$ can then be chosen at random from $\mathcal{R}$ on every execution of $[k]P$. The general method is shown in Algorithm 18. We use Algorithm 8 as `ECMULT` in line 3, so the addition in line 2 should be done in affine coordinates. Algorithm 18 introduces two additional `ECADD` compared to the efficient, non-secure scheme. The addition in line 4 can be done in mixed affine/Jacobian coordinates. The total cost of the two additions is

$$\begin{aligned} T_{Add} &= t(\mathcal{A} + \mathcal{A}) + t(\mathcal{J} + \mathcal{A} = \mathcal{J}) \\ &= I + 10M + 4S. \end{aligned}$$

Algorithm 18 Scalar multiplication with point blinding

Input: $P \in E(\mathbb{F}_p)$, $\mathcal{R}$ and $k \in \mathbb{Z}$, $k \geq 1$.
Output: $[k]P \in E(\mathbb{F}_p)$.
1: $(Q, [k](-Q)) \leftarrow (Q_{i_0}, [k](-Q_{i_0})) \in \mathcal{R}$; //Randomly chosen
2: $R \leftarrow P \oplus Q$;
3: $R \leftarrow$ ECMULT(R, k);
4: $R \leftarrow R \oplus [k](-Q)$;
5: **return** R

	$l = 192$	$l = 224$	$l = 256$	$l = 384$	$l = 521$
Field multiplications	$2041M$	$2356M$	$2670M$	$3896M$	$5207M$
Overhead	1.5%	1.3%	1.1%	0.8%	0.6%

Table 6.8: The table shows the average number of field multiplications required by Algorithm 18 and overhead compared to the efficient, non-secure scheme.

The total averages are shown in Table 6.8. As the table shows, the constant amount of extra field multiplications implies a small overhead for large values of l. Algorithm 18 requires storage for the table $\mathcal{R}$. As the algorithm does not preserve zero-valued coordinates, the scheme is secure against Goubin-type attacks.

6.2.3 Point randomization by redundancy

To reduce the number of extra field multiplications involved in a DPA countermeasure, one can randomize the base point P in a way different from the one described in Section 6.2.2. Recall, from Section 1.1, that a point $(\xi : \eta : \zeta)$ in Jacobian coordinates is equivalent to the point $(\lambda^2\xi : \lambda^3\eta : \lambda\zeta)$ for any $\lambda \in \mathbb{F}_p^*$. This makes it possible to construct an efficient randomization technique using redundant representations of the base point: Whenever a scalar multiplication is performed, one simply uses the map $(\xi : \eta : \zeta) \mapsto (\lambda^2\xi : \lambda^3\eta : \lambda\zeta)$ with a randomly chosen $\lambda \in \mathbb{F}_p^*$. Combining the randomization with Algorithm 8, we get the method shown in Algorithm 19. The randomization only introduces an extra $3M + S = 4M$. In order to be able to use the modifications from Section 4.2.2 to reduce the amount of initial doublings, one must, however, accept an overhead of $4M + S = 5M$, as the randomization should take place in $\mathcal{J}$, when using equation (4.6). The randomization is performed after the addition in equation (4.6) and before the doublings are done. Alternatively, the addition in equation (4.6) could be done by $\mathcal{J} + \mathcal{A} = \mathcal{J}$. However, this is not optimal, as $f(\mathcal{J} + \mathcal{A} = \mathcal{J}) > t(\mathcal{A} + \mathcal{A} = \mathcal{J}) + M$. We get the average requirements shown in Table 6.9.

Algorithm 19 Width-w NAF scalar multiplication with point randomization by redundancy.

Input: A point $P \in E(\mathbb{F}_p)$, $w > 1$, and $k = (d_l \cdots d_0)_{NAF_w}$.
Output: $[k]P \in E(\mathbb{F}_p)$.

Compute the odd multiples $[\pm 3]P, [\pm 5]P, \ldots, [\pm(2^{w-1}-1)]P$. //Use the modification from section 4.2.1.
$(\xi : \eta : 1) \leftarrow [d_l]P$; //Precomputed points are represented in $\mathcal{A}$.
Randomly choose $\lambda \in \mathbb{F}_p^*$.
$Q \leftarrow (\lambda^2\xi : \lambda^3\eta : \lambda)$; //Redundant representation of $[d_l]P$.
$i \leftarrow l - 1$;
while $i \geq 0$ **do**
  $Q \leftarrow [2]Q$;
  if $d_i \neq 0$ **then**
    $Q \leftarrow Q \oplus [d_i]P$;
  end if
  $i \leftarrow i - 1$;
end while
return Q in $\mathcal{A}$

	$l = 192$	$l = 224$	$l = 256$	$l = 384$	$l = 521$
Field multiplications	$2016M$	$2331M$	$2645M$	$3871M$	$5182M$
Overhead	0.24%	0.21%	0.19%	0.13%	0.10%

Table 6.9: The table shows the average number of field multiplications required by Algorithm 19 and overhead compared to the efficient, non-secure scheme.

Remark 6.5. Due to the small overhead of randomization by redundancy, one can perform several randomizations of the intermediate points in Algorithm 19 without introducing a high performance penalty.

○

Point randomization by redundancy preserves zero-valued coordinates. Therefore, the scheme is not secure against Goubin-type attacks.

6.2.4 Curve randomization by curve isomorphisms

Another possibility is to randomize the curve E itself. The idea is to pick a random curve $\widetilde{E}$ for which there exists an isomorphism

$$\psi : E(\mathbb{F}_p) \rightarrow \widetilde{E}(\mathbb{F}_p)$$

and calculate $[k]P$ as $\psi^{-1}([k]\psi(P))$. The situation is shown in the diagram in Figure 6.5. The ability to randomize the curve rests on the following proposition:

$$\begin{array}{ccc} P \in E(\mathbb{F}_p) & \longrightarrow & [k]P \in E(\mathbb{F}_p) \\ \psi \downarrow & & \uparrow \psi^{-1} \\ \widetilde{P} \in \widetilde{E}(\mathbb{F}_p) & \longrightarrow & [k]\widetilde{P} \in \widetilde{E}(\mathbb{F}_p) \end{array}$$

Figure 6.5: Calculation of $[k]P$ by using the isomorphism ψ.

Proposition 6.4. *Let $\boldsymbol{K}$ be a field with char$(\boldsymbol{K}) \neq 2, 3$. Let E and $\widetilde{E}$ be elliptic curves over $\boldsymbol{K}$ given by*

$$\begin{aligned} E &: y^2 = x^3 + ax + b \\ \widetilde{E} &: y^2 = x^3 + \tilde{a}x + \tilde{b}. \end{aligned}$$

The curves E and $\widetilde{E}$ are isomorphic if, and only if, there exists $u \in \boldsymbol{K}^*$ such that $\tilde{a} = u^{-4}a$ and $\tilde{b} = u^{-6}b$. In the affirmative, an isomorphism $\psi : E(\boldsymbol{K}) \to \widetilde{E}(\boldsymbol{K})$ is given by

$$\psi(P) = \begin{cases} (u^{-2}x, u^{-3}y), & P \neq \mathcal{O} \\ \mathcal{O}, & P = \mathcal{O}. \end{cases} \tag{6.9}$$

The inverse $\psi^{-1} : \widetilde{E}(\boldsymbol{K}) \to E(\boldsymbol{K})$ is given by

$$\psi^{-1}(P) = \begin{cases} (u^{2}x, u^{3}y), & P \neq \mathcal{O} \\ \mathcal{O}, & P = \mathcal{O}. \end{cases} \tag{6.10}$$

Applying Proposition 6.4 to the case of the NIST curves, we see that a curve

$$E : y^2 = x^3 - 3x + b \tag{6.11}$$

is isomorphic to a curve

$$\widetilde{E} : y^2 = x^3 + \tilde{a}x + \tilde{b} \tag{6.12}$$

if, and only if, $\tilde{a} = -3u^{-4}$ and $\tilde{b} = bu^{-6}$ for some $u \in \mathbb{F}_p^*$. Algorithm 20 uses this result to thwart DPA.

Algorithm 20 Width-w NAF scalar multiplication with curve randomization by isomorphism.

Input: A point $P = (x, y) \in E(\mathbb{F}_p)$, and a positive integer k.
Output: $[k]P \in E(\mathbb{F}_p)$.
1: Select a random $u \in \mathbb{F}_p^*$.
2: $\tilde{a} \leftarrow -3u^{-4}$;
3: $\widetilde{P} \leftarrow (u^{-2}x, u^{-3}y)$;
4: $(\tilde{x}, \tilde{y}) \leftarrow \texttt{ECMULT}(\widetilde{P}, k, \tilde{a})$;
5: **return** $(u^2\tilde{x}, u^3\tilde{y})$

Notice that, in line 4, we have to give $\tilde{a}$ as input to the scalar multiplication algorithm, as this element is used in the formulas for `ECDBL` and `ECADD`. The element $\tilde{b}$ is never used.

Apart from the cost of `ECMULT`, Algorithm 20 requires $I+6M+3S$. If $\tilde{a} \neq -3$, we cannot use our analysis from Section 4.2.3. In this case, one needs to perform a similar analysis using the general formulas from Section 1.1 to determine the number of field operations involved in the elliptic curve operations for different coordinates (see for instance [ACD+05] and [CMO98] for such an analysis). With the notation of Chapter 4, the conclusion is that if $a \neq -3$, $S = M$ and $I/M = 16$, one should represent the precomputed points in $\mathcal{A}$, perform $(\kappa_i - 1)$ doublings in $\mathcal{J}^m$ and one doubling from $\mathcal{J}^m$ to $\mathcal{J}$. Additions are done by $\mathcal{A} + \mathcal{J} = \mathcal{J}^m$. In other words, one should choose $(\mathcal{C}^1, \mathcal{C}^2, \mathcal{C}^3) = (\mathcal{J}^m, \mathcal{J}, \mathcal{A})$. Also, one should use the modifications from Sections 4.2.1 and 4.2.2.

To calculate the average number of field operations required by Algorithm 20, one needs to know the probability of the event $\tilde{a} = -3$ occurring. This event corresponds to $u^{-4} = 1$ for the randomly selected $u \in \mathbb{F}_p^*$, i.e. the event that the order of u is 1, 2 or 4. As $\mathbb{F}_p^*$ is cyclic, there is exactly one subgroup $H_1 \subset \mathbb{F}_p^*$ of order 2 and exactly one subgroup $H_2 \subset \mathbb{F}_p^*$ of order 4, if $4 \mid p-1$. If $4 \nmid p-1$, no subgroups of order 4 exist.

Assume that $4 \mid p-1$. We know that both H_1 and H_2 are cyclic. The subgroup H_1 contains one element g of order 2. The element g is also in H_2 which additionally contains two elements h_1 and h_2 of order 4. No subgroups of higher order contains elements of order 2 or 4 different from g, h_1 and h_2. This means that $\mathbb{F}_p^*$ contains exactly 4 elements of order $1, 2$ or 4. Therefore, the probability of $\tilde{a} = -3$ occurring is $\frac{4}{p-1}$, when u is selected randomly. Assuming that $4 \nmid p-1$, the probability is $\frac{2}{p-1}$. In the case of the NIST primes, the only p for which $4 \mid p-1$ is $p = p_{224}$.

Let $t(\texttt{ECMULT}_1)$ denote the average requirement of Algorithm 8 using

$$(\mathcal{C}^1, \mathcal{C}^2, \mathcal{C}^3) = (\mathcal{J}^m, \mathcal{J}, \mathcal{A})$$

(the case $\tilde{a} \neq -3$). Similarly, let $t(\texttt{ECMULT}_2)$ denote the average requirement of

Algorithm 8 with

$$(\mathcal{C}^1, \mathcal{C}^2, \mathcal{C}^3) = (\mathcal{J}, \mathcal{J}, \mathcal{A})$$

(the case $\tilde{a} = -3$). The average requirement of Algorithm 20 is

$$T_{\text{Alg. 20}} = I + 6M + 3S + \left(1 - \frac{r}{p-1}\right) \cdot t(\texttt{ECMULT}_1) + \frac{r}{p-1} \cdot t(\texttt{ECMULT}_2),$$

where $r = 4$ for $p = p_{224}$ and $r = 2$ otherwise. This gives the values shown in table 6.10.

	$l = 192$	$l = 224$	$l = 256$	$l = 384$	$l = 521$
Field multiplications	$2039M$	$2353M$	$2668M$	$3894M$	$5205M$
Overhead	1.4%	1.2%	1.0%	0.7%	0.5%

Table 6.10: The table shows the average number of field multiplications required by Algorithm 20 and overhead compared to the efficient, non-secure scheme.

Algorithm 20 is a less efficient countermeasure than point randomization using redundant representations (Algorithm 19). Even in the case $\tilde{a} = -3$, Algorithm 20 introduces an overhead of $I + 6M + 3S = 25M$, while Algorithm 19 requires only an extra $5M$.

Curve randomization preserves zero-valued coordinates. Therefore, the scheme is not secure against Goubin-type attacks.

6.2.5 Comparison and conclusion

Algorithms 17-20 are implemented in Java (source code is enclosed in Appendix C.5). Table 6.11 shows the number of field multiplications (M) required on average by the five methods and timings of the implementations. For comparison, the same values are shown for the efficient, non-secure scheme. As one can see from Table 6.11, point randomization by redundancy is the more efficient choice. As previously mentioned, this countermeasure provides no security against Goubin-type attacks. For curves containing no points with zero-valued coordinates or curves being used in protocols with short-term keys, this is not a problem, as Goubin-type attacks cannot be used in these situations. In all other cases, one should use point blinding or scalar randomization as countermeasure. We notice that scalar randomization requires precomputation of 15 points when $l = 256$ instead of the 7 points needed by the non-secured version. This special case is disregarded.

When choosing a countermeasure against DPA attacks, one must consider both the number of required field multiplications, storage requirements and vulnerability to Goubin-type attacks. Assuming that one wants to secure a scalar multiplication algorithm against DPA attacks and that Goubin-type attacks are disregarded, point randomization by redundancy should be used. If the algorithm should be secure against Goubin-type attacks, point randomization by blinding is the better choice. Scalar randomization is also an alternative, as this countermeasure secures the algorithm against both Goubin-type attacks and FI attacks.

Field multiplications:

Countermeasure	$l = 192$		$l = 224$		$l = 256$		$l = 384$		$l = 521$	
	M	$\sharp$	M	$\sharp$	M	$\sharp$	M	$\sharp$	M	$\sharp$
None (algorithm 8)	2011	7	2326	7	3640	7	3866	15	5177	15
Scalar randomization (algorithm 17)*	2336	7	2650	7	2957	15	4182	15	5493	15
Point randomization by blinding (algorithm 18)*	2031	7	2346	7	2661	7	3886	15	5198	15
Point randomization by redundancy (algorithm 19)	2006	7	2321	7	2636	7	3861	15	5173	15
Curve randomization by isomorphism (algorithm 20)	2029	7	2344	7	2658	7	3884	15	5196	15
$\star$: Secure against Goubin-type attacks.										

Timings:

Countermeasure	$l = 192$	$l = 224$	$l = 256$	$l = 384$	$l = 521$
	Time	Time	Time	Time	Time
None (algorithm 8)	7352 μs	10052 μs	13698 μs	34482 μs	81 ms
Scalar randomization (algorithm 17)*	8810 μs	11764 μs	16250 μs	38018 μs	87 ms
Point randomization by blinding (algorithm 18)*	7490 μs	10204 μs	13888 μs	35087 μs	82 ms
Point randomization by redundancy (algorithm 19)	7352 μs	10101 μs	13698 μs	34741 μs	81 ms
Curve randomization by isomorphism (algorithm 20)	7380 μs	10101 μs	13793 μs	35087 μs	81 ms
$\star$: Secure against Goubin-type attacks.					

Table 6.11: The table shows the average number of field multiplications required (M), number of points stored ($\sharp$) and timings of implementations.

Chapter 7

Securing an Implementation

The purpose of this chapter is to construct a scalar multiplication scheme which is secure against SPA and DPA. As is apparent form Chapter 6, such a construction is partially based on choices of what amount of extra storage one is willing to use and whether one wants security against FI attacks and/or Goubin-type attacks.

7.1 Combinations of Countermeasures

We need to examine all combinations of the following cases:

Storage

A: Unlimited

B: Limited (only storage available for the precomputed points in the non-secure version of algorithm 8.

Security against FI attacks

1: Yes

0: No

Security against Goubin-type attacks

1: Yes

0: No

A combination of unlimited storage, security against FI attacks and Goubin-type attacks are written as (A,1,1). Similar notation is used for the remaining combinations. Regardless of the combinations, the resulting algorithm must *always* be secure against both SPA and DPA. For each of the eight combinations of the conditions, we seek a pair (M_1, M_2), where M_1 is a countermeasure against SPA and M_2 is a countermeasure against DPA. We want the implementation of the combined countermeasures to involve the least possible overhead compared to the efficient, non-secure version. There are eight combinations to examine. In the sequel, the countermeasures will be denoted as follows:

DA := Double-and-add always (Algorithm 11)

WD := W-double-and-add always (Algorithm 13)

MG := Montgomery's ladder algorithm (Algorithm 15)

UA := Unified addition (Algorithm 16)

AT := Side channel atomicity

SR := Scalar randomization (Algorithm 17)

PB := Point randomization by blinding (Algorithm 18)

PR := Point randomization by redundancy (Algorithm 19)

CR := Curve randomization (Algorithm 20)

(A,1,1): We assume unlimited storage available and want security against both FI attacks and Goubin-type attacks. The straightforward choice is $(M_1, M_2) = (\mathtt{MG}, \mathtt{PB})$, as Montgomery's ladder algorithm is the only SPA countermeasure which is secure against FI attacks, and point blinding is the most efficient DPA countermeasure with security against Goubin-type attacks. This results in a total cost of

$$t(\text{Algorithm 15}) + I + 10M + 4S.$$

We can, however, do better if we remember, that SR is a DPA countermeasure which provides security against both FI attacks *and* Goubin-type attacks. Therefore, $(M_1, M_2) = (\mathtt{AT}, \mathtt{SR})$ is the optimal choice. Table 7.1 shows the average number of field operations required by this combined countermeasure.

	$l = 192$	$l = 224$	$l = 256$	$l = 384$	$l = 521$
Field multiplications	$2348M$	$2662M$	$2969M$	$4194M$	$5505M$
Overhead	16.8%	14.4%	12.5%	8.5%	6.3%

Table 7.1: The table shows the average number of required field multiplications for scalar multiplication with countermeasures $(M_1, M_2) = (\mathtt{AT}, \mathtt{SR})$ and the overhead compared to the efficient, non-secure scheme.

(A,1,0): We assume that we have unlimited storage available. We want security against FI attacks and disregard Goubin-type attacks. One combined countermeasure, which satisfies the conditions, is $(M_1, M_2) = (\mathtt{MG}, \mathtt{PR})$. However, the large overhead of MG makes the combination inferior to $(M_1, M_2) = (\texttt{AT}, \texttt{SR})$, which is the optimal choice. Table 7.1 shows the average number of required field multiplications.

(A,0,1): We assume that we have unlimited storage available. We disregard FI attacks and want security against Goubin-type attacks. As AT is the most efficient SPA countermeasure, we set $M_1 = \texttt{AT}$. The most efficient DPA countermeasure with security against Goubin-type attacks is PB. Therefore, we get $(M_1, M_2) = (\texttt{AT}, \texttt{PB})$ to be optimal. Table 7.2 shows the average number of required field multiplications.

	$l = 192$	$l = 224$	$l = 256$	$l = 384$	$l = 521$
Field multiplications	$2053M$	$2368M$	$2682M$	$3908M$	$5220M$
Overhead	2.1%	1.8%	1.6%	1.1%	1.0%

Table 7.2: The table shows the average number of required field multiplications for scalar multiplication with countermeasures $(M_1, M_2) = (\texttt{AT}, \texttt{PB})$ and overhead compared to the efficient, non-secure scheme.

(A,0,0): We assume that we have unlimited storage available and disregard FI attacks and Goubin-type attacks. This is the most straightforward case, and we get the optimal choice to be $(M_1, M_2) = (\texttt{AT}, \texttt{PR})$. Table 7.3 shows the average number of required field multiplications.

	$l = 192$	$l = 224$	$l = 256$	$l = 384$	$l = 521$
Field multiplications	$2028M$	$2343M$	$2657M$	$3883M$	$5195M$
Overhead	0.85%	0.73%	0.64%	0.44%	0.35%

Table 7.3: The table shows the average number of required field multiplications for scalar multiplication with countermeasures $(M_1, M_2) = (\texttt{AT}, \texttt{PR})$ and overhead compared to the efficient, non-secured version.

(B,1,1): We assume that we have limited storage available. We want security against both FI attacks and Goubin-type attacks. We cannot use $M_1 = \texttt{AT}$ or $M_1 = \texttt{WD}$, because of the need to store the matrix and extra precomputed points respectively. Similarly, we cannot use $M_2 = \texttt{PB}$, because of the need to store the table of points. Therefore, the optimal choice is $(M_1, M_2) = (\texttt{MG}, \texttt{SR})$. Table 7.4 shows the number of required field multiplications.

(B,1,0): We assume that we have limited storage available. We want security against FI attacks and disregard Goubin-type attacks. As in the previous case,

	$l = 192$	$l = 224$	$l = 256$	$l = 384$	$l = 521$
Field multiplications	$3867M$	$4411M$	$4955M$	$7131M$	$9460M$
Overhead	92.3%	89.6%	87.7%	84.5%	82.7%

Table 7.4: The table shows the number of required field multiplications for scalar multiplication with countermeasures $(M_1, M_2) = (\texttt{MG}, \texttt{SR})$ and overhead compared to the efficient, non-secure scheme.

we have $M_1 \neq \texttt{AT}, \texttt{WD}$, because of the storage requirements. The optimal choice is $(M_1, M_2) = (\texttt{MG}, \texttt{PR})$. Table 7.5 shows the number of required field multiplications.

	$l = 192$	$l = 224$	$l = 256$	$l = 384$	$l = 521$
Field multiplications	$3311M$	$3855M$	$4399M$	$6575M$	$8904M$
Overhead	64.6%	65.7%	66.6%	70.1%	72.0%

Table 7.5: The table shows the number of required field multiplications for scalar multiplication with countermeasures $(M_1, M_2) = (\texttt{MG}, \texttt{PR})$ and overhead compared to the efficient, non-secure scheme.

(B,0,1): We assume that we have limited storage available. We disregard FI attacks and want security against Goubin-type attacks. The storage limitations still exclude AT and WD as SPA countermeasures and PB as DPA countermeasure. Therefore, we choose $(M_1, M_2) = (\texttt{MG}, \texttt{SR})$. Table 7.4 shows the number of required field multiplications.

(B,0,0): We assume that we have limited storage available. We disregard FI attacks and Goubin-type attacks. The storage limitations, once again, exclude $M_1 = \texttt{AT}, \texttt{WD}$. Therefore, the optimal choice is $(M_1, M_2) = (\texttt{MG}, \texttt{PR})$. Table 7.5 shows the average number of required field multiplications.

7.2 Comparison and Conclusion

We now compare the five combinations of countermeasures selected in Section 7.1. The five combinations are:

$$(M_1, M_2) = \begin{cases} (\texttt{AT}, \texttt{SR}) \\ (\texttt{AT}, \texttt{PB}) \\ (\texttt{AT}, \texttt{PR}) \\ (\texttt{MG}, \texttt{SR}) \\ (\texttt{MG}, \texttt{PR}) \end{cases}$$

Table 7.6 summarizes the number of required field multiplications, the number of precomputed points and the overhead compared to the efficient, non-secure scheme. Timings of implementations of scalar multiplication using the combined countermeasures are also shown in the table (source code for all implementations is enclosed in Appendix C.6). The timings are done as described in Section 5.1.

Field multiplications:

	$l = 192$		$l = 224$		$l = 256$		$l = 384$		$l = 521$	
Countermeasure	M	$\sharp$	M	$\sharp$	M	$\sharp$	M	$\sharp$	M	$\sharp$
None	2011	7	2326	7	2640	7	3866	15	5177	15
(AT, SR)*	2348	7	2662	7	2969	15	4194	15	5505	15
(AT, PB)**	2053	7	2368	7	2682	7	3908	15	5220	15
(AT, PR)*	2028	7	2343	7	2657	7	3883	15	5195	15
(MG,SR)	3867	0	4411	0	4955	0	7131	0	9460	0
(MG,PR)	3311	0	3855	0	4399	0	6575	0	8904	0

$\star$ Needs storage for matrix.
$\star\star$ Needs storage for matrix and table of points.

Overhead:

	$l = 192$	$l = 224$	$l = 256$	$l = 384$	$l = 521$
Countermeasure	Overhead	Overhead	Overhead	Overhead	Overhead
(AT,SR)	16.8%	14.4%	12.5%	8.5%	6.3%
(AT,PB)	2.1%	1.8%	1.6%	1.1%	1.0%
(AT,PR)	0.85%	0.73%	0.64%	0.44%	0.35%
(MG,SR)	92.3%	89.6%	87.7%	84.5%	82.7%
(MG,PR)	64.6%	65.7%	66.6%	70.1%	72.3%

Timings:

	$l = 192$	$l = 224$	$l = 256$	$l = 384$	$l = 521$
Countermeasure	Time	Time	Time	Time	Time
None	7352 μs	10052 μs	13698 μs	34482 μs	81 ms
(AT,SR)	9803 μs	13333 μs	17391 μs	41 ms	93 ms
(AT,PB)	8333 μs	11299 μs	15267 μs	38037 μs	87 ms
(AT,PR)	8196 μs	11173 μs	15037 μs	37735 μs	87 ms
(MG,SR)	14285 μs	19230 μs	25974 μs	67 ms	159 ms
(MG,PR)	12195 μs	16949 μs	22988 μs	62 ms	150 ms

Table 7.6: The tables show the average number of field multiplications (M), number of precomputed points ($\sharp$) and timings of implementations using the combined SPA/DPA countermeasures.

We will assume that the algorithm must always be secure against both FI attacks and Goubin-type attacks. From the discussion above, one sees that we should choose $(M_1, M_2) = (\texttt{MG}, \texttt{SR})$ if storage is limited and $(M_1, M_2) = (\texttt{AT}, \texttt{SR})$ otherwise.

We now compare the fully secured implementation to the non-secure test implementation from IBM and to our efficient, non-secure implementation (both described in Chapter 5). In the sequel, $T_{(M_1,M_1)}$ denotes the number of field multiplications required on average by our scalar multiplication scheme with countermeasures M_1 and M_2, while T_{IBM} and $T_{Efficient}$ denote the number of field multiplications required on average by the scheme implemented by IBM and our efficient, non-secure scheme respectively. Table 7.7 shows the average number of field multiplications and timings. As the table shows, one can achieve an imple-

	$l = 192$	$l = 224$	$l = 226$	$l = 384$	$l = 521$
	M/Time	M/Time	M/Time	M/Time	M/Time
T_{IBM}	$5056M/$ $15625\,\mu s$	$5899M/$ $22222\,\mu s$	$6741M/$ $30769\,\mu s$	$10112M/$ $80\,ms$	$13720M/$ $175\,ms$
$T_{Efficient}$	$2011M/$ $7352\,\mu s$	$2326M/$ $10052\,\mu s$	$2640M/$ $13698\,\mu s$	$3866M/$ $34482\,\mu s$	$5177M/$ $81\,ms$
$T_{(\texttt{MG},\texttt{SR})}$	$3867M/$ $14285\,\mu s$	$4411M/$ $19230\,\mu s$	$4955M/$ $25974\,\mu s$	$7131M/$ $67\,ms$	$9460M/$ $159\,ms$
$T_{(\texttt{AT},\texttt{SR})}$	$2348M/$ $9803\,\mu s$	$2662M/$ $13333\,\mu s$	$2969M/$ $17391\,\mu s$	$4194M/$ $41\,ms$	$5505M/$ $93\,ms$

Table 7.7: The table shows the average number of field multiplications and timings of implementations.

mentation which is secure against SPA, DPA, FI attacks and Goubin-type attacks and which uses approximately 57% less field multiplications than the scheme implemented by IBM does on average, if extra storage is available. If one cannot afford to use extra storage, the secure implementation requires approximately 27% less field multiplications than the scheme implemented by IBM does.

If extra storage is available, secure scalar multiplication introduces an average overhead of 12% compared to the efficient, non-secure scheme. If no extra storage is available, the secure version introduces an overhead of 87% on average.

The timings in Table 7.7 supports our conclusions in that the choices we have made remains advantageous in the experiments. However, the timings do not entirely match the number of field multiplications required by the individual schemes. Our assumptions on the time required to execute the individual field implementation is the cause of the discrepancies (cf. Section 5.3).

When combining SPA and DPA countermeasures, one must consider both the available storage and the need for security against FI attacks and Goubin-type attacks. We demand full SPA/DPA-security as well as security against both FI attacks and Goubin-type attacks. The result of Section 7.1 is that if storage is limited, one should use Montgomery's ladder algorithm as an SPA countermeasure and scalar randomization as a DPA countermeasure. If extra storage is available, one should use side channel atomicity as an SPA countermeasure and scalar randomization as a DPA countermeasure. Comparing our secured versions of the scalar multiplication algorithm with the scheme implemented by IBM, we see that, even with full security against SPA, DPA, FI attacks and Goubin-type attacks, we achieve a 57% reduction in field multiplications on average in the case where extra storage is available and a 27% reduction in the case where no extra storage is available. Compared to the efficient, non-secure scheme, the secure scheme introduces an average overhead of 12% in the case where extra storage is available and an average overhead of 87% if no extra storage is available. Timings of the implementations support our conclusions in that the choices we have made remains advantageous in the experiments.

Part IV

Conclusion

Chapter 8

Results and Recommendations

In this chapter we summarize the observations and results acquired in Part I, II and III of our examination. The goal is to sum up the necessary and recommended steps to take when implementing an efficient and secure scalar multiplication algorithm in an ECC-system. Our point of reference is the implementation provided by IBM. This is an implementation of the addition-subtraction method, using exclusively affine coordinates, which is developed solely for test purposes. It provides no security against side channel attacks.

We have chosen to focus on the NIST curves in our examination, as these curves are considered to be secure for cryptographic purposes. Additionally, these curves are described in details in standards and are used in real-life applications. We cover only NIST curves over prime fields.

In Chapter 2 we saw that the most time-consuming operation performed in an ECC-system is scalar multiplication. In Chapter 3 we performed an examination and comparison of various scalar multiplication methods with a greater degree of detail than other publications on the subject. We observed that NAF_w scalar multiplication (Algorithm 8) is the optimal choice. This method uses storage for precomputed points, but the storage requirement is acceptable compared to other scalar multiplication methods using precomputation.

In Chapter 4 an examination of different coordinate representations showed that, given our computational environment, we should choose affine coordinates for precomputed points, Jacobian coordinates for intermediate points being doubled and perform additions in mixed affine/Jacobian coordinates. We concluded that one should use Montgomery's trick of simultaneous inversions for the precomputations, and we constructed an algorithm for doing this. We also saw that one should take steps to reduce the number of initial doublings performed, and we deduced a formula for this purpose.

In Chapter 5 we showed that we achieve a 61% reduction in field multiplications compared to the scheme implemented by IBM on average, when using the scalar multiplication method, coordinate representations and optimizations described above. Timings of the implementations were documented and supported our conclusions.

In Chapter 6 we remarked that the existence of successful SPA/DPA attacks have shown that power analysis should be considered a threat against the security of ECC-systems. The established literature on elliptic curve cryptography describes various mathematical countermeasures against side channel attacks based on power analysis, but, so far, no comparisons of these countermeasures have been published. In Chapter 6 we performed such a comparison based on a detailed examination of a number of known countermeasures. We presented the overhead in field multiplications and extra storage requirements introduced by the countermeasures. We also documented timings of implementations of all countermeasures and evaluated the security of the countermeasures against FI attacks and Goubin-type attacks.

Section 6.1 showed that one must base countermeasures against SPA on the use of algorithms with uniform behaviour, unified addition formulas or dummy field operations. A countermeasure based on side channel atomicity was shown to be the most efficient SPA countermeasure. We constructed specifications for side channel atomic `ECDBL` adapted to the NIST curves and side channel atomic `ECADD` in mixed affine/Jacobian coordinates on the NIST curves. No such specifications have previously been published. Side channel atomicity requires extra storage for the matrix being used and is not secure against FI attacks. If one cannot afford to use extra storage, Montgomery's ladder algorithm should be used. Aside from introducing no extra storage requirements, Montgomery's ladder algorithm is also secure against FI attacks.

In Section 6.2 we saw that countermeasures against DPA are based on randomization. We showed that point randomization by redundancy is the better choice, when Goubin-type attacks are disregarded. If the algorithm must be secure against Goubin-type attacks, point randomization by blinding should be used. We noticed that scalar randomization provides security against both DPA, Goubin-type attacks *and* FI attacks. Scalar randomization requires extra storage for precomputed points, when the scalar is a 256-bit integer. We chose to disregard the extra requirement in this special case.

In Chapter 7 we constructed an efficient scalar multiplication scheme which is secure against both SPA, DPA, FI attacks and Goubin-type attacks. We showed that if one can afford to use extra storage, a combination of side channel atomicity and scalar randomization should be used. If no extra storage is available, one should use a combination of Montgomery's ladder algorithm and scalar randomization. When comparing our efficient, secure scheme to the scheme implemented by IBM, we saw that our version uses 57% fewer field operations in the case where extra storage is available and 27% fewer field operations in the case where no extra storage is available. We also saw that the efficient, secure scheme introduces an average overhead of 12% in the case where extra storage is available and 87% if no extra storage is available, compared to the efficient, non-secure scheme. Timings of the implementations were documented and supported our conclusions.

Based on the computational environment at hand, we have thus made optimal choices of

1) Scalar multiplication method.

2) Coordinate representations.

3) Countermeasures against SPA/DPA.

The resulting algorithms have been compared to the scheme implemented by IBM and timings of all implementations have been documented. We have developed an efficient scalar multiplication scheme which is secure against both SPA, DPA, FI attacks and Goubin-type attacks. Our efficient and secure scheme offers a higher degree of efficiency than the scheme implemented by IBM – both when storage is limited and when extra storage is available. This concludes our examination.

Part V

Appendix

Appendix A

Random Processes and Markov Chains

Markov chains is a useful tool when analyzing scalar multiplication methods. This section provides a brief introduction to the theory. The presentation is based on the one by Semay ([Sem04]).

A.1 Basic Definitions and Results

Let $(X_n)_{\mathbb{N}_0}$ be a sequence of random variables with $X_i \in \mathcal{S} = \{s_1, \ldots, s_k\}$ for all $i \in \mathbb{N}_0$ and some integer $k \geq 1$. The sequence $(X_n)_{\mathbb{N}_0}$ is known as a *random process* with *state space* $\mathcal{S}$.

Definition A.1 (Memoryless process)**.** The random process $(X_n)_{\mathbb{N}_0}$ is a *memoryless process* if

$$\begin{aligned}&\forall n \in \mathbb{N}_0 \, \forall i_0, \ldots, i_{n-1} \in \{1, \ldots, k\} \, \forall i, j \in \{1, \ldots, k\} \ : \\ &P(X_{n+1} = s_j \,|\, X_0 = s_{i_0}, \ldots, X_{n-1} = s_{i_{n-1}}, X_n = s_i) = \\ &P(X_{n-1} = s_j \,|\, X_n = s_i).\end{aligned}$$

◦

Considering n as a point in time, a memoryless process can be interpreted as a random process, for which the outcome of the next event in the process only depends on the outcome of the previous event (if any at all).

Definition A.2 (Homogeneity)**.** The random process $(X_n)_{\mathbb{N}_0}$ is *homogeneous* if

$$\begin{aligned}&\forall n, n' \in \mathbb{N}_0 \, \forall i, j \in \{1, \ldots, k\} \ : \\ &P(X_{n+1} = s_j \,|\, X_n = s_i) = P(X_{n'+1} = s_j \,|\, X_{n'} = s_i).\end{aligned}$$

If n denotes steps in time, we speak of *time homogeneity.*

Example A.1. As an example of a time homogeneous memoryless process, we will consider the process of starting a car in the morning $(X_0, X_1, \ldots, X_n, \ldots)$. The state space is $\mathcal{S} = \{$"The car starts", "The car doesn't start"$\}$. It is assumed, that the possibility of starting the car is only dependant on whether the car could start the day before or not and that the possibility of being able to start the car given that it could start the day before is the same at all times. The following probabilities are defined for the example:

$$
\begin{aligned}
P(X_{n+1} = \text{“The car starts”} \mid X_n = \text{“The car starts”}) &= \frac{7}{10} \\
P(X_{n+1} = \text{“The car doesn't start”} \mid X_n = \text{“The car starts”}) &= \frac{3}{10} \\
P(X_{n+1} = \text{“The car starts”} \mid X_n = \text{“The car doesn't start”}) &= \frac{4}{10} \\
P(X_{n+1} = \text{“The car doesn't start”} \mid X_n = \text{“The car doesn't start”}) &= \frac{6}{10}
\end{aligned}
$$

Let $s_1 =$ "The car starts" and $s_2 =$ "The car doesn't start". The matrix T below contains probabilities such that T_{ij} is the probability of getting from state j to state i in one step.

$$T = \begin{bmatrix} \frac{7}{10} & \frac{3}{10} \\ \frac{4}{10} & \frac{6}{10} \end{bmatrix},$$

∘

We now introduce the notion of a Markov chain:

Definition A.3 (Homogeneous Markov chain)**.** A a homogeneous memoryless process $M = (X_n)_{\mathbb{N}_0}$ with finite state space $\mathcal{S} = \{s_1, \ldots, s_k\}$ is said to be a *homogeneous Markov chain.* Let T be a $k \times k$ matrix such that

$$\forall i, j \in \{1, \ldots, k\}\, \forall n \in \mathbb{N}_0 \,:\, P(X_{n+1} = s_j \mid X_n = s_i) = T_{ij}.$$

The matrix T is called the *transition matrix* of M, and the entries of T are called *transition probabilities.*

∘

From definition A.3 we see, that every transition matrix T must satisfy

(i) $\forall i, j \in \{1, \ldots, k\} \,:\, T_{ij} \geq 0.$

(ii) $\forall i \in \{1, \ldots, k\} \,:\, \sum_{j=1}^{k} T_{ij} = 1.$

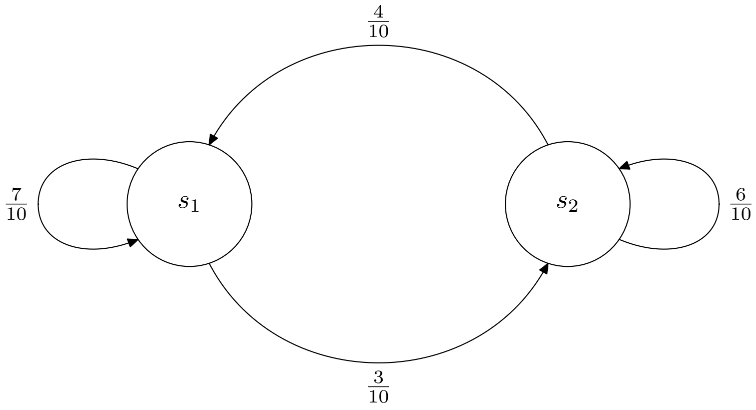


Figure A.1: Transition graph for the Markov chain in the car example.

It is often useful to illustrate a Markov chain with a *transition graph.* A transition graph is a graph $G = (N, V)$ with nodes N, vertices V and $|N| = k$, $|V| = k^2$ such that the nodes in N represent the states of the Markov chain and the vertices in V represent the transition probabilities. This means, that

$$\forall n_i, n_j \in N \, : \, (n_j, n_i) \in V \Leftrightarrow T_{ij} > 0.$$

Example A.2. The transition graph of the car example is shown in figure A.1.

Definition A.4 (Initial distribution)**.** The *initial distribution* of a Markov chain $(X_n)_{\mathbb{N}_0}$ with state space $\mathcal{S} = \{s_1, \ldots, s_k\}$ is a vector $\mu^{(0)} \in \mathbb{R}^k$ such that

$$\mu^{(0)} = (\mu_1^{(0)}, \ldots, \mu_k^{(0)}) = (P(X_0 = s_1), \ldots, P(X_0 = s_k)).$$

○

The initial distribution, in some sense, provides information about how the Markov chain "starts".

Example A.3. Returning to the car example, we assume that the car is brand new and in perfect condition. If we assume, that this is enough to ensure that the car will start the first day, we get the initial distribution $\mu^{(0)} = (1, 0)$.

○

Using notation similar to the one in definition A.4, we let $\mu^{(1)}, \mu^{(2)}, \ldots \in \mathbb{R}^k$ be given by

$$\mu^{(n)} = (\mu_1^{(n)}, \ldots, \mu_k^{(n)}) = (P(X_n = s_1), \ldots, P(X_n = s_k)),\ n \in \mathbb{N},$$

so $\mu^{(i)}$ represents the distribution of X_i. The distributions $\mu^{(1)}, \mu^{(2)}, \ldots$ can all be computed using the initial distribution and the transition matrix T:

Theorem A.1. *Let M be a Markov chain with initial distribution $\mu^{(0)}$ and transition matrix T. For all $n \in \mathbb{N}_0$, we have*

$$\mu^{(n)} = \mu^{(0)} T^n. \tag{A.1}$$

Proof: The proof is by induction on n, the case $n = 0$ being trivially true. Assume that $n > 0$ and that (A.1) holds for smaller n. One has that

$$\begin{aligned} \mu^{(0)} T^n &= (\mu^{(0)} T^{n-1}) T \\ &= \mu^{(n-1)} T \\ &= \mu^{(n)}, \end{aligned}$$

because for each $j = 1, \ldots, k$ we have

$$\begin{aligned} \mu_j^{(n)} &= P(X_{n-1} = s_1) P(X_n = s_j \mid X_n = s_1) + \cdots + \\ &\quad P(X_{n-1} = s_k) P(X_n = s_j \mid X_n = s_k) \\ &= \mu_1^{(n-1)} T_{1j} + \cdots + \mu_k^{(n-1)} T_{kj}. \end{aligned}$$

■

Corollary A.2. *The probability of being in state s_j at time n when starting in state s_i is*

$$P(X_n = s_j \mid X_0 = s_i) = (T^n)_{ij}.$$

Example A.4. In our car example, the probability that the car doesn't start at day one (the second day after having bought the car) equals the second coordinate of

$$\begin{aligned} \mu^{(1)} &= \mu^{(0)} T \\ &= (1, 0) \begin{pmatrix} \frac{7}{10} & \frac{3}{10} \\ \frac{2}{5} & \frac{3}{5} \end{pmatrix} \\ &= (\frac{7}{10}, \frac{3}{10}), \end{aligned}$$

i.e. the probability is $\frac{3}{10}$.

A.2 Properties

Three important properties of Markov chains, *irreducibility*, *aperiodicity* and *stationary distributions*, will play a role in our analysis. The properties will come into play, when the theorem about the asymptotic behaviour of certain Markov chains is stated in Section A.3.

If $T_{ij} > 0$ for some $i, j \in \{1, \ldots, k\}$, we write $s_i \to s_j$ and say that s_i *communicates* with s_j, meaning that there is a chance that state s_j will be reached in a finite number of steps when starting at state s_i.

Definition A.5 (Irreducible chain)**.** A Markov chain with state space $\mathcal{S} = \{s_1, \ldots, s_k\}$ is *irreducible* if $s_i \to s_j$ for all $s_i, s_j \in \mathcal{S}$. Otherwise the chain is said to be *reducible*.

○

In other words, a chain is irreducible if all states communicate with each other.

Example A.5. From the transition graph in figure A.1 one can see, that the Markov chain in our car example is irreducible as all states communicate with each other. Equivalently, one sees, that all the entries in the transition matrix are non-zero.

Definition A.6 (Aperiodicity)**.** Let M be a Markov chain with state space $\mathcal{S}$ and transition matrix T. The *period* $d(s_i)$ of a state $s_i \in \mathcal{S}$ is defined as

$$d(s_i) = \gcd(\{n \geq 1 \,|\, (T^n)_{ii} > 0\}).$$

If $d(s_i) = 1$, we say that s_i is *aperiodic*. M is said to be aperiodic if all states in $\mathcal{S}$ are aperiodic. Otherwise, M is said to be *periodic*.

○

The period of a state s_i is the greatest common divisor of the set of points in (discrete) time at which the chain has a chance of being in state s_i. It is assumed, that the starting state is s_i.

Example A.6. In the car example, we have $T_{11}, T_{22} > 0$, so

$$1 \in \{n \geq 1 \,|\, (T^n)_{ii} > 0\} \text{ for } i = 1, 2.$$

This gives $d(s_1) = d(s_2) = 1$, so the Markov chain is aperiodic.

Definition A.7 (Stationary distribution)**.** Let M be a Markov chain with finite state space and transition matrix T. A row vector $\pi = (\pi_1, \ldots, \pi_k)$ is said to be a *stationary distribution* for M if

(i) $\pi_i > 0, i = 1, \ldots, k$ and $\sum_{i=1}^{k} \pi_i = 1$.

(ii) $\pi T = \pi$.

○

This implies, that if π is a stationary distribution and $\mu^{(N)} = \pi$ for some N, then $\mu^{(n)} = \pi$ for all $n \geq N$. Condition (ii) in definition A.7 says, that the stationary distribution is a left eigenvector of T corresponding to the eigenvalue one.

Example A.7. In the car example, the distribution $\pi = (\frac{4}{7}, \frac{3}{7})$ is a stationary distribution.

A.3 Asymptotic Behaviour

What can be said about a Markov chain which has been running for a long time? More precisely: What happens to $\mu^{(n)}$ as $n \to \infty$? As we shall see, the distributions $\mu^{(n)}$ will converge to a fixed distribution under suitable circumstances. To apply meaning to this, one has to define, what is meant by convergence of sequences of probability distributions.

Definition A.8 (Total variation)**.** Let P and Q be probability distributions. The *total variation* $V(P, Q)$ is defined as

$$V(P, Q) = \sum_{a \in \mathbb{A}} |P(a) - Q(a)|.$$

Let $(P_n)_{\mathbb{N}}$ be a sequence of probability distributions. We say, that P_n *converges* to Q in total variation if $lim_{n \to \infty} V(P_n, Q) = 0$. In this case, we write $P_n \xrightarrow{V} Q$.

○

With the notion of convergence for distributions at hand, we can state the main theorem of this chapter:

Theorem A.3. *Let M be an irreducible aperiodic Markov chain with finite state space and initial distribution $\mu^{(0)}$. Then, there exists a unique stationary distribution π for M, and $\mu^{(n)} \xrightarrow{V} \pi$.*

Example A.8. The Markov chain in the car example is irreducible, aperiodic and has a finite state space. Theorem A.3 says, that

$$\mu^{(n)} \xrightarrow{V} \pi = \left(\frac{4}{7}, \frac{3}{7}\right),$$

so according to this model, the car would tend to start $\frac{4}{7} \approx 57\%$ of the mornings as the car got older.

Appendix B

Test Vectors

The tables in this chapter show the scalar k, the base point $P = (x_1, y_1)$ and the point $[k]P = (x_2, y_2)$ used in the timings of the operations on the NIST curves P-192, P-224, P-256, P-384 and P-521.

	P-192
k	0x 7FFFFFFF FFFFFFFF FFFFFFFF CCEF7C1B 0A35E4D8 DA691418
x_1	0x 188DA80E B03090F6 7CBF20EB 43A18800 F4FF0AFD 82FF1012
y_1	0x 07192B95 FFC8DA78 631011ED 6B24CDD5 73F977A1 1E794811
x_2	0x 7B4603CC 4AC84726 4022B071 44C25277 F2AD8FBE 9224728F
y_2	0x 7890050B B4048924 0DEBBC68 5B5B68A9 FE531DE5 9F92B5A2

Table B.1: Scalar k, base point $P = (x_1, y_1)$ and value of $[k]P = (x_2, y_2)$ for P-192

	P-224
k	0x 7FFFFFFF FFFFFFFF FFFFFFFF FFFF8B51 705C781F 09EE94A2 AE2E151E
x_1	0x B70E0CBD 6BB4BF7F 321390B9 4A03C1D3 56C21122 343280D6 115C1D21
y_1	0x BD376388 B5F723FB 4C22DFE6 CD4375A0 5A074764 44D58199 85007E34
x_2	0x E7F24028 5C2D03A7 EE519EFB 8DA70F8F F7292C0D F5E20B89 668CDDDA
y_2	0x D8DDF2DB A3C1E407 6BF19DC7 F0DCA56B A5BA9A1E A7FCBA26 CF993DEC

Table B.2: Scalar k, base point $P = (x_1, y_1)$ and value of $[k]P = (x_2, y_2)$ for P-224

P-256	
k	0x 7FFFFFFF 80000000 7FFFFFFF FFFFFFFF DE737D56 D38BCF42 79DCE561 7E3192A8
x_1	0x 6B17D1F2 E12C4247 F8BCE6E5 63A440F2 77037D81 2DEB33A0 F4A13945 D898C296
y_1	0x 4FE342E2 FE1A7F9B 8EE7EB4A 7C0F9E16 2BCE3357 6B315ECE CBB64068 37BF51F5
x_2	0x 2AFA386B 3F2BDCDB 83F4D83F 8FA3874D 7B74DCB4 54BD644F DD6BF3D1 F2DA8DB6
y_2	0x 72184BE1 CAA85634 62B536F1 0852D665 AE8A64FD F1EB8D4C 946AD589 796F729C

Table B.3: Scalar k, base point $P = (x_1, y_1)$ and value of $[k]P = (x_2, y_2)$ for P-256

P-384	
k	0x 7FFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF E3B1A6C0 FA1B96EF AC0D06D9 245853BD 76760CB5 666294B9
x_1	0x AA87CA22 BE8B0537 8EB1C71E F320AD74 6E1D3B62 8BA79B98 59F741E0 82542A38 5502F25D BF55296C 3A545E38 72760AB7
y_1	0x 3617DE4A 96262C6F 5D9E98BF 9292DC29 F8F41DBD 289A147C E9DA3113 B5F0B8C0 0A60B1CE 1D7E819D 7A431D7C 90EA0E5F
x_2	0x D36FED39 CA71063A 5163E811 9A37AFF1 0F6B86D5 0F02F1D3 24238D2B 090D8067 08495505 66396FF5 778738C0 B39B107A
y_2	0x 46C3E62B 85B82F0D DFACB8F5 32101B4B 82E07DB1 C8FDC36D 1F572843 416840AC DCF2BC1C BD532667 81FCFBA9 739AAE51

Table B.4: Scalar k, base point $P = (x_1, y_1)$ and value of $[k]P = (x_2, y_2)$ for P-384

P-521							
k	0x	000000FF	FFFFFFFF	FFFFFFFF	FFFFFFFF	FFFFFFFF	FFFFFFFF
		FFFFFFFF	FFFFFFFF	FFFFFFFD	28C343C1	DF97CB35	BFE600A4
		7B84D2E8	1DDAE4DC	44CE23D7	5DB7DB8F	489C3204	
x_1	0x	000000C6	858E06B7	0404E9CD	9E3ECB66	2395B442	9C648139
		053FB521	F828AF60	6B4D3DBA	A14B5E77	EFE75928	FE1DC127
		A2FFA8DE	3348B3C1	856A429B	F97E7E31	C2E5BD66	
y_1	0x	00000118	39296A78	9A3BC004	5C8A5FB4	2C7D1BD9	98F54449
		579B4468	17AFBD17	273E662C	97EE7299	5EF42640	C550B901
		3FAD0761	353C7086	A272C240	88BE9476	9FD16650	
x_2	0x	0000007C	1BB67BC4	F1A47A2C	AB98F683	2FD9681F	D803A639
		451943B3	5EEB82B7	05FD4132	7338840F	7B531313	F188DE7E
		42BB46B6	8E0FA5CB	05B53558	C1CA8E31	D783223F	
y_2	0x	000000E0	F5C012BC	C94FE001	953F1E6F	96550AE0	E02D9950
		D5014495	8EB2F55A	BDC30EAF	239F0274	00854830	6FCE7EFB
		146970BC	87CDAC12	D98D9376	DD2E3EBA	550A9CBF	

Table B.5: Scalar k, base point $P = (x_1, y_1)$ and value of $[k]P = (x_2, y_2)$ for P-521

Appendix C

Source Code

C.1 Field Implementations

C.1.1 Field Interface

```
interface IFieldElement{
    public IFieldElement add(IFieldElement val);
    public int compareTo(IFieldElement val);
    public boolean equals(java.lang.Object pObj);
    public IFieldElement inv();
    public IFieldElement mul(int n);
    public IFieldElement mul(IFieldElement val);
    public IFieldElement negate();
    public IFieldElement pow(int exp);
    public IFieldElement shl(int val);
    public IFieldElement shr(int val);
    public IFieldElement sqr();
    public IFieldElement sub(IFieldElement val);
    public java.math.BigInteger toBigInteger();
    public java.lang.String toString();
}
```


C.1.2 Implementation of $\mathbb{F}_{p_{192}}$

```
import java.math.BigInteger;
import java.lang.Math;
import java.util.Random;

public final class P192Impl implements IFieldElement{
    private BigInteger n;
    private static final BigInteger p192 =
        new BigInteger("
            6277101735386680763835789423207666416083908700390324961279
            ");

    /**
     * Constructor
     * @param a
     */
    public P192Impl(BigInteger a){
        n = a;
    }

    /**
     * Adddition
     * @param val
     */
    public P192Impl add(IFieldElement val){
        BigInteger b = val.toBigInteger();
        BigInteger c = modularAdd(n,b);
        return new P192Impl(c);
    }

    /**
     * Compare
     * @param val
     */
    public int compareTo(IFieldElement val){
        return n.compareTo(val.toBigInteger());
    }

    /**
     * Equality testing
     * @param pObj
```

```java
    */
    public boolean equals(java.lang.Object pObj){
        return n.equals(pObj);
    }

    /**
     * Inversion
    */
    public P192Impl inv(){
        return new P192Impl(n.modInverse(p192));
    }

    /**
     * Multiplication by integer
     * @param m
    */
    public P192Impl mul(int m){
        return new P192Impl(n.multiply(BigInteger.valueOf(m)).mod(p192));
    }

    /**
     * Multiplication
     * @param val
    */
    public P192Impl mul(IFieldElement val){
        return new P192Impl(n.multiply(val.toBigInteger()).mod(p192));
    }

    /**
     * Negation
    */
    public P192Impl negate(){
        return new P192Impl(n.negate().mod(p192));
    }

    /**
     * Exponentiation
     * @param exp
    */
    public P192Impl pow(int exp){
        return new P192Impl(n.pow(exp).mod(p192));
    }

    /**
     * Multiplication by power of two
     * @param value
    */
    public P192Impl shl(int value){
        return new P192Impl(n.shiftLeft(value).mod(p192));
    }

    /**
     * Division by power of two
     * @param value
    */
    public P192Impl shr(int value){
        return new P192Impl(n.shiftRight(value).mod(p192));
    }

    /**
     * Squaring
    */
    public P192Impl sqr(){
        return new P192Impl(n.pow(2).mod(p192));
    }

    /**
     * Subtraction
     * @param val
    */
    public P192Impl sub(IFieldElement val){
        BigInteger c = modularSub(n, val.toBigInteger());
        return new P192Impl(c);
    }

    /**
     * Conversion to BigInteger
    */
    public BigInteger toBigInteger(){
        return n;
    }

```

```
    /**
     * Conversion to String
    */
    public String toString(){
        return n.toString();
    }

    /**
     * Addition modulo p
     * @param a
     * @param b
    */
    private BigInteger modularAdd(BigInteger a,
        BigInteger b){
        BigInteger c = a.add(b);
        if(c.bitLength() > 192)
            c = c.subtract(p192);
        if(c.compareTo(p192) == 1)
            c = c.subtract(p192);
        return c;
    }

    /**
     * Subtraction modulo p
     * @param a
     * @param b
    */
    private BigInteger modularSub(BigInteger a,
        BigInteger b){
        BigInteger c = a.subtract(b);
        if(c.signum() ==-1)
            c = c.add(p192);
        return c;
    }
}
```

C.1.3 Implementation of $\mathbb{F}_{p_{224}}$

```
import java.math.BigInteger;

public final class P224Impl implements IFieldElement{
    private BigInteger n;
    private static final BigInteger p224 =
        new BigInteger("26959946667150639794667015O
            87019630673557916260026308143510066298881");
    /**
     * Constructor
     * @param a
    */
    public P224Impl(BigInteger a){
        n = a;
    }

    /**
     * Adddition
     * @param val
    */
    public P224Impl add(IFieldElement val){
        BigInteger c =
            modularAdd(n,val.toBigInteger());
        return new P224Impl(c);
    }

    /**
     * Compare
     * @param val
    */
    public int compareTo(IFieldElement val){
        return n.compareTo(val.toBigInteger());
    }

    /**
     * Equality testing
     * @param pObj
    */
    public boolean equals(java.lang.Object pObj){
        return n.equals(pObj);
    }

    /**
     * Inversion
    */
    public P224Impl inv(){
        return new P224Impl(n.modInverse(p224));
    }
```

```

    /**
     * Multiplication by integer
     * @param m
     */
    public P224Impl mul(int m){
        return new P224Impl(n.multiply(BigInteger.
            valueOf(m)).mod(p224));
    }

    /**
     * Multiplication
     * @param val
     */
    public P224Impl mul(IFieldElement val){
        return new P224Impl(n.multiply(val.toBigInteger
            ()).mod(p224));
    }

    /**
     * Negation
     */
    public P224Impl negate(){
        return new P224Impl(n.negate().mod(p224));
    }

    /**
     * Exponentiation
     * @param exp
     */
    public P224Impl pow(int exp){
        return new P224Impl(n.pow(exp).mod(p224));
    }

    /**
     * Multiplication by power of two
     * @param value
     */
    public P224Impl shl(int value){
        return new P224Impl(n.shiftLeft(value).mod(p224)
            );
    }

    /**
     * Division by power of two
     * @param value
     */
    public P224Impl shr(int value){
        return new P224Impl(n.shiftRight(value).mod(p224
            ));
    }

    /**
     * Squaring
     */
    public P224Impl sqr(){
        return new P224Impl(n.pow(2).mod(p224));
    }

    /**
     * Subtraction
     * @param val
     */
    public P224Impl sub(IFieldElement val){
        BigInteger c = modularSub(n,val.toBigInteger());
        return new P224Impl(c);
    }

    /**
     * Conversion to BigInteger
     */
    public BigInteger toBigInteger(){
        return n;
    }

    /**
     * Conversion to String
     */
    public String toString(){
        return n.toString();
    }

    /**
     * Addition modulo p
     * @param a
     * @param b
```

```
    */
    private BigInteger modularAdd(BigInteger a,
        BigInteger b){
        BigInteger c = a.add(b);
        if(c.bitLength() > 224)
            c = c.subtract(p224);
        if(c.compareTo(p224) == 1)
            c = c.subtract(p224);
        return c;
    }

    /**
     * Subtraction modulo p
     * @param a
     * @param b
    */
    private BigInteger modularSub(BigInteger a,
        BigInteger b){
        BigInteger c = a.subtract(b);
        if(c.signum() ==-1)
            c = c.add(p224);
        return c;
    }

}
```

C.1.4 Implementation of $\mathbb{F}_{p_{256}}$

```
import java.math.BigInteger;

public final class P256Impl implements IFieldElement{
    private BigInteger n;
    private static final BigInteger p256 =
        new BigInteger("115792089210356248762697446949
            40757353008614341529031419553363130886709785
            3951");

    /**
     * Constructor
     * @param a
    */
    public P256Impl(BigInteger a){
        n = a;
    }

    /**
     * Adddition
     * @param val
    */
    public P256Impl add(IFieldElement val){
        BigInteger c =
            modularAdd(n,val.toBigInteger());
        return new P256Impl(c);
    }

    /**
     * Compare
     * @param val
    */
    public int compareTo(IFieldElement val){
        return n.compareTo(val.toBigInteger());
    }

    /**
     * Equality testing
     * @param pObj
    */
    public boolean equals(java.lang.Object pObj){
        return n.equals(pObj);
    }

    /**
     * Inversion
    */
    public P256Impl inv(){
        return new P256Impl(n.modInverse(p256));
    }

    /**
     * Multiplication by integer
     * @param m
    */
    public P256Impl mul(int m){
        return new P256Impl(n.multiply(BigInteger.
            valueOf(m)).mod(p256));
```

```java
    }

    /**
     * Multiplication
     * @param val
     */
    public P256Impl mul(IFieldElement val){
        return new P256Impl(n.multiply(val.toBigInteger()).mod(p256));
    }

    /**
     * Negation
     */
    public P256Impl negate(){
        return new P256Impl(n.negate().mod(p256));
    }

    /**
     * Exponentiation
     * @param exp
     */
    public P256Impl pow(int exp){
        return new P256Impl(n.pow(exp).mod(p256));
    }

    /**
     * Multiplication by power of two
     * @param value
     */
    public P256Impl shl(int value){
        return new P256Impl(n.shiftLeft(value).mod(p256));
    }

    /**
     * Division by power of two
     * @param value
     */
    public P256Impl shr(int value){
        return new P256Impl(n.shiftRight(value).mod(p256));
    }

    /**
     * Squaring
     */
    public P256Impl sqr(){
        return new P256Impl(n.pow(2).mod(p256));
    }

    /**
     * Subtraction
     * @param val
     */
    public P256Impl sub(IFieldElement val){
        BigInteger c = modularSub(n, val.toBigInteger());
        return new P256Impl(c);
    }

    /**
     * Conversion to BigInteger
     */
    public BigInteger toBigInteger(){
        return n;
    }

    /**
     * Conversion to String
     */
    public String toString(){
        return n.toString();
    }

    /**
     * Addition modulo p
     * @param a
     * @param b
     */
    private BigInteger modularAdd(BigInteger a, BigInteger b){
        BigInteger c = a.add(b);
        if(c.bitLength() > 256)
            c = c.subtract(p256);
        if(c.compareTo(p256) == 1)
            c = c.subtract(p256);
```

```
        return c;
    }

    /**
     * Subtraction modulo p
     * @param a
     * @param b
     */
    private BigInteger modularSub(BigInteger a,
        BigInteger b){
        BigInteger c = a.subtract(b);
        if(c.signum() ==-1)
            c = c.add(p256);
        return c;
    }

}
```

C.1.5 Implementation of $\mathbb{F}_{p_{384}}$

```
import java.math.BigInteger;

public final class P384Impl implements IFieldElement{
    private BigInteger n;
    private static final BigInteger p384 =
        new BigInteger("394020061963944792122790
                401001436138050797392704654466679482934O
                4245721771496870329047266088258938001861
                606973112319");

    /**
     * Constructor
     * @param a
     */
    public P384Impl(BigInteger a){
        n = a;
    }

    /**
     * Adddition
     * @param val
     */
    public P384Impl add(IFieldElement val){
        BigInteger c =
            modularAdd(n,val.toBigInteger());
        return new P384Impl(c);
    }

    /**
     * Compare
     * @param val
     */
    public int compareTo(IFieldElement val){
        return n.compareTo(val.toBigInteger());
    }

    /**
     * Equality testing
     * @param pObj
     */
    public boolean equals(java.lang.Object pObj){
        return n.equals(pObj);
    }

    /**
     * Inversion
     */
    public P384Impl inv(){
        return new P384Impl(n.modInverse(p384));
    }

    /**
     * Multiplication by integer
     * @param m
     */
    public P384Impl mul(int m){
        return new P384Impl(n.multiply(BigInteger.
            valueOf(m)).mod(p384));
    }

    /**
     * Multiplication
     * @param val
     */
    public P384Impl mul(IFieldElement val){
```

```
        return new P384Impl(n.multiply(val.toBigInteger()).mod(p384));
    }

    /**
     * Negation
     */
    public P384Impl negate(){
        return new P384Impl(n.negate().mod(p384));
    }

    /**
     * Exponentiation
     * @param exp
     */
    public P384Impl pow(int exp){
        return new P384Impl(n.pow(exp).mod(p384));
    }

    /**
     * Multiplication by power of two
     * @param value
     */
    public P384Impl shl(int value){
        return new P384Impl(n.shiftLeft(value).mod(p384));
    }

    /**
     * Division by power of two
     * @param value
     */
    public P384Impl shr(int value){
        return new P384Impl(n.shiftRight(value).mod(p384));
    }

    /**
     * Squaring
     */
    public P384Impl sqr(){
        return new P384Impl(n.pow(2).mod(p384));
    }

    /**
     * Subtraction
     * @param val
     */
    public P384Impl sub(IFieldElement val){
        BigInteger c = modularSub(n, val.toBigInteger());
        return new P384Impl(c);
    }

    /**
     * Conversion to BigInteger
     */
    public BigInteger toBigInteger(){
        return n;
    }

    /**
     * Conversion to String
     */
    public String toString(){
        return n.toString();
    }

    /**
     * Addition modulo p
     * @param a
     * @param b
     */
    private BigInteger modularAdd(BigInteger a,
        BigInteger b){
        BigInteger c = a.add(b);
        if(c.bitLength() > 384)
            c = c.subtract(p384);
        if(c.compareTo(p384) == 1)
            c = c.subtract(p384);
        return c;
    }

    /**
     * Subtraction modulo p
     * @param a
     * @param b
```

```
    */
    private BigInteger modularSub(BigInteger a,
        BigInteger b){
        BigInteger c = a.subtract(b);
        if(c.signum() ==-1)
            c = c.add(p384);
        return c;
    }

}
```

C.1.6 Implementation of $\mathbb{F}_{p_{521}}$

```
import java.math.BigInteger;

public final class P521Impl implements IFieldElement{
    private BigInteger n;
    private static final BigInteger p521 =
        new BigInteger("686479766013060971498190
                0799081393217269435300143305409394463459
                1855431833976560521225596406614545549772
                9631139148085803712198799971664381257402
                8291115057151");

    /**
     * Constructor
     * @param a
    */
    public P521Impl(BigInteger a){
        n = a;
    }

    /**
     * Adddition
     * @param val
    */
    public P521Impl add(IFieldElement val){
        BigInteger c =
            modularAdd(n,val.toBigInteger());
        return new P521Impl(c);
    }

    /**
     * Compare
     * @param val
    */
    public int compareTo(IFieldElement val){
        return n.compareTo(val.toBigInteger());
    }

    /**
     * Equality testing
     * @param pObj
    */
    public boolean equals(java.lang.Object pObj){
        return n.equals(pObj);
    }

    /**
     * Inversion
    */
    public P521Impl inv(){
        return new P521Impl(n.modInverse(p521));
    }

    /**
     * Multiplication by integer
     * @param m
    */
    public P521Impl mul(int m){
        return new P521Impl(n.multiply(BigInteger.
            valueOf(m)).mod(p521));
    }

    /**
     * Multiplication
     * @param val
    */
    public P521Impl mul(IFieldElement val){
        return new P521Impl(n.multiply(val.toBigInteger
            ()).mod(p521));
    }

    /**
     * Negation
```

```
    */
    public P521Impl negate(){
        return new P521Impl(n.negate().mod(p521));
    }

    /**
     * Exponentiation
     * @param exp
    */
    public P521Impl pow(int exp){
        return new P521Impl(n.pow(exp).mod(p521));
    }

    /**
     * Multiplication by power of two
     * @param value
    */
    public P521Impl shl(int value){
        return new P521Impl(n.shiftLeft(value).mod(p521)
            );
    }

    /**
     * Division by power of two
     * @param value
    */
    public P521Impl shr(int value){
        return new P521Impl(n.shiftRight(value).mod(p521
            ));
    }

    /**
     * Squaring
    */
    public P521Impl sqr(){
        return new P521Impl(n.pow(2).mod(p521));
    }

    /**
     * Subtraction
     * @param val
    */
    public P521Impl sub(IFieldElement val){
        BigInteger c =
            modularSub(n,val.toBigInteger());
        return new P521Impl(c);
    }

    /**
     * Conversion to BigInteger
    */
    public BigInteger toBigInteger(){
        return n;
    }

    /**
     * Conversion to String
    */
    public String toString(){
        return n.toString();
    }

    /**
     * Addition modulo p
     * @param a
     * @param b
    */
    private BigInteger modularAdd(BigInteger a,
        BigInteger b){
        BigInteger c = a.add(b);
        if(c.bitLength() > 521)
            c = c.subtract(p521);
        if(c.compareTo(p521) == 1)
            c = c.subtract(p521);
        return c;
    }

    /**
     * Subtraction modulo p
     * @param a
     * @param b
    */
    private BigInteger modularSub(BigInteger a,
        BigInteger b){
        BigInteger c = a.subtract(b);
        if(c.signum() ==-1)
```

```
            c = c.add(p521);
        return c;
    }
}
```

C.2 Addition and Doubling

```
import java.math.BigInteger;
import java.lang.Math;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.Map;


public final class Addition{

    /*********************************************
     * Addition of distinct points
     *********************************************/

    /**
     * Add two points in Chudnovsky Jacobian coordinates
     * and express the result in Chudnovsky Jacobian
     * coordinates.
     * @param x1
     * @param y1
     * @param z1
     * @param z1_2
     * @param z1_3
     * @param x2
     * @param y2
     * @param z2
     * @param z2_2
     * @param z2_3
     * @param pq
     * @throws IllegalArgumentException
     */

    static void addPointsJC (IFieldElement x1,
                             IFieldElement y1,
                             IFieldElement z1,
                             IFieldElement z1_2,
                             IFieldElement z1_3,
                             IFieldElement x2,
                             IFieldElement y2,
                             IFieldElement z2,
                             IFieldElement z2_2,
                             IFieldElement z2_3,
                             IFieldElement[] pq)
        throws IllegalArgumentException {

        if (z1.equals(BigInteger.ZERO)) { // P == O?
            pq[0] = x2;
            pq[1] = y2;
            pq[2] = z2;
            pq[3] = z2_2;
            pq[4] = z2_3;
            return;
        }

        if(z2.equals(BigInteger.ZERO)) // Q==0?
            {
                pq[0] = x1;
                pq[1] = y1;
                pq[2] = z1;
                pq[3] = z1_2;
                pq[4] = z1_3;
                return;
            }

        //Temporary variables
        IFieldElement t1,t2,t3,t4,t5,t6,t7;

        t1 = x1.mul(z2_2); //A
        t3 = y1.mul(z2_3); //C

        t2 = x2.mul(z1_2); //B
        t4 = y2.mul(z1_3); //D

        t5 = t2.sub(t1); //E
        t2 = t4.sub(t3); //F

        if(t5.equals(BigInteger.ZERO) &&
           t2.equals(BigInteger.ZERO)) //P=Q?
```

```
            //Should use double instead
            throw new IllegalArgumentException();

        t4 = t5.sqr(); //E^2
        t6 = t4.mul(t5); // E^3
        t4 = t1.mul(t4); // AE^2
        t7 = t2.sqr(); // F^2
        t1 = t6.negate().sub(t4.shl(1)).add(t7); // x3
        t2 = t2.mul(t4.sub(t1));
        t2 = t2.sub(t3.mul(t6)); // y3
        t3 = t5;
        t3 = t3.mul(z1);
        t3 = t3.mul(z2); //z3
        t4 = t3.sqr(); // z3_2
        t5 = t4.mul(t3); // z3_3

        //Return values
        pq[0] = t1;
        pq[1] = t2;
        pq[2] = t3;
        pq[3] = t4;
        pq[4] = t5;
    }

    /**
     * Add two points in Chudnovsky Jacobian / Jacobian
     * coordinates and express the result in Jacobian
     * coordinates.
     * @param x1
     * @param y1
     * @param z1
     * @param z1_2
     * @param z1_3
     * @param x2
     * @param y2
     * @param z2
     * @param pq
     * @throws IllegalArgumentException
     */

    static void addPointsJCJtoJ (IFieldElement x1,
                                 IFieldElement y1,
                                 IFieldElement z1,
                                 IFieldElement z1_2,
                                 IFieldElement z1_3,
                                 IFieldElement x2,
                                 IFieldElement y2,
                                 IFieldElement z2,
                                 IFieldElement[] pq)
        throws IllegalArgumentException {

        if (z1.equals(BigInteger.ZERO)){ // P == O?
            pq[0] = x2;
            pq[1] = y2;
            pq[2] = z2;
            return;
        }

        if(z2.equals(BigInteger.ZERO)){ // Q==0?
            pq[0] = x1;
            pq[1] = y1;
            pq[2] = z1;
            return;
        }

        //Temporary variables
        IFieldElement t1,t2,t3,t4,t5,t6,t7;

        t5 = z2.sqr();
        t1 = x1.mul(t5); // A
        t5 = t5.mul(z2);
        t3 = y1.mul(t5); // C

        t2 = x2.mul(z1_2); // B
        t4 = y2.mul(z1_3); // D

        t5 = t2.sub(t1); // E
        t2 = t4.sub(t3); // F

        if(t5.equals(BigInteger.ZERO) &&
          t2.equals(BigInteger.ZERO)) //P=Q?
            //Should use double instead
            throw new IllegalArgumentException();

        t4 = t5.sqr(); // E^2
        t6 = t4.mul(t5); // E^3
```

```java
        t4 = t1.mul(t4); // AE^2
        t7 = t2.sqr(); // F^2
        t1 = t7.sub(t6).sub(t4.shl(1)); // x3
        t2 = t2.mul(t4.sub(t1));
        t2 = t2.sub(t3.mul(t6)); // y3
        t5 = t5.mul(z1);
        t5 = t5.mul(z2); //z3

        //Return values
        pq[0] = t1;
        pq[1] = t2;
        pq[2] = t5;
    }


    /**
     * Add two points in Chudnovsky Jacobian / Jacobian
     * coordinates and express the result in modified
     * Jacobian coordinates.
     * @param x1
     * @param y1
     * @param z1
     * @param z1_2
     * @param z1_3
     * @param x2
     * @param y2
     * @param z2
     * @param a
     * @param pq
     * @throws IllegalArgumentException
     */

    static void addPointsJCJtoJM (IFieldElement x1,
                                  IFieldElement y1,
                                  IFieldElement z1,
                                  IFieldElement z1_2,
                                  IFieldElement z1_3,
                                  IFieldElement x2,
                                  IFieldElement y2,
                                  IFieldElement z2,
                                  IFieldElement a,
                                  IFieldElement[] pq)
        throws IllegalArgumentException {

        if (z1.equals(BigInteger.ZERO)) { // P == O?
            pq[0] = x2;
            pq[1] = y2;
            pq[2] = z2;
            pq[3] = pq[2].sqr();
            pq[3] = pq[3].sqr();
            pq[3] = pq[3].mul(a);
            return;
        }

        if(z2.equals(BigInteger.ZERO)){// Q==0?
            pq[0] = x1;
            pq[1] = y1;
            pq[2] = z1;
            pq[3] = pq[2].sqr();
            pq[3] = pq[3].sqr();
            pq[3] = pq[3].mul(a);
            return;
        }

        //Temporary variables
        IFieldElement t1,t2,t3,t4,t5,t6,t7;

        t5 = z2.sqr();
        t1 = x1.mul(t5); // A
        t5 = t5.mul(z2);
        t3 = y1.mul(t5); // C

        t2 = x2.mul(z1_2); // B
        t4 = y2.mul(z1_3); // D

        t5 = t2.sub(t1); // E
        t2 = t4.sub(t3); // F
        if(t5.equals(BigInteger.ZERO) &&
           t2.equals(BigInteger.ZERO)) //P=Q?
            //Should use double instead
            throw new IllegalArgumentException();

        t4 = t5.sqr(); // E^2
        t6 = t4.mul(t5); // E^3
        t4 = t1.mul(t4); // AE^2
        t7 = t2.sqr(); // F^2
```

```java
        t1 = t7.sub(t6).sub(t4.shl(1)); // x3
        t2 = t2.mul(t4.sub(t1));
        t2 = t2.sub(t3.mul(t6)); // y3
        t3 = t5.mul(z1);
        t3 = t3.mul(z2); //z3

        t4 = t3.sqr(); // z3^2
        t4 = t4.sqr(); // z3^4
        t4 = t4.mul(a); // az3^4

        //Return values
        pq[0] = t1;
        pq[1] = t2;
        pq[2] = t3;
        pq[3] = t4;
    }

    /**
     * Add two points in Affine / Jacobian coordinates
     * and express the result in Jacobian coordinates.
     * @param x1
     * @param y1
     * @param x2
     * @param y2
     * @param z2
     * @param pq
     * @throws IllegalArgumentException
     */

    static void addPointsAtoJ (IFieldElement x1,
                               IFieldElement y1,
                               IFieldElement x2,
                               IFieldElement y2,
                               IFieldElement[] pq,
                               IFieldElement one)
        throws IllegalArgumentException {
        if (x1==null){ // P == O?
            pq[0] = x2;
            pq[1] = y2;
            pq[2] = one;
            return;
        }

        if(x2==null){ // Q==0?
            pq[0] = x1;
            pq[1] = y1;
            pq[2] = one;
            return;
        }

        IFieldElement t1,t2,t3,t4,t5,t6,t7;

        t5 = x2.sub(x1); // E
        t2 = y2.sub(y1); // F
        if(t5.equals(BigInteger.ZERO) &&
           t2.equals(BigInteger.ZERO)) //P=Q?
            //Should use double instead
            throw new IllegalArgumentException();

        t4 = t5.sqr(); // E^2
        t6 = t4.mul(t5); // E^3
        t4 = x1.mul(t4); // AE^2
        t7 = t2.sqr(); // F^2
        t1 = t7.sub(t6).sub(t4.shl(1)); // x3
        t2 = t2.mul(t4.sub(t1));
        t2 = t2.sub(y1.mul(t6)); // y3

        t3 = t5; // z3

        //Return values
        pq[0] = t1;
        pq[1] = t2;
        pq[2] = t3;

    }

    /**
     * Add two points in Affine / Jacobian coordinates
     * and express the result in Jacobian coordinates.
     * @param x1
     * @param y1
     * @param x2
     * @param y2
     * @param z2
     * @param pq
     * @throws IllegalArgumentException
```

```java
     */

    static void addPointsAJtoJ (IFieldElement x1,
                                IFieldElement y1,
                                IFieldElement x2,
                                IFieldElement y2,
                                IFieldElement z2,
                                IFieldElement[] pq,
                                IFieldElement one)
        throws IllegalArgumentException {

        if (x1==null) { // P == O?
            pq[0] = x2;
            pq[1] = y2;
            pq[2] = z2;
            return;
        }

        if(z2.equals(BigInteger.ZERO)){ // Q==0?
            pq[0] = x1;
            pq[1] = y1;
            pq[2] = one;
            return;
        }

        IFieldElement t1,t2,t3,t4,t5,t6,t7;

        t2 = x2; // B
        t4 = y2; // D

        t5 = z2.sqr();
        t1 = x1.mul(t5); // A
        t5 = t5.mul(z2);
        t3 = y1.mul(t5); // C

        t5 = t2.sub(t1); // E
        t2 = t4.sub(t3); // F
        if(t5.equals(BigInteger.ZERO) &&
           t2.equals(BigInteger.ZERO)) //P=Q?
            //Should use double instead
            throw new IllegalArgumentException();

        t4 = t5.sqr(); // E^2
        t6 = t4.mul(t5); // E^3
        t4 = t1.mul(t4); // AE^2
        t7 = t2.sqr(); // F^2
        t1 = t7.sub(t6).sub(t4.shl(1)); // x3
        t2 = t2.mul(t4.sub(t1));
        t2 = t2.sub(t3.mul(t6)); // y3
        t3 = t5.mul(z2); // z3

        //Return values
        pq[0] = t1;
        pq[1] = t2;
        pq[2] = t3;

    }

    /**
     * Add two points in Affine / Jacobian coordinates
     * express the result in modified Jacobian
     * coordinates.
     * @param x1
     * @param y1
     * @param x2
     * @param y2
     * @param z2
     * @param a
     * @param pq
     * @param one
     * @throws IllegalArgumentException
     */

    static void addPointsAJtoJM (IFieldElement x1,
                                 IFieldElement y1,
                                 IFieldElement x2,
                                 IFieldElement y2,
                                 IFieldElement z2,
                                 IFieldElement a,
                                 IFieldElement[] pq,
                                 IFieldElement one)
        throws IllegalArgumentException {

        if (x1==null) { // P == O?
            pq[0] = x2;
            pq[1] = y2;
```

```
            pq[2] = z2;
            pq[3] = pq[2].sqr();
            pq[3] = pq[3].sqr();
            pq[3] = pq[3].mul(a);
            return;
        }

        if(z2.equals(BigInteger.ZERO)){ // Q==0?
            pq[0] = x1;
            pq[1] = y1;
            pq[2] = one;
            pq[3] = a;
            return;
        }

        IFieldElement t1,t2,t3,t4,t5,t6,t7;

        t2 = x2; // B
        t4 = y2; // D

        t5 = z2.sqr();
        t1 = x1.mul(t5); // A
        t5 = t5.mul(z2);
        t3 = y1.mul(t5); // C

        t5 = t2.sub(t1); // E
        t2 = t4.sub(t3); // F
        if(t5.equals(BigInteger.ZERO) &&
           t2.equals(BigInteger.ZERO)) //P=Q?
            //Should use double instead
            throw new IllegalArgumentException();

        t4 = t5.sqr(); // E^2
        t6 = t4.mul(t5); // E^3
        t4 = t1.mul(t4); // AE^2
        t7 = t2.sqr(); // F^2
        t1 = t7.sub(t6).sub(t4.shl(1)); // x3
        t2 = t2.mul(t4.sub(t1));
        t2 = t2.sub(t3.mul(t6)); // y3
        t3 = t5.mul(z2); //z3

        t4 = t3.sqr(); // z3^2
        t4 = t4.sqr(); // z3^4
        t4 = t4.mul(a); // az3^4

        //Return values
        pq[0] = t1;
        pq[1] = t2;
        pq[2] = t3;
        pq[3] = t4;

    }

    /**
     * Add two points in Affine  coordinates and
     * express the result in modified Jacobian
     * coordinates.
     * @param x1
     * @param y1
     * @param x2
     * @param y2
     * @param z2
     * @param a
     * @param pq
     * @param one
     * @throws IllegalArgumentException
     */

    static void addPointsAtoJM (IFieldElement x1,
                                IFieldElement y1,
                                IFieldElement x2,
                                IFieldElement y2,
                                IFieldElement a,
                                IFieldElement[] pq,
                                IFieldElement one)
        throws IllegalArgumentException {

        if (x1==null) { // P == O?
            pq[0] = x2;
            pq[1] = y2;
            pq[2] = one;
            pq[3] = a;
            return;
        }

        if(x2 == null){ // Q==0?
```

```java
            pq[0] = x1;
            pq[1] = y1;
            pq[2] = one;
            pq[3] = a;
            return;
        }

        IFieldElement t1,t2,t3,t4,t5,t6,t7;

        t5 = x2.sub(x1); // E
        t2 = y2.sub(y1); // F
        if(t5.equals(BigInteger.ZERO) &&
           t2.equals(BigInteger.ZERO)) //P=Q?
            //Should use double instead
            throw new IllegalArgumentException();

        t4 = t5.sqr(); // E^2
        t6 = t4.mul(t5); // E^3
        t4 = x1.mul(t4); // AE^2
        t7 = t2.sqr(); // F^2
        t1 = t7.sub(t6).sub(t4.shl(1)); // x3
        t2 = t2.mul(t4.sub(t1));
        t2 = t2.sub(y1.mul(t6)); // y3
        t3 = t5; //z3

        t4 = t3.sqr(); // z3^2
        t4 = t4.sqr(); // z3^4
        t4 = t4.mul(a); // az3^4

        //Return values
        pq[0] = t1;
        pq[1] = t2;
        pq[2] = t3;
        pq[3] = t4;

    }

    /**
     * Add two affine points..
     * @param x1
     * @param y1
     * @param x2
     * @param y2
     * @param pq
     * @throws IllegalArgumentException
     */

    static void addPointsA (IFieldElement x1,
                            IFieldElement y1,
                            IFieldElement x2,
                            IFieldElement y2,
                            IFieldElement[] pq)
        throws IllegalArgumentException {
        if (x1 == null) { // P == O?
            pq[0] = x2; pq[1] = y2;
            return;
        }

        if(x2 == null){ // Q==0?
            pq[0] = x1;
            pq[1] = y1;
            return;
        }

        if (x1.equals(x2) &&
            (y1.equals(y2) || y1.equals(y2.negate())))
            //P = \pm Q?
            throw new IllegalArgumentException();

        IFieldElement d =
            (y2.sub(y1)).mul((x2.sub(x1)).inv());

        pq[0] = d.sqr().sub(x1).sub(x2);
        pq[1] = d.mul(x1.sub(pq[0])).sub(y1);
    }

    /**
     * Add two affine points without doing inversion.
     * @param x1
     * @param y1
     * @param x2
     * @param y2
     * @param d
     * @param pq
     * @throws IllegalArgumentException
     */
```

```java

    static void addPointsA_NoInversions(IFieldElement x1,
                                        IFieldElement y1,
                                        IFieldElement x2,
                                        IFieldElement y2,
                                        IFieldElement e,
                                        IFieldElement[] pq)
        throws IllegalArgumentException {

        if (x1 == null) { // P == O?
            pq[0] = x2; pq[1] = y2;
            return;
        }

        if(x2 == null){ // Q==0?
            pq[0] = x1;
            pq[1] = y1;
            return;
        }

        if (x1.equals(x2) &&
            (y1.equals(y2) || y1.equals(y2.negate())))
            throw new IllegalArgumentException();

        //The element e is the inverted one.
        IFieldElement d = (y2.sub(y1)).mul(e);

        pq[0] = d.sqr().sub(x1).sub(x2);
        pq[1] = d.mul(x1.sub(pq[0])).sub(y1);
    }

    /**
     * Adds two points in projective coordinates using
     * Montgomerys trick (in general form by Briet
     * and Joye). The algorithm assumes that the point
     * of difference is in affine coordinates.
     * @param x1
     * @param z1
     * @param x2
     * @param z2
     * @param x
     * @param a
     * @param b
     * @param pq
     * @throws IllegalArgumentException
     */

    static void addPointsMontgomeryP(IFieldElement x1,
                                     IFieldElement z1,
                                     IFieldElement x2,
                                     IFieldElement z2,
                                     IFieldElement x,
                                     IFieldElement a,
                                     IFieldElement b,
                                     IFieldElement[] pq)
        throws IllegalArgumentException {

        if(z1.equals(BigInteger.ZERO) ||
           z2.equals(BigInteger.ZERO)) //P=O or Q=O?
            throw new IllegalArgumentException();

        //Temporary values
        IFieldElement t1,t2,t3,t4,t5,t6;

        t1 = b.shl(2);  // 4b
        t1 = t1.negate(); //-4b
        t2 = z1.mul(z2); //z1z2
        t1 = t1.mul(t2); //-4bz1z2

        t3 = x1.mul(z2); //x1z2
        t4 = x2.mul(z1); //x2z1
        t5 = t3.add(t4); //x1z2 + x2z1
        t1 = t1.mul(t5); //-4bz1z2(x1z2 + x2z1)

        t5 = x1.mul(x2); //x1x2

        //t2 = a.mul(t2); //az1z2
        t2 = t2.negate();
        t6 = t2;
```

```java
        t2 = t2.shl(1);
        t2 = t2.add(t6); //-3z1z2

        t2 = t5.sub(t2); //x1x2-az1z2
        t2 = t2.sqr(); //(x1x2-az1z2)^2

        pq[0] = t1.add(t2); //x3

        t1 = t3.sub(t4); //x1z2 - x2z1

        if(t1.equals(BigInteger.ZERO)) //P= \pm Q?
            throw new IllegalArgumentException();

        t1 = t1.sqr(); //(x1z2 - x2z1)^2

        pq[1] = x.mul(t1); //z3

    }

    /**
     * Add two points in projective coordinates.
     * @param x1
     * @param y1
     * @param z1
     * @param x2
     * @param y2
     * @param z2
     * @param pq
     * @throws IllegalArgumentException
     *
     */
    static void addPointsP (IFieldElement x1,
                            IFieldElement y1,
                            IFieldElement z1,
                            IFieldElement x2,
                            IFieldElement y2,
                            IFieldElement z2,
                            IFieldElement[] pq)
        throws IllegalArgumentException {

        //Temporary variables
        IFieldElement t1,t2,t3,t4,t5,t6,t7,t8;

        t1 = y2.mul(z1); //y2z1
        t2 = y1.mul(z2); //y1z2
        t1 = t1.sub(t2); //A
        t3 = x2.mul(z1); //x2z1
        t4 = x1.mul(z2); //x1z2
        t3 = t3.sub(t4); //B
        if(t1.equals(BigInteger.ZERO) &&
           t3.equals(BigInteger.ZERO)) //P=Q?
            //Should use double instead
            throw new IllegalArgumentException();

        t5 = z1.mul(z2); //z1z2
        t6 = t1.sqr(); //A^2
        t6 = t6.mul(t5); //A^2z1z2
        t7 = t3.sqr(); //B^2
        t8 = t7.mul(t3); //B^3
        t6 = t6.sub(t8); //A^2z1z2-B^3
        t7 = t7.mul(t4); //B^2x1z2
        t6 = t6.sub(t7.shl(1)); //C
        t3 = t3.mul(t6); //X3
        t7 = t7.sub(t6); //B^2x1z2-C
        t7 = t1.mul(t7); //A(B^2x1z2-C)
        t4 = t7.sub(t8.mul(t2)); //Y3
        t5 = t8.mul(t5); //Z3

        pq[0] = t3;
        pq[1] = t4;
        pq[2] = t5;
    }



    /*********************************************
     * Doubling of a point
     ********************************************/

    /**
     * Double a point in modified Jacobian coordinates.
     * Express the result in Jacobian coordinates.
     * @param x1
     * @param y1
     * @param z1
     * @param az1_4
```

```
 * @param pp
 */

static void doublePointJMtoJ (IFieldElement x1,
                              IFieldElement y1,
                              IFieldElement z1,
                              IFieldElement az1_4,
                              IFieldElement[] pp) {

    //Temporary variables
    IFieldElement t1,t2,t3,t4,t5,t6;

    t1=y1.sqr();
    t2=x1.shl(2).mul(t1); //A
    t3=x1.sqr();
    t3=t3.add(t3).add(t3);
    t3=t3.add(az1_4); //B
    t1=t1.sqr().shl(3); //C
    t4=t3.sqr(); //B^2
    t4=t4.sub(t2.shl(1)); //x3
    t5=t3.mul(t2.sub(t4)).sub(t1); //y3
    t6=y1.shl(1).mul(z1); //z3

    //Return values
    pp[0] = t4;
    pp[1] = t5;
    pp[2] = t6;
}



/**
 * Double a point in modified Jacobian coordinates.
 * Express the result in modified Jacobian
     coordinates.
 * @param x1
 * @param y1
 * @param z1
 * @param az1_4
 * @param pp
 */

static void doublePointJM (IFieldElement x1,
                           IFieldElement y1,
                           IFieldElement z1,
                           IFieldElement az1_4,
                           IFieldElement[] pp){

    //Temporary variables
    IFieldElement t1,t2,t3,t4,t5,t6,t7;

    t1=y1.sqr();
    t2=x1.shl(2).mul(t1); //A
    t3=x1.sqr();
    t3=t3.add(t3).add(t3);
    t3=t3.add(az1_4); //B
    t1=t1.sqr().shl(3); //C
    t4=t3.sqr(); //B^2
    t4=t4.sub(t2.shl(1)); //x3
    t5=t3.mul(t2.sub(t4)).sub(t1); //y3
    t6=y1.shl(1).mul(z1); //z3
    t7=t1.shl(1).mul(az1_4); //az3^4

    //Return values
    pp[0] = t4;
    pp[1] = t5;
    pp[2] = t6;
    pp[3] = t7;
}

/**
 * Double a point in affine coordinates.
 * Express the result in modified Jacobian
     coordinates.
 * @param x1
 * @param y1
 * @param z1
 * @param az1_4
 * @param pp
 */

static void doublePointAtoJM (IFieldElement x1,
                              IFieldElement y1,
                              IFieldElement a,
                              IFieldElement[] pp) {

```

```
        //Temporary variables
        IFieldElement t1,t2,t3,t4,t5,t6,t7;

        t1=y1.sqr();
        t2=x1.shl(2).mul(t1); //A
        t3=x1.sqr();
        t3=t3.add(t3).add(t3);
        t3=t3.add(a); //B
        t1=t1.sqr().shl(3); //C
        t4=t3.sqr(); //B^2
        t4=t4.sub(t2.shl(1)); //x3
        t5=t3.mul(t2.sub(t4)).sub(t1); //y3
        t6=y1.shl(1); //z3
        t7=t1.shl(1).mul(a); //az3^4

        //Return values
        pp[0] = t4;
        pp[1] = t5;
        pp[2] = t6;
        pp[3] = t7;
    }

    /**
     * Double a point in affine coordinates. Express the
     * result in Chudnovsky Jacobian coordinates.
     * @param x1
     * @param y1
     * @param pp
     */
    static void doublePointAtoJC (IFieldElement x1,
                                  IFieldElement y1,
                                  IFieldElement zero,
                                  IFieldElement one,
                                  IFieldElement three,
                                  IFieldElement[] pp)
        throws IllegalArgumentException {

        if (x1 == null) { // P == O?
            pp[0] = one;
            pp[1] = one;
            pp[2] = zero;
            pp[3] = zero;
            pp[4] = zero;
            return;
        }

        //Temporary variables
        IFieldElement t1,t2,t3,t4,t5,t6;

        t1 = y1.sqr(); // y1^2
        t2 = x1.mul(t1);
        t2 = t2.shl(2); // A
        t3 = y1.shl(1); // z3
        t4 = t1.shl(2); // z3^2 = 4y1^2
        t5 = t4.shl(1);
        t5 = t5.mul(y1); // z3^3 = 8y1^3
        t6 = x1.sqr();
        t6 = t6.add(t6.add(t6)).sub(three); // B
        t1 = t6.sqr(); // B^2
        t1 = t1.sub(t2.shl(1)); // x3
        t2 = t6.mul(t2.sub(t1));
        t2 = t2.sub(t5.mul(y1)); // y3

        //Return values
        pp[0] = t1;
        pp[1] = t2;
        pp[2] = t3;
        pp[3] = t4;
        pp[4] = t5;
    }

    /**
     * Double a point in affine coordinates. Express the
     * result in Jacobian coordinates.
     * @param x1
     * @param y1
     * @param pp
     */

    static void doublePointAtoJ (IFieldElement x1,
                                 IFieldElement y1,
                                 IFieldElement zero,
                                 IFieldElement one,
                                 IFieldElement[] pp) {

        if (x1==null) { // P == O?
```

```
            pp[0] = one;
            pp[1] = one;
            pp[2] = zero;
            return;
        }

        //Temporary variables
        IFieldElement t1,t2,t3,t4,t5;

        t1 = y1.sqr(); // y1^2
        t2 = x1.shl(2);
        t2 = t2.mul(t1); // A

        t3 = x1.sub(one).mul(x1.add(one));
        t3 = t3.shl(1).add(t3); // B
        t4 = t3.sqr(); // B^2
        t5 = t1.shl(1);
        t5 = t5.sqr();
        t5 = t5.shl(1); // 8y1^4
        t1 = t4.sub(t2.shl(1)); // x3
        t2 = t3.mul(t2.sub(t1)).sub(t5); // y3
        t3 = y1.shl(1);

        //Return values
        pp[0] = t1;
        pp[1] = t2;
        pp[2] = t3;
    }

    /**
     * Double a point in Jacobian coordinates.
     * @param x1
     * @param y1
     * @param z1
     * @param pp
     */
    static void doublePointJ (IFieldElement x1,
                              IFieldElement y1,
                              IFieldElement z1,
                              IFieldElement zero,
                              IFieldElement one,
                              IFieldElement[] pp){

        if (z1.equals(BigInteger.ZERO)) { // P == O?
            pp[0] = one;
            pp[1] = one;
            pp[2] = zero;
            return;
        }

        //Temporary variables
        IFieldElement t1,t2,t3,t4,t5;

        t1 = y1.sqr(); // y1^2
        t2 = x1.shl(2);
        t2 = t2.mul(t1); // A
        t3 = z1.sqr();
        t3 = x1.sub(t3).mul(x1.add(t3));
        t3 = t3.shl(1).add(t3); // B
        t4 = t3.sqr(); // B^2
        t5 = t1.shl(1);
        t5 = t5.sqr();
        t5 = t5.shl(1); // 8y1^4
        t1 = t4.sub(t2.shl(1)); // x3
        t2 = t3.mul(t2.sub(t1)).sub(t5); // y3
        t3 = y1.shl(1);
        t3 = t3.mul(z1); // z3

        //Return values
        pp[0] = t1;
        pp[1] = t2;
        pp[2] = t3;
    }

    /**
     * Double a point in affine coordinates.
     * @param x1
     * @param y1
     * @param a
     * @param pp
     * @throws IllegarArgumentException
     */

    static void doublePointA (IFieldElement x1,
                              IFieldElement y1,
                              IFieldElement a,
```

```
                                    IFieldElement[] pp){

        if (x1 == null) { // P == O?
            pp[0] = pp[1] = null;
            return;
        }

        if (y1.equals(y1.negate()))
            throw new IllegalArgumentException();

        //Temporary variables
        IFieldElement t1,t2,t3,t4,t5,t6;

        t1 = x1.sqr();
        t1 = t1.shl(1).add(t1);
        t1 = t1.add(a);

        t2 = y1.shl(1);
        t2 = t2.inv();

        t3 = t1.mul(t2);

        t4 = t3.sqr();
        t4 = t4.sub(x1.shl(1));

        t5 = x1.sub(t4);
        t6 = t3.mul(t5);
        t6 = t6.sub(y1);

        pp[0] = t4;
        pp[1] = t6;
    }

    /**
     * Double a point in affine coordinates
     * with no iversions.
     * @param x1
     * @param y1
     * @param a
     * @param d
     * @param pp
     * @throws IllegalArgumentException
     */

    static void doublePointA_NoInversions (IFieldElement
        x1,
                                           IFieldElement
                                               y1,
                                           IFieldElement
                                               a,
                                           IFieldElement
                                               d,
                                           IFieldElement
                                               [] pp)
        throws IllegalArgumentException {

        if (x1 == null) { // P == O?
            pp[0] = pp[1] = null;
            return;
        }

        if (y1.equals(y1.negate()))
            throw new IllegalArgumentException();

        IFieldElement t1,t2,t3,t4,t5,t6;

        t1 = x1.sqr();
        t1 = t1.shl(1).add(t1);
        t1 = t1.add(a);
        t2 = d;
        t3 = t1.mul(t2);

        t4 = t3.sqr();
        t4 = t4.sub(x1.shl(1));

        t5 = x1.sub(t4);
        t6 = t3.mul(t5);
        t6 = t6.sub(y1);

        pp[0] = t4;
        pp[1] = t6;
    }


    /**
     * Double a point in projective coordinates using
```

```
 * Montgomerys trick (in general form by
 * Briet and Joye).
 * @param x1
 * @param z1
 * @param a
 * @param b
 * @param pp
 */
static void doublePointMontgomeryP (IFieldElement x1
    ,
                                    IFieldElement z1
                                        ,
                                    IFieldElement a,
                                    IFieldElement b,
                                    IFieldElement[]
                                        pp)
    throws IllegalArgumentException
{
    //Temporary values
    IFieldElement t1,t2,t3,t4,t5;

    t1 = x1.sqr(); //x1^2
    t2 = z1.sqr(); //z1^2
    t3 = z1.mul(t2); //z1^3
    t3 = b.mul(t3); //bz1^3
    //t2 = t2.mul(a); //az1^2
    t2 = t2.negate();
    t4 = t2;
    t2 = t2.shl(1);
    t2 = t2.add(t4); //az1^2

    t4 = t1.sub(t2); //x1^2-az1^2
    t4 = t4.sqr(); //(x1^2-az1^2)^2
    t5 = x1.mul(t3);// x1bz1^3
    t5 = t5.shl(3); // 8x1bz1^3

    pp[0] = t4.sub(t5); //x3

    t4 = t1.add(t2); //x1^2+az1^2
    t4 = t4.mul(x1); //x1(x1^2+az1^2)
    t4 = t4.add(t3); //x1(x1^2+az1^2)+bz1^3
    t4 = t4.mul(z1); //z1(x1(x1^2+az1^2)+bz1^3);

    pp[1] = t4.shl(2); //z3
}

/**
 * Double a point in affine coordinates using
 * Montgomerys trick and give the result in
 * projective coordinates (in general form
 * by Briet and Joye).
 * @param x1
 * @param z1
 * @param a
 * @param b
 * @param pp
 */

static void doublePointMontgomeryAtoP (IFieldElement
    x1,
                                       IFieldElement
                                           a,
                                       IFieldElement
                                           b,
                                       IFieldElement
                                           [] pp){

    //Temporary values
    IFieldElement t1,t2,t3,t4,t5;

    t1 = x1.sqr(); //x1^2
    t2 = t1.sub(a).sqr(); //(x1^2-a)^2
    t3 = b.mul(x1); //bx1
    t3 = t3.shl(3); //8bx1
    pp[0] = t2.sub(t3); //(x1^2-a)^2-8bx1

    t1 = t1.mul(x1); //x1^3
    t2 = x1.negate();
    t3 = t2;
    t2 = t2.shl(1);
    t2 = t2.add(t3); //-3x1
    t1 = t1.add(t2).add(b);
    pp[1] = t1.shl(2);
}

/**
```

```
 * Double a point in affine coordinates to
     projective coordinates.
 * @param x1
 * @param y1
 * @param pp
 */

static void doublePointAtoP (IFieldElement x1,
                             IFieldElement y1,
                             IFieldElement three,
                             IFieldElement[] pp){
    //Temporary variables
    IFieldElement t1,t2,t3,t4;

    t1 = x1.sqr(); //x1^2
    t1 = t1.shl(1).add(t1); //3x1^2
    t1 = t1.sub(three); //A
    t2 = y1.sqr(); //y1^2
    t3 = x1.mul(t2); //C
    t4 = t1.sqr(); //A^2
    t3 = t3.shl(2); //4C
    t4 = t4.sub(t3.shl(1)); //D
    t3 = t3.sub(t4); //4C-D
    t3 = t1.mul(t3); //A(4C-D)
    t3 = t3.sub(t2.sqr().shl(3)); //Y3
    t1 = t2.shl(3).mul(y1); //Z3
    t2 = y1.shl(1).mul(t4); //X3;

    pp[0] = t2;
    pp[1] = t3;
    pp[2] = t1;

}

/*********************************************
 * Unified addition
 *********************************************/


/**
 * Adds two points in projective coordinates using
 * the unified addition formula (in general form
 * by Briet and Joye).
 * @param x1
 * @param y1
 * @param z1
 * @param x2
 * @param y2
 * @param z2
 * @param pq
 * @throws IllegalArgumentException
 */

static void addPointsUnifP (IFieldElement x1,
                            IFieldElement y1,
                            IFieldElement z1,
                            IFieldElement x2,
                            IFieldElement y2,
                            IFieldElement z2,
                            IFieldElement[] pq)
    throws IllegalArgumentException{

    if(y1.equals(y2.negate())) //P==-Q?
        throw new IllegalArgumentException();

    //Temporary values
    IFieldElement t1,t2,t3,t4,t5,t6;

    t1 = z1.mul(z2) ; //A
    t2 = x1.mul(z2); //B
    t3 = x2.mul(z1); //C
    t4 = y1.mul(z2); //D
    t5 = y2.mul(z1); //E
    t6 = t2.add(t3); //F
    t4 = t4.add(t5); //G
    t5 = t6.sqr(); //F^2
    t5 = t5.sub(t2.mul(t3)); //F^2-BC

    t2 = t1.sqr(); //A^2
    t2 = t2.negate(); //-A^2
    t3 = t2.shl(1); //-2A^2
    t2 = t3.add(t2); //-3A^2
    t5 = t5.add(t2); //H

    t2 = t1.mul(t4); //J
    t1 = t2.mul(t4); //K
```

```
        t3 = t6.mul(t1); //L
        t4 = t5.sqr().sub(t3); //M
        t3 = t3.sub(t4.shl(1)); //L-2M
        t5 = t5.mul(t3); //H(L-2M)
        t5 = t5.sub(t1.sqr()); //Y3
        t4 = t4.shl(1); //2M
        t4 = t4.mul(t2); //X3
        t6 = t2.sqr(); //J^2
        t6 = t6.mul(t2); //J^3
        t6 = t6.shl(1); //2J^3

        pq[0] = t4;
        pq[1] = t5;
        pq[2] = t6;
    }
}
```

C.3 Scalar Multiplication without SPA/DPA Countermeasures

C.3.1 Original IBM Test Implementation

```
import java.math.BigInteger;

public final class ECCIBM {
    private static final BigInteger THREE =
        BigInteger.valueOf(3);
    /**
     * Scalar multiplication using addition-subtraction;
     * see IEEE P1363-2004: A.10.3.
     * @param p_x
     * @param p_y
     * @param a
     * @param m
     * @param k
     * @param bitlen
     * @param kp
     * @throws IllegalArgumentException
     */
static void fp_multiplyPointA (BigInteger p_x,
                               BigInteger p_y,
                               BigInteger a,
                               BigInteger m,
                               BigInteger k,
                               int bitlen,
                               BigInteger[] kp)
    throws IllegalArgumentException {
    BigInteger e = k.mod(m);
    BigInteger h =
        k.multiply(BigInteger.valueOf(3));

    BigInteger[] P = { p_x, p_y };
    BigInteger[] R = { p_x, p_y };

    for (int i = h.bitLength() - 2; i > 0; i--) {
        fp_doublePointA(R[0],R[1],a,m,R);
        if (h.testBit(i) && !e.testBit(i))
            fp_addPointsA(R[0],R[1],P[0],P[1],m,R);
        else if (!h.testBit(i) && e.testBit(i))
            fp_addPointsA(R[0],R[1],P[0],
                          P[1].negate(),m,R);
    }
    kp[0] = R[0];
    kp[1] = R[1];
}

/**
 * Add two affine points;
 * see IEEE P1363-2004: A.10.1.
 * @param p_x
 * @param p_y
 * @param q_x
 * @param q_y
 * @param m
 * @param pq
 * @throws IllegalArgumentException
 */
static void fp_addPointsA (BigInteger p_x,
                           BigInteger p_y,
                           BigInteger q_x,
                           BigInteger q_y,
                           BigInteger m,
```

```
                                    BigInteger[] pq)
        throws IllegalArgumentException {
        if (p_x == null) { // P == O?
            pq[0] = q_x; pq[1] = q_y;
            return;
        }

        if (p_x.equals(q_x) &&
            (p_y.equals(q_y) ||
             p_y.equals(q_y.negate())))
            throw new IllegalArgumentException();

        BigInteger d =
            (q_y.subtract(p_y)).
            multiply((q_x.subtract(p_x)).modInverse(m));

        pq[0] =
            d.pow(2).subtract(p_x).subtract(q_x).mod(m);
        pq[1] =
            d.multiply(p_x.subtract(pq[0])).
            subtract(p_y).mod(m);
    }

    /**
     * Double a point in affine coordinates;
     * see IEEE P1363-2004: A.10.1.
     * @param p_x
     * @param p_y
     * @param a
     * @param m
     * @param pp
     * @throws IllegalArgumentException
     */
    static void fp_doublePointA (BigInteger p_x,
                                 BigInteger p_y,
                                 BigInteger a,
                                 BigInteger m,
                                 BigInteger[] pp)
        throws IllegalArgumentException {
        if (p_x == null) { // P == O?
            pp[0] = pp[1] = null;
            return;
        }

        if (p_y.equals(p_y.negate()))
            throw new IllegalArgumentException();

        BigInteger d =
            (p_x.pow(2).multiply(THREE).add(a)).
            multiply(p_y.shiftLeft(1).modInverse(m));
        pp[0] =
            d.pow(2).subtract(p_x.shiftLeft(1)).mod(m);
        pp[1] =
            d.multiply(p_x.subtract(pp[0])).
            subtract(p_y).mod(m);
    }
}
```

C.3.2 Modified IBM Implementation

```
import java.math.BigInteger;

public final class ECCIBM implements IECCMultiply{
    /**
     * Scalar multiplication using addition-subtraction;
     * see IEEE P1363-2004: A.10.3.
     * @param p_x
     * @param p_y
     * @param a
     * @param m
     * @param k
     * @param kp
     * @throws IllegalArgumentException
     */
    public void multiplyPoint (IFieldElement x1,
                               IFieldElement y1,
                               IFieldElement zero,
                               IFieldElement one,
                               IFieldElement three,
                               int[] naf, int w,
                               BigInteger k,
                               IFieldElement[] kp)
        throws IllegalArgumentException {

        IFieldElement[] P = { x1, y1 };
```

```java
        IFieldElement[] Q = { x1, y1 };

        for(int i=naf.length-2;i>=0;i--) {
            Addition.doublePointA(Q[0],Q[1],
                                  three.negate(),Q);
            if (naf[i] == 1)
                Addition.addPointsA(Q[0],Q[1],
                                    P[0],P[1],Q);
            else if (naf[i] == -1)
                Addition.addPointsA(Q[0],Q[1],P[0],
                                    P[1].negate(),Q);
        }
        kp[0] = Q[0];
        kp[1] = Q[1];
    }


}
```

C.3.3 Efficient Implementation

```java
import java.math.BigInteger;
import java.lang.Math;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.Map;


public final class ECCNCM implements IECCMultiply{


    /**
     * Scalar multiplication using wNAF method and mixed
     * coordinates assuming I/M < 23;
     * @param x1
     * @param y1
     * @param a
     * @param m
     * @param k
     * @param kp
     * @throws IllegalArgumentException
     */

    public void multiplyPoint (IFieldElement x1,
                               IFieldElement y1,
                               IFieldElement zero,
                               IFieldElement one,
                               IFieldElement three,
                               int[] naf,int w,
                               BigInteger k,
                               IFieldElement[] kp)
        throws IllegalArgumentException{
        int limit = ((int)Math.pow(2,w-1))-1;

        HashMap<Integer,IFieldElement[]> precomputed =
            new HashMap<Integer,IFieldElement[]>
            (3*limit);

        //Use simultaneous inversions in
        //the precomputations
        Auxiliary.precompAffine(x1,y1,w,three.negate(),
                                precomputed);


        //Use the modification to reduce the number of
        //initial doublings.
        IFieldElement[] q;
        int s;
        int k_l = naf[naf.length-1];

        //Get the value of kappa
        int kappa = 1;
        int c = naf.length-2;
        while(naf[c]==0){
            kappa++;
            c--;
        }
        //If k_l < limit, something can be saved.
        if(k_l < limit){
            q = new IFieldElement[3];
            int l = BigInteger.valueOf(k_l).bitLength();
            int t = (k_l-(int) Math.pow(2,l-1))*
                ((int)Math.pow(2,w-1))+1;
            IFieldElement[] p1 = precomputed.get(limit);
            IFieldElement[] p2 = precomputed.get(t);
```

```
            Addition.addPointsAtoJ(p1[0],p1[1],p2[0],
                                   p2[1],q,one);
            for(int i=1; i<= kappa-w+l-1;i++)
                Addition.doublePointJ(q[0],q[1],q[2],
                                      zero,one,q);
            s = c;
        }

        //If k_l==limit, nothing can be saved.
        else{
            IFieldElement[] temp = precomputed.get(k_l);
            q = new IFieldElement[3];
            Addition.doublePointAtoJ(temp[0],temp[1],
                                     zero,one,q);
            s = naf.length-3;
        }

        for(int i=s;i>=0;i--){
            Addition.doublePointJ(q[0],q[1],q[2],zero,
                                  one,q);
            if(naf[i] != 0){
                //If naf[i] != 0 it is odd,
                //and iP has been precomputed.
                IFieldElement[] pre =
                    precomputed.get(naf[i]);
                Addition.addPointsAJtoJ(pre[0],pre[1],
                                        q[0],q[1],q[2],
                                        q,one);
            }

        }

        //Convert the result to affine coordinates
        Auxiliary.jacobianToAffine(q,kp);
    }


}
```

C.4 Scalar Multiplication with SPA Countermeasures

C.4.1 Double-and-add Always

```
import java.math.BigInteger;
import java.lang.Math;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.Map;


public final class ECCDAA implements IECCMultiply{

    /**
     * Scalar multiplication using wNAF method and
     * mixed coordinates assuming I/M < 23;
     * @param x1
     * @param y1
     * @param a
     * @param m
     * @param k
     * @param kp
     * @throws IllegalArgumentException
     */

    public void multiplyPoint (IFieldElement x1,
                               IFieldElement y1,
                               IFieldElement zero,
                               IFieldElement one,
                               IFieldElement three,
                               int[] naf, int w,
                               BigInteger k,
                               IFieldElement[] kp)
        throws IllegalArgumentException {

        IFieldElement[] q_0 =
            new IFieldElement[]{x1,y1,one};
        IFieldElement[] q_1 =
            new IFieldElement[]{one,one,zero};
```

```
        for(int i=k.bitLength()-2;i>=0;i--){
            Addition.doublePointJ(q_0[0],q_0[1],q_0[2],
                                  zero,one,q_0);
            if(k.testBit(i)){
                Addition.addPointsAJtoJ(x1,y1,q_0[0],
                                        q_0[1],q_0[2],
                                        q_0,one);
            }
            else{
                Addition.addPointsAJtoJ(x1,y1,q_0[0],
                                        q_0[1],q_0[2],
                                        q_1,one);
            }
        }
        Auxiliary.jacobianToAffine(q_0,kp);
    }

}
```

C.4.2 W-double-and-add Always

```
import java.math.BigInteger;
import java.lang.Math;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.Map;


public final class ECCWD1A implements IECCMultiply{

    /**
     * Scalar multiplication using wNAF method
     * (w-double-and-one-add-always) and mixed
     * coordinates.
     * @param x1
     * @param y1
     * @param a
     * @param m
     * @param k
     * @param kp
     * @throws IllegalArgumentException
     */
    public void multiplyPoint (IFieldElement x1,
                               IFieldElement y1,
                               IFieldElement zero,
                               IFieldElement one,
                               IFieldElement three,
                               int[] naf,int w,
                               BigInteger k,
                               IFieldElement[] kp)
        throws IllegalArgumentException {

        //Precomputations
        int limit = ((int)Math.pow(2,w-1));

        HashMap<Integer,IFieldElement[]> precomputed =
            new HashMap<Integer,IFieldElement[]>(3*limit
                );

        //Both even and odd multiples should be
        //precomputed
        Auxiliary.precompAffineWithEven(x1,y1,w,
                                        three.negate(),
                                        precomputed);
        precomputed.put(0,new IFieldElement[]{x1,y1});


        IFieldElement[] start =
            precomputed.get(naf[naf.length-1]);
        IFieldElement[] q0 =
            new IFieldElement[]{start[0],start[1],one};
        IFieldElement[] q1 = new IFieldElement[3];

        for(int i=naf.length-2;i>=0;i--){
            for(int j=w; j>0;j--){
                Addition.doublePointJ(q0[0],q0[1],q0[2],
                                      zero,one,q0);
            }

            IFieldElement[] pre = precomputed.get(naf[i
                ]);
            Addition.addPointsAJtoJ(pre[0],pre[1],q0[0],
                                    q0[1],q0[2],q1,one);

```

```
            if(naf[i] != 0){
                q0[0] = q1[0];
                q0[1] = q1[1];
                q0[2] = q1[2];
            }
            else{
                q0[0] = q0[0];
                q0[1] = q0[1];
                q0[2] = q0[2];
            }
        }

        Auxiliary.jacobianToAffine(q0,kp);
    }
}
```

C.4.3 Montgomery's Ladder Algorithm

```
import java.math.BigInteger;
import java.lang.Math;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.Map;


public final class ECCMontgomery implements IECCMultiply
    {

    /**
     * Scalar multiplication using Montgomery's ladder;
     * @param x1
     * @param y1
     * @param a
     * @param b
     * @param one
     * @param k
     * @param kp
     * @throws IllegalArgumentException
     */

    public void multiplyPoint (IFieldElement x1,
                               IFieldElement y1,
                               IFieldElement a,
                               IFieldElement b,
                               IFieldElement one,
                               int[] naf, int w,
                               BigInteger k,
                               IFieldElement[] kp)
        throws IllegalArgumentException {

        IFieldElement[] p1,p2;
        p1 = new IFieldElement[]{x1,one};
        p2 = new IFieldElement[2];

        //First doubling in affine -> projective
        Addition.doublePointMontgomeryAtoP(x1,a,b,p2);

        for (int i = k.bitLength() - 2; i >= 0; i--) {
            if(!k.testBit(i)){
                //Addition and doubling
                Addition.addPointsMontgomeryP(p1[0],
                                              p1[1],
                                              p2[0],
                                              p2[1],
                                              x1,a,b,
                                              p2);
                Addition.doublePointMontgomeryP(p1[0],
                                                p1[1],
                                                a,b,p1);

            }
            else{
                //Addition and doubling
                Addition.addPointsMontgomeryP(p1[0],
                                              p1[1],
                                              p2[0],
                                              p2[1],
                                              x1,a,b,
                                              p1);
                Addition.doublePointMontgomeryP(p2[0],
                                                p2[1],
                                                a,b,p2);
            }
        }
        //Get the affine representation of [k]P
```

```
            Auxiliary.getAffine(p1[0],p1[1],p2[0],p2[1],x1,
                                y1,a,b,kp);
    }
}
```

C.4.4 Unified Addition

```
import java.math.BigInteger;
import java.lang.Math;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.Map;


public final class ECCUnif implements IECCMultiply{

    /**
     * Scalar multiplication using unified
     * addition formulas.
     * @param x1
     * @param y1
     * @param zero
     * @param one
     * @param three
     * @param naf
     * @param w
     * @param k
     * @param kp
     * @throws IllegalArgumentException
     */

    public void multiplyPoint (IFieldElement x1,
                               IFieldElement y1,
                               IFieldElement zero,
                               IFieldElement one,
                               IFieldElement three,
                               int[] naf, int w,
                               BigInteger k,
                               IFieldElement[] kp)
        throws IllegalArgumentException {

        //Precomputations
        int limit = ((int)Math.pow(2,w-1))-1;

        HashMap<Integer,IFieldElement[]> precomputed =
            new HashMap<Integer,IFieldElement[]>(3*limit
                );

        precomputed.put(1,new IFieldElement[]{x1,y1,one
            });
        precomputed.put(-1,
                        new IFieldElement[]{x1,
                                            y1.negate(),
                                            one});
        IFieldElement[] p_sqr = new IFieldElement[3];
        Addition.doublePointAtoP(x1,y1,three,p_sqr);

        //Precompute [\pm 3]P,...,[\pm(2^{w-1}-1)]P
        IFieldElement[] most_recent =
            new IFieldElement[]{x1,y1,one};

        for(int i =3; i <= limit;i+=2){
            //Calculate [i]P
            Addition.addPointsP(most_recent[0],
                                most_recent[1],
                                most_recent[2],
                                p_sqr[0],p_sqr[1],
                                p_sqr[2],most_recent);
            precomputed.put(i,new IFieldElement[]{
                                most_recent[0],
                                most_recent[1],
                                most_recent[2]});
            //Calculate [-i]P
            precomputed.put(-i,new IFieldElement[]{
                                most_recent[0],
                                most_recent[1].negate(),
                                most_recent[2]});
        }

        IFieldElement[] start =
            precomputed.get(naf[naf.length-1]);
        IFieldElement[] q =
            new IFieldElement[]{start[0],
                                start[1],
                                start[2]};
```

```
        precomputed.put(0,q);

        int sigma = 0;
        int i = naf.length-2;
        while(i >= 0)
            {
                IFieldElement[] pre =
                    precomputed.get(sigma);
                Addition.addPointsUnifP(pre[0],pre[1],
                                        pre[2],q[0],
                                        q[1],q[2],q);
                sigma = Auxiliary.psi(sigma,naf[i]);
                i = i+Auxiliary.phi(sigma)-1;
            }

        Auxiliary.projectiveToAffine(q,kp);
    }
}
```

C.4.5 Side channel Atomicity

```
import java.lang.Math;
import java.util.ArrayList;


public final class AtomicityMatrix{

    /**
     * Returns a matrix defining the side-channel
     * atomic blocks used in the atomicity algorithm.
     */

    public static int[][] getMatrix(){

        return new int[][]{
            //Define double
            new int[]{4,3,3,5,1,4,4,4,1,4},
            new int[]{4,4,5,5,4,4,6,6,4,5},
            new int[]{6,2,3,5,4,5,5,3,6,6},
            new int[]{4,2,2,4,4,4,6,6,4,5},
            new int[]{6,4,1,6,6,6,6,7,6,6},
            new int[]{8,5,5,1,7,8,7,7,2,3},
            new int[]{4,4,4,4,4,4,4,6,1,6},
            new int[]{6,5,6,2,4,6,8,8,4,5},
            //Define addition
            new int[]{4,3,3,5,2,3,5,5,3,4},
            new int[]{5,10,4,6,10,4,5,6,5,10},
            new int[]{6,11,3,7,1,5,7,8,5,7},
            new int[]{6,6,4,8,6,4,8,8,4,8},
            new int[]{3,7,3,8,7,3,3,8,8,7},
            new int[]{8,7,7,9,10,11,9,9,8,7},
            new int[]{5,5,8,9,5,5,6,4,5,8},
            new int[]{7,7,8,9,7,9,4,4,2,6},
            new int[]{1,4,4,1,1,9,4,5,1,5},
            new int[]{5,4,5,8,5,5,7,8,5,7},
            new int[]{2,6,7,2,2,5,7,7,2,5}};
    }
}
```

```
import java.math.BigInteger;
import java.lang.Math;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.Map;


public final class ECCAtomicity implements IECCMultiply{

    private static final int[][] A_low =
        AtomicityMatrix.getMatrix();

    /**
     * Scalar multiplication using wNAF method and
     * mixed coordinates. SPA countermeasure:
     * side-channel atomicity.
     * @param x1
     * @param y1
     * @param a
     * @param m
     * @param k
     * @param kp
     * @throws IllegalArgumentException
     */

    public void multiplyPoint (IFieldElement x1,
```

```
                                IFieldElement y1,
                                IFieldElement zero,
                                IFieldElement one,
                                IFieldElement three,
                                int[] naf,int w,
                                BigInteger k,
                                IFieldElement[] kp)
        throws IllegalArgumentException {

        //Initialise temporary variables
        IFieldElement[] R = new IFieldElement[12];
        for(int i=0;i<12;i++)
            R[i] = one;

        //Precomputations
        int limit = ((int)Math.pow(2,w-1))-1;

        HashMap<Integer,IFieldElement[]> precomputed =
            new HashMap<Integer,IFieldElement[]>(3*limit
                );

        Auxiliary.precompAffine(x1,y1,w,three.negate(),
                                precomputed);
        precomputed.put(0,new IFieldElement[]{x1,y1});


        IFieldElement[] start =
            precomputed.get(naf[naf.length-1]);
        R[1]=start[0];
        R[2]=start[1];
        int s=1;
        int m=0;
        for(int i=naf.length-2;i>=0;i-=s){
            int k_i = naf[i];
            IFieldElement[] p = precomputed.get(naf[i]);
            R[10] = p[0];
            R[11] = p[1];
            m = (s==1)? 0 : m+1;
            int t = Auxiliary.phi(k_i);
            s = (int)((t==0)? Math.floor(m/7) :
                    Math.floor(m/18));

            //Perform the side-channel atomic block
            R[A_low[m][0]] =
                R[A_low[m][1]].mul(R[A_low[m][2]]);
            R[A_low[m][3]] =
                R[A_low[m][4]].add(R[A_low[m][5]]);
            R[A_low[m][6]] =
                R[A_low[m][6]].negate();
            R[A_low[m][7]] =
                R[A_low[m][8]].add(R[A_low[m][9]]);
        }
        Auxiliary.jacobianToAffine(new IFieldElement[]{
                                        R[1],R[2],R[3]},
                                    kp);
    }
}
```

C.5 Scalar Multiplication with DPA Countermeasures

C.5.1 Point Randomization by Blinding

```
import java.math.BigInteger;
import java.lang.Math;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.Map;


public final class ECCPointBlinding implements
    IECCMultiply{

    /**
     * Scalar multiplication using wNAF method and mixed
     * coordinates. DPA countermeasure: Point
     * randomization by blinding.
     * @param x1
     * @param y1
     * @param a
     * @param m
     * @param k
```

```
     * @param kp
     * @throws IllegalArgumentException
     */

    public void multiplyPoint (IFieldElement x1,
                               IFieldElement y1,
                               IFieldElement zero,
                               IFieldElement one,
                               IFieldElement three,
                               int[] naf,int w,
                               BigInteger k,
                               IFieldElement[] kp)
        throws IllegalArgumentException {

        //Get the random pair (Q,[-k]Q). The method
        //getRandom of the class RandomPoints simulates
        //a random point on the curve.
        IFieldElement[] r =
            RandomPoints.getRandom(k.bitLength());
        IFieldElement[] t = new IFieldElement[2];
        Addition.addPointsA(x1,y1,r[0],r[1],t);

        //Precompute [\pm 3]P,...,[\pm(2^{w-1}-1)]P
        int limit = ((int)Math.pow(2,w-1))-1;
        HashMap<Integer,IFieldElement[]> precomputed =
            new HashMap<Integer,IFieldElement[]>(3*limit
                );

        Auxiliary.precompAffine(t[0],t[1],w,
                                three.negate(),
                                precomputed);

        //Use the modification to reduce the number
        //of initial doublings.
        IFieldElement[] q;
        int s;
        int k_l = naf[naf.length-1];

        //Get the value of kappa
        int kappa = 1;
        int c = naf.length-2;
        while(naf[c]==0){
            kappa++;
            c--;
        }

        //If k_l < limit, something can be saved.
        if(k_l < limit){
            q = new IFieldElement[3];
            //Number of bits in k_l
            int l = BigInteger.valueOf(k_l).bitLength();
            int m = (k_l-(int) Math.pow(2,l-1))*
                ((int)Math.pow(2,w-l))+1;
            IFieldElement[] p1 = precomputed.get(limit);
            IFieldElement[] p2 = precomputed.get(m);
            Addition.addPointsAtoJ(p1[0],p1[1],
                                   p2[0],p2[1],q,one);
            for(int i=1; i<= kappa-w+l-1;i++)
                Addition.doublePointJ(q[0],q[1],q[2],
                                      zero,one,q);
            s = c;
        }

        //If k_l==limit, nothing can be saved.
        else{
            IFieldElement[] temp = precomputed.get(k_l);
            q = new IFieldElement[3];
            Addition.doublePointAtoJ(temp[0],temp[1],
                                     zero,one,q);
            s = naf.length-3;
        }

        for(int i=s;i>=0;i--){
            Addition.doublePointJ(q[0],q[1],q[2],
                                  zero,one,q);
            if(naf[i] != 0){
                //If naf[i] != 0 it is odd, and
                //[i]P is precomputed.
                IFieldElement[] pre =
                    precomputed.get(naf[i]);
                Addition.addPointsAJtoJ(pre[0],pre[1],
                                        q[0],q[1],q[2],
                                        q,one);
            }
        }
```

```
            //Add [-k]Q to get [k]P.
            Addition.addPointsAJtoJ(r[2],r[3],q[0],q[1],q
                [2],q,one);
            Auxiliary.jacobianToAffine(q,kp);
        }
}
```

C.5.2 Point Randomization by Redundancy

```
import java.math.BigInteger;
import java.lang.Math;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.Map;


public final class ECCPointRandomization implements
    IECCMultiply{

    /**
     * Scalar multiplication using wNAF method and mixed
     * coordinates. DPA countermeasure: Point
     * randomization by redundancy.
     * @param x1
     * @param y1
     * @param a
     * @param m
     * @param k
     * @param kp
     * @throws IllegalArgumentException
     */

    public void multiplyPoint (IFieldElement x1,
                               IFieldElement y1,
                               IFieldElement zero,
                               IFieldElement one,
                               IFieldElement three,
                               int[] naf,int w,
                               BigInteger k,
                               IFieldElement[] kp)
            throws IllegalArgumentException {

        //Precomputations
        int limit = ((int)Math.pow(2,w-1))-1;

        HashMap<Integer,IFieldElement[]> precomputed =
            new HashMap<Integer,IFieldElement[]>(3*limit
                );

        Auxiliary.precompAffine(x1,y1,w,three.negate(),
                                precomputed);


        //Randomize the representation in
        //Jacobian coordinates
        IFieldElement rfe =
            RandomFieldElement.getRandom(k.bitLength());
        IFieldElement rfe_2 = rfe.sqr();
        IFieldElement rfe_3 = rfe_2.mul(rfe);

        //Use the modification to reduce the number of
        //initial doublings.
        IFieldElement[] q;
        int s;
        int k_l = naf[naf.length-1];

        //Get the value of kappa
        int kappa = 1;
        int c = naf.length-2;
        while(naf[c]==0){
            kappa++;
            c--;
        }

        //If k_l < limit something can be saved.
        if(k_l < limit){
            q = new IFieldElement[3];
            //Number of bits in k_l
            int l = BigInteger.valueOf(k_l).bitLength();
            int t = (k_l-(int) Math.pow(2,l-1))*
                    ((int)Math.pow(2,w-1))+1;
            IFieldElement[] p1 = precomputed.get(limit);
            IFieldElement[] p2 = precomputed.get(t);
```

```java
            Addition.addPointsAtoJ(p1[0],p1[1],p2[0],
                                   p2[1],q,one);

            //Randomize the point
            q[0] = q[0].mul(rfe_2);
            q[1] = q[1].mul(rfe_3);
            q[2] = q[2].mul(rfe);
            for(int i=1; i<= kappa-w+l-1;i++)
                Addition.doublePointJ(q[0],q[1],q[2],
                                      zero,one,q);
            s = c;
        }

        //If k_l==limit nothing can be saved.
        else{
            IFieldElement[] temp =
                precomputed.get(k_l);
            q = new IFieldElement[3];
            Addition.doublePointAtoJ(temp[0],temp[1],
                                     zero,one,q);

            //Randomize the point
            q[0] = q[0].mul(rfe_2);
            q[1] = q[1].mul(rfe_3);
            q[2] = q[2].mul(rfe);
            s = naf.length-3;
        }

        for(int i=s;i>=0;i--){
            Addition.doublePointJ(q[0],q[1],q[2],
                                  zero,one,q);
            if(naf[i] != 0){ //If naf[i] != 0 it is odd,
                             //and [i]P has
                             //been precomputed.
                IFieldElement[] pre =
                    precomputed.get(naf[i]);
                Addition.addPointsAJtoJ(pre[0],pre[1],
                                        q[0],q[1],q[2],
                                        q,one);
            }
        }
        Auxiliary.jacobianToAffine(q,kp);
    }
}
```

C.5.3 Curve Randomization by Isomorphisms

```java
import java.math.BigInteger;
import java.lang.Math;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.Map;


public final class ECCCurveRandomization implements
    IECCMultiply{

    private static final ECCNCM ncm = new ECCNCM();

    /**
     * Scalar multiplication using wNAF method and mixed
     * coordinates. DPA countermeasure: Point
     * randomization by redundancy.
     * @param x1
     * @param y1
     * @param a
     * @param m
     * @param k
     * @param kp
     * @throws IllegalArgumentException
     */

    public void multiplyPoint (IFieldElement x1,
                               IFieldElement y1,
                               IFieldElement zero,
                               IFieldElement one,
                               IFieldElement three,
                               int[] naf,int w,
                               BigInteger k,
                               IFieldElement[] kp)
        throws IllegalArgumentException {
        int limit = ((int)Math.pow(2,w-1))-1;

```

```java
        HashMap<Integer,IFieldElement[]> precomputed =
            new HashMap<Integer,IFieldElement[]>(3*limit
                );

        //Randomize the curve
        IFieldElement mu =
            RandomFieldElement.getRandom(k.bitLength());
        IFieldElement mu_inv = mu.inv();
        IFieldElement mu_inv_2 = mu_inv.sqr();
        IFieldElement mu_inv_3 = mu_inv_2.mul(mu_inv);

        //The coefficient a' on the new curve
        IFieldElement a_prime = mu_inv_2.sqr();
        a_prime =
            a_prime.negate().sub(a_prime).sub(a_prime);

        if(a_prime.equals(BigInteger.valueOf(-3))){
            //We can use the efficient scheme
            ncm.multiplyPoint(x1,y1,zero,one,three,naf,w
                ,k,kp);
            return;
        }

        //The point P' on the new curve
        x1 = x1.mul(mu_inv_2);
        y1 = y1.mul(mu_inv_3);

        //Values needed to retrieve [k]P
        IFieldElement mu_2 = mu.sqr();
        IFieldElement mu_3 = mu_2.mul(mu);

        //Precomputations
        Auxiliary.precompAffine(x1,y1,w,
                                a_prime,precomputed);

        //Use the modification to reduce the number
        //of initial doublings.
        IFieldElement[] q_m;
        int s;
        int k_l = naf[naf.length-1];

        //Get the value of kappa
        int kappa = 1;
        int c = naf.length-2;
        while(naf[c]==0){
            kappa++;
            c--;
        }

        //If k_l < limit something can be saved.
        if(k_l < limit){
           q_m = new IFieldElement[4];
            //Number of bits in k_l
            int l = BigInteger.valueOf(k_l).bitLength();
            int t = (k_l-(int) Math.pow(2,l-1))*
                ((int)Math.pow(2,w-1))+1;
            IFieldElement[] p1 = precomputed.get(limit);
            IFieldElement[] p2 = precomputed.get(t);
            Addition.addPointsAtoJM(p1[0],p1[1],p2[0],
                                    p2[1],a_prime,
                                    q_m,one);
            for(int i=1; i<= kappa-w+l-1;i++)
                Addition.doublePointJM(q_m[0],q_m[1],
                                       q_m[2],q_m[3],
                                       q_m);
            s = c;
        }

        //If k_l==limit nothing can be saved.
        else{
            IFieldElement[] temp = precomputed.get(k_l);
           q_m = new IFieldElement[4];
            Addition.doublePointAtoJM(temp[0],temp[1],
                                      a_prime,q_m);
            s = naf.length-3;
        }

        //Perform the scalar multiplication
        IFieldElement[] q_j = new IFieldElement[3];
        for(int i=s;i>=0;i--){
            if(naf[i] !=0){
                //Double to Jacobian coordinates.
                //This gives a more efficient addition.
                Addition.doublePointJMtoJ(q_m[0],q_m[1],
                                          q_m[2],q_m[3],
                                          q_j);
```

```

                //Get the precomputed point
                IFieldElement[] pre =
                    precomputed.get(naf[i]);

                //Express the result of the addition in
                //modified Jacobian coordinates
                //to get a more efficient doubling.
                Addition.addPointsAJtoJM(pre[0],pre[1],
                                         q_j[0],q_j[1],
                                         q_j[2],a_prime,
                                         q_m,one);
            }
            else
                Addition.doublePointJM(q_m[0],q_m[1],
                                       q_m[2],q_m[3],
                                       q_m);
        }

        //Return the affine point [k]P using the
        //random isomorphism to get from [k]P' to [k]P.
        Auxiliary.jacobianToAffine(new IFieldElement[]{
                                   q_m[0],q_m[1],
                                   q_m[2]},kp);
        kp[0] = kp[0].mul(mu_2);
        kp[1] = kp[1].mul(mu_3);

    }
}
```

C.6 Scalar Multiplication with SPA & DPA Countermeasures

C.6.1 Montgomery's Ladder Algorithm & Point Randomization by Redundancy

```
import java.math.BigInteger;
import java.lang.Math;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.Map;


public final class ECCMgPr implements IECCMultiply{

    /**
     * Scalar multiplication. SPA countermeasure:
     * Montgomery's ladder. DPA countermeasure:
     * Point randomization by redundancy.
     * @param x1
     * @param y1
     * @param a
     * @param b
     * @param one
     * @param k
     * @param kp
     * @throws IllegalArgumentException
     */

    public void multiplyPoint (IFieldElement x1,
                               IFieldElement y1,
                               IFieldElement a,
                               IFieldElement b,
                               IFieldElement one,
                               int[] naf, int w,
                               BigInteger k,
                               IFieldElement[] kp)
        throws IllegalArgumentException {

        //Randomize the representation in
        //projective coordinates
        IFieldElement rfe =
            RandomFieldElement.getRandom(k.bitLength());
        IFieldElement rndx = x1.mul(rfe);
        IFieldElement rndy = y1.mul(rfe);
        IFieldElement[] p1,p2;
        p1 = new IFieldElement[]{rndx,rfe};
        p2 = new IFieldElement[2];

        //First doubling is affine -> projective
        Addition.doublePointMontgomeryP(x1,one,a,b,p2);
```

```
        for (int i = k.bitLength() - 2; i >= 0; i--){
            if(!k.testBit(i)){
                //Addition and doubling
                Addition.addPointsMontgomeryP(p1[0],
                                              p1[1],
                                              p2[0],
                                              p2[1],
                                              x1,a,b,
                                              p2);
                Addition.doublePointMontgomeryP(p1[0],
                                                p1[1],
                                                a,b,p1);

            }
            else{
                //Addition and doubling
                Addition.addPointsMontgomeryP(p1[0],
                                              p1[1],
                                              p2[0],
                                              p2[1],
                                              x1,a,b,
                                              p1);
                Addition.doublePointMontgomeryP(p2[0],
                                                p2[1],
                                                a,b,p2);
            }
        }
        //Get the affine representation of [k]P
        Auxiliary.getAffine(p1[0],p1[1],p2[0],p2[1],x1,
                            y1,a,b,kp);
    }
}
```

C.6.2 Side channel Atomicity & Point Randomization by Blinding

```
import java.math.BigInteger;
import java.lang.Math;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.Map;


public final class ECCAtPb implements IECCMultiply{

    //The atomicity matrix
    private static final int[][] A_low =
        AtomicityMatrix.getMatrix();

    /**
     * Scalar multiplication using wNAF method and mixed
     * coordinates. SPA countermeasure: Atomicity. DPA
     * countermeasure: Point randomization by blinding.
     * @param x1
     * @param y1
     * @param a
     * @param m
     * @param k
     * @param kp
     * @throws IllegalArgumentException
     */
    public void multiplyPoint (IFieldElement x1,
                               IFieldElement y1,
                               IFieldElement zero,
                               IFieldElement one,
                               IFieldElement three,
                               int[] naf, int w,
                               BigInteger k,
                               IFieldElement[] kp)
        throws IllegalArgumentException {

        //Get the random pair (Q,[-k]Q) The method
        //getRandom of the class RandomPoints simulates
        //a random point on the curve.
        IFieldElement[] r =
            RandomPoints.getRandom(k.bitLength());
        IFieldElement[] t = new IFieldElement[2];
        Addition.addPointsA(x1,y1,r[0],r[1],t);

        IFieldElement[] R = new IFieldElement[12];
        for(int i=0;i<12;i++)
            R[i] = one;

        int limit = ((int)Math.pow(2,w-1))-1;
```

```java

        HashMap<Integer,IFieldElement[]> precomputed =
            new HashMap<Integer,IFieldElement[]>(3*limit
                );

        Auxiliary.precompAffine(t[0],t[1],w,
                                three.negate(),
                                precomputed);
        precomputed.put(0,t);

        IFieldElement[] start =
            precomputed.get(naf[naf.length-1]);
        R[1]=start[0];
        R[2]=start[1];
        int s=1;
        int m=0;
        for(int i=naf.length-2;i>=0;i-=s){
            int k_i = naf[i];
            IFieldElement[] p = precomputed.get(naf[i]);
            R[10] = p[0];
            R[11] = p[1];
            m = (s==1)? 0 : m+1;
            int u = Auxiliary.phi(k_i);
            s = (int)((u==0)? Math.floor(m/7) :
                    Math.floor(m/18));

            //Perform the side-channel atomic block
            R[A_low[m][0]] =
                R[A_low[m][1]].mul(R[A_low[m][2]]);
            R[A_low[m][3]] =
                R[A_low[m][4]].add(R[A_low[m][5]]);
            R[A_low[m][6]] =
                R[A_low[m][6]].negate();
            R[A_low[m][7]] =
                R[A_low[m][8]].add(R[A_low[m][9]]);
        }
        IFieldElement[] q = new IFieldElement[3];
        Addition.addPointsAJtoJ(r[2],r[3],R[1],R[2],
                                R[3],q,one);
        Auxiliary.jacobianToAffine(q,kp);
    }
}
```

C.6.3 Side channel Atomicity & Point Randomization by Redundancy

```java
import java.math.BigInteger;
import java.lang.Math;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.Map;


public final class ECCAtPr implements IECCMultiply{

    //The atomicity matrix
    private static final int[][] A_low =
        AtomicityMatrix.getMatrix();

    /**
     * Scalar multiplication using wNAF method and mixed
     * coordinates. SPA countermeasure: Atomicity. DPA
     * countermeasure: Point randomization by redundancy
     * @param x1
     * @param y1
     * @param a
     * @param m
     * @param k
     * @param kp
     * @throws IllegalArgumentException
     */

    public void multiplyPoint (IFieldElement x1,
                               IFieldElement y1,
                               IFieldElement zero,
                               IFieldElement one,
                               IFieldElement three,
                               int[] naf,int w,
                               BigInteger k,
                               IFieldElement[] kp)
        throws IllegalArgumentException {

        //Initialize temporary variables
        IFieldElement[] R = new IFieldElement[12];
```

```
        for(int i=0;i<12;i++)
            R[i] = one;

        //Precomputations
        int limit = ((int)Math.pow(2,w-1))-1;

        HashMap<Integer,IFieldElement[]> precomputed =
            new HashMap<Integer,IFieldElement[]>(3*limit
                );

        Auxiliary.precompAffine(x1,y1,w,three.negate(),
                                precomputed);
        precomputed.put(0,new IFieldElement[]{x1,y1});

        //Randomize the representation in
        //Jacobian coordinates
        IFieldElement rfe =
            RandomFieldElement.getRandom(k.bitLength());
        IFieldElement rfe_2 = rfe.sqr();
        IFieldElement rfe_3 = rfe_2.mul(rfe);

        //Randomize the point
        IFieldElement[] start =
            precomputed.get(naf[naf.length-1]);
        R[1]=start[0].mul(rfe_2);
        R[2]=start[1].mul(rfe_3);
        R[3] = rfe;
        int s=1;
        int m=0;

        for(int i=naf.length-2;i>=0;i-=s)
            {
                int k_i = naf[i];
                IFieldElement[] p =
                    precomputed.get(naf[i]);
                R[10] = p[0];
                R[11] = p[1];
                m = (s==1)? 0 : m+1;
                int t = Auxiliary.phi(k_i);
                s = (int)((t==0)? Math.floor(m/7) :
                          Math.floor(m/18));

                //Perform the side-channel atomic block.
                R[A_low[m][0]] =
                    R[A_low[m][1]].mul(R[A_low[m][2]]);
                R[A_low[m][3]] =
                    R[A_low[m][4]].add(R[A_low[m][5]]);
                R[A_low[m][6]] =
                    R[A_low[m][6]].negate();
                R[A_low[m][7]] =
                    R[A_low[m][8]].add(R[A_low[m][9]]);
            }
        Auxiliary.jacobianToAffine(new IFieldElement[]{
                                   R[1],R[2],R[3]},
                                   kp);
    }
}
```

C.7 Auxiliary Methods

```
import java.util.HashMap;
import java.util.Map;
import java.util.ArrayList;
import java.math.BigInteger;

public final class Auxiliary{

    /**
     * Calculation of the non adjacent form.
     * @param n
     * @param w
     */

    static int[] getNAF(BigInteger n){
        return getWNAF(n,2);
    }


    /**
     * Calculation of the width-w non-adjacent form.
     * @param n
     * @param w
     * @throws IllegalArgumentException
     */
```

```java

    static int[] getWNAF(BigInteger n, int w)
    {
        ArrayList<Integer> ns =
            new ArrayList<Integer>();

        BigInteger two = BigInteger.valueOf(2);
        BigInteger pow = two.pow(w);

        if(w <= 1)
            throw new IllegalArgumentException();

        while(n.compareTo(BigInteger.valueOf(0)) == 1){
            //n odd?
            if(n.testBit(0)){
                BigInteger n_i = mods(n,pow);
                ns.add(n_i.intValue());
                n = n.subtract(n_i);
            }
            else
                ns.add(0);
            //Divide by 2
            n = n.shiftRight(1);
        }

        int[] res = new int[ns.size()];
        for(int i=0;i<ns.size();i++)
            res[i] = ns.get(i);
        return res;
    }

    /**
     * Simultaneous inversion in F_p.
     * @param a
     * @param b
     */

    static void simInv(IFieldElement[] a,
                       IFieldElement[] b){
        int j = a.length;
        IFieldElement[] c = new IFieldElement[j];
        c[0] = a[0];
        for(int i=1;i<= j-1; i++)
            c[i] = a[i].mul(c[i-1]);
        IFieldElement u = c[j-1].inv();
        for(int i=j-1; i>=1; i--){
            b[i] = u.mul(c[i-1]);
            u = u.mul(a[i]);
        }
        b[0] = u;
    }

    /**
     * Precomputation in affine coordinates using
     * simultaneous inversion.
     * Returns [1]P,[3]P,...,[2^{w-1}-1]P.
     * @param x1
     * @param y1
     * @param w
     * @param a
     * @param precomputed
     */

    static void precompAffine(IFieldElement x1,
                              IFieldElement y1,
                              int w, IFieldElement a,
                              HashMap<Integer,
                              IFieldElement[]>
                              precomputed){

        //Tempotary arrays for the coordinates of the
        //precomputed points.
        IFieldElement[] x =
            new IFieldElement[((int)Math.pow(2,w-1))];
        IFieldElement[] y =
            new IFieldElement[((int)Math.pow(2,w-1))];
        IFieldElement[] temp = new IFieldElement[2];

        Addition.doublePointA(x1,y1,a,temp);
        x[0] = x1;
        y[0] = y1;
        x[1] = temp[0]; //x-coordinate of 2p
        y[1] = temp[1]; //y-coordinate of 2p

        IFieldElement[] d;
        IFieldElement[] e;
```

```
        for(int i=1; i<=w-2;i++){
            int s = ((int)Math.pow(2,i-1))+1;
            int pw = ((int)Math.pow(2,i));
            int t = ((int)Math.pow(2,i+1));


            //Use simultaneous inversion
            if(i != w-2){
                d = new IFieldElement[s];
                e = new IFieldElement[s];
                for(int k=0; k< s-1; k++)
                    d[k] = x[pw-1].sub(x[2*k]);
                d[s-1] = y[pw-1].shl(1);
            }
            else{//[2^{w-1}]P is not used
                s--;
                d = new IFieldElement[s];
                e = new IFieldElement[s];
                for(int k=0; k<= s-1; k++)
                    d[k] = x[pw-1].sub(x[2*k]);
            }

            simInv(d,e);

            //Compute [2s-1]P,...,[2s-3+2^i]P,[2^{i+1}]P
            int k=0;
            for(int j=pw+1; j <= t-1; j+=2){
                Addition.
                    addPointsA_NoInversions(x[j-pw-1],
                                            y[j-pw-1],
                                            x[pw-1],
                                            y[pw-1],
                                            e[k],
                                            temp);
                x[j-1] = temp[0];
                y[j-1] = temp[1];
                k++;
            }
            if(i != w-2){ //[2^{w-1}]P is not used
                int h = 2*pw;
                Addition.
                    doublePointA_NoInversions(x[pw-1],
                                              y[pw-1],
                                              a,e[s-1],
                                              temp);
                x[h-1] = temp[0];
                y[h-1] = temp[1];
            }
        }

        //Return the precomputed points in the
        //supplied Hashmap.
        int limit = ((int)Math.pow(2,w-1))-1;
        for(int i = 1; i<=limit; i+=2){
                precomputed.put(i,new IFieldElement[]{
                                    x[i-1],y[i-1]});
                precomputed.put(-i,new IFieldElement[]{
                                    x[i-1],
                                    y[i-1].negate()});
            }
    }

    /**
     * Calculation of the width-w non-adjacent form for
     * w-double-one-add always (Okeya & Takagi).
     * @param n
     * @param w
     */

    static int[] getWNAFDummy(BigInteger n, int w){

        ArrayList<Integer> ns =
            new ArrayList<Integer>();
        BigInteger two = BigInteger.valueOf(2);
        BigInteger pow = two.pow(w);
        while(n.compareTo(BigInteger.valueOf(0))==1){
            BigInteger n_i = mods(n,pow);
            ns.add(n_i.intValue());
            n = n.subtract(n_i);
            n = n.shiftRight(w);
        }
        int[] res = new int[ns.size()];
        for(int i=0;i<ns.size();i++)
            res[i] = ns.get(i);
        return res;
    }
```

```java


/**
 * Precomputation in affine coordinates using
 * simultaneous inversion.
 * Returns [1]P,[2]P,...,[2^{w-1}]P.
 * @param x1
 * @param y1
 * @param w
 * @param a
 * @param precomputed
*/

static void precompAffineWithEven(IFieldElement x1,
                                  IFieldElement y1,
                                  int w,
                                  IFieldElement a,
                                  HashMap<Integer,
                                  IFieldElement[]>
                                  precomputed){

    //Tempotary arrays for the coordinates of the
    //precomputed points.
    IFieldElement[] x =
        new IFieldElement[((int)Math.pow(2,w-1))];
    IFieldElement[] y =
        new IFieldElement[((int)Math.pow(2,w-1))];
    IFieldElement[] temp = new IFieldElement[2];

    Addition.doublePointA(x1,y1,a,temp);
    x[0] = x1;
    y[0] = y1;
    x[1] = temp[0]; //x-coordinate of 2p
    y[1] = temp[1]; //y-coordinate of 2p

    for(int i=1; i<=w-2;i++){
        int s = ((int)Math.pow(2,i-1))+1;
        int pw = ((int)Math.pow(2,i));
        int h = ((int)Math.pow(2,i+1));

        //Use simultaneous inversion
        IFieldElement[] d = new IFieldElement[pw];
        IFieldElement[] e = new IFieldElement[pw];
        for(int k=0; k < pw-1; k++)
            d[k] = x[pw-1].sub(x[k]);
        d[pw-1] = y[pw-1].shl(1);
        simInv(d,e);

        //Compute [2s-1]P,[2s]P...,[2s-3+2^i]P,
        //[2^{i+1}]P
        int k=0;
        for(int j=pw+1; j <= h-1; j++){
            Addition.
                addPointsA_NoInversions(x[j-pw-1],
                                        y[j-pw-1],
                                        x[pw-1],
                                        y[pw-1],
                                        e[k],temp);
            x[j-1] = temp[0];
            y[j-1] = temp[1];
            k++;
        }

        Addition.
            doublePointA_NoInversions(x[pw-1],
                                      y[pw-1],
                                      a,e[pw-1],
                                      temp);
        x[h-1] = temp[0];
        y[h-1] = temp[1];
    }

    //Return the precomputed points in the
    //supplied Hashmap.
    int limit = ((int)Math.pow(2,w-1));
    for(int i = 1; i<=limit; i++){
        precomputed.put(i,new IFieldElement[]{
                        x[i-1],y[i-1]});
        precomputed.put(-i,new IFieldElement[]{
                        x[i-1],
                        y[i-1].negate()});
    }
}

/**
 * Returns n mod s - the smallest residue in
```

```
     * absolute value. This is  unique
     * if n is odd.
     * @param n
     * @param s
     */

    private static BigInteger mods(BigInteger n,
                                    BigInteger s){
        BigInteger r1 = n.mod(s);
        BigInteger r2 = r1.subtract(s);
        if(r2.abs().compareTo(r1.abs()) == -1)
            return r2;
        else
            return r1;
    }

    /**
     * Convert a Jacobian point to an affine one.
     * @param j
     * @param a
     */

    static void jacobianToAffine(IFieldElement[] j,
                                  IFieldElement[] a){
        if(j[2].equals(BigInteger.valueOf(0))){
            a[0] = null;
            a[1] = null;
            return;
        }

        IFieldElement z_inv = j[2].inv();
        IFieldElement z_sqr_inv = z_inv.sqr();
        IFieldElement z_cube_inv = z_sqr_inv.mul(z_inv);

        a[0] = j[0].mul(z_sqr_inv);
        a[1] = j[1].mul(z_cube_inv);
    }

    /**
     * Returns a point from the HashMap
     * @param j
     * @param a
     */

    static IFieldElement[]
        getPrecomputed(HashMap<Integer,IFieldElement[]>
                       p, int n, IFieldElement p_x,
                       IFieldElement p_y){

        IFieldElement[] pre = p.get(n);
        return (pre != null)? pre : p.get(3);
    }


    /**
     * Calculation of affine coordinates
     * including y-recovery.
     * @param x1
     * @param z1
     * @param x2
     * @param z2
     * @param x
     * @param y
     * @param a
     * @param b
     * @param r
     * @throws IllegalArgumentException
     */

    static void getAffine(IFieldElement x1,
                          IFieldElement z1,
                          IFieldElement x2,
                          IFieldElement z2,
                          IFieldElement x,
                          IFieldElement y,
                          IFieldElement a,
                          IFieldElement b,
                          IFieldElement[] r){

        if(z2.equals(BigInteger.ZERO)) //Q=O?
            throw new IllegalArgumentException();

        //Temporary variables
        IFieldElement t1,t2,t3,t4,t5,t6,t7;

        t1 = x1.mul(z1.inv());
```

```java
        t2 = x2.mul(z2.inv());

        t3 = t1.mul(x).add(a);
        t4 = t1.add(x);
        t3 = t3.mul(t4); //(x3x1-3)(x3+x1)

        t4 = x.sub(t1);
        t4 = t4.sqr();
        t4 = t4.mul(t2); //x2(x3-x1)^2

        t3 = t3.sub(t4);
        t3 = t3.add(b.shl(1)); //2b+(x3x1-3)(x3+x1)-
                               //x2(x3-x1)^2

        t2 = (y.shl(1)).inv();
        t3 = t3.mul(t2);

        r[0] = t1;
        r[1] = t3;
    }


    /**
     * Calculation of affine coordinates from
     * projective ones.
     * @param p
     * @param a
     * @throws IllegalArgumentException
     */

    static void projectiveToAffine(IFieldElement[] p,
                                   IFieldElement[] a){

        if(p[2].equals(BigInteger.ZERO)) //P=O?
            throw new IllegalArgumentException();

        IFieldElement inv = p[2].inv();

        a[0] = p[0].mul(inv);
        a[1] = p[1].mul(inv);
    }

    /**
     * Returns 0 or k depending on the value of sigma
     * @param sigma
     * @param k
     */

    static int psi(int sigma, int k){
        return (sigma == 0)? k : 0;
    }

    /**
     * Returns 0 or 1 depending on the value of sigma
     * @param sigma
     */

    static int phi(int sigma){
        return (sigma == 0)? 0 : 1;
    }

}
```

Bibliography

[ACD$^+$05] R. M. Avanzi, H. Cohen, C. Doche, G. Frey, T. Lange, K. Nguyen, and F. Vercauteren. *Handbook of Elliptic and Hyperelliptic Curve Cryptography.* CRC Press, 2005.

[AO00] M. Aigner and E. Oswald. *Power Analysis Tutorial*, 2000. Available online at `http://www.iaik.tugraz.at/research/index.php`.

[Ava05] R. M. Avanzi. *A note on the signed sliding window integer recoding and a left-to-right analogue.* In *H. Handschuh and M.A. Hasan, editors, Selected Areas in Cryptography*, volume 3357, pages 130–143. Springer-Verlag, 2005.

[BDL97] D. Boneh, R. DeMillo, and R. Lipton. *On the importance of checking cryptographic protocols.* In *Advances in Cryptology - Eurocrypt 1997*, volume 1233 of *Lecture Notes in Computer Science*, pages 37–51. Springer-Verlag, 1997.

[BHLM01] M. Brown, D. Hankerson, J. López, and A. Menezes. *Software Implementation of the NIST Elliptic Curves Over Prime Fields.* In *Topics in Cryptology – CT-RSA 2001*, pages 250–265. Springer, 2001.

[BJ02] E. Brier and M. Joye. *Weierstrass Elliptic Curves and Side-Channel Attacks.* In *D. Naccache and Pascal Paillier, Eds., Public Key Cryptography*, volume 2274 of *Lecture Notes in Computer Science*, pages 335–345. Springer-Verlag, 2002.

[BSS99] I. Blake, G. Seroussi, and N. Smart. *Elliptic Cruves in Cryptography.* Cambridge University Press, first edition, 1999.

[BSS04] I. Blake, G. Seroussi, and N. Smart. *Elliptic Cruves in Cryptography II: Further Topics.* Cambridge University Press, 2004.

[CLRS01] T. H. Cormen, C. E. Leisereson, R. L. Rivest, and C. Stein. *Introduction to Algorithms.* MIT Press, second edition, 2001.

[CMCJ04] B. Chevallier-Mames, M. Ciet, and M. Joye. *Low-Cost Solutions for Preventing Simple Side-Channel Analysis: Side-Channel Atomicity.* In *IEEE Transactions on Computers*, volume 53, 2004.

[CMO98] H. Cohen, A. Miyaji, and T. Ono. *Efficient elliptic curve exponentiation using mixed coordinates.* In *Advances in Cryptology – ASIACRYPT 98*, volume 1514 of *Lecture Notes in Computer Science*, pages 51–65. Springer-Verlag, 1998.

[Cor99] J. S. Coron. *Resistance Against Differential Power Analysis for Elliptic Curve Cryptosystems.* In *Cryptographic Hardware and Embedded Systems*, volume 1717 of *Lecture Notes in Computer Science*, pages 292–302. Springer-Verlag, 1999.

[ECR05] ECRYPT. *Yearly Report on Algorithms and Keysizes*, 2005. D.SPA.16. Available online at `http://www.ecrypt.eu.org/documents/D.SPA.16-1.0.pdf`.

[Joy05] Marc Joye. *Defences Against Side-Channel Analysis.* In *I.F Blake, G. Seroussi and N.P. Smart, Eds., Advances in Elliptic Curve Cryptography*, volume 317 of *London Mathematical Society Lecture Note Series*, pages 89–114. Cambridge University Press, 2005.

[KJJ99] P. Kocher, J. Jaffe, and B. Jun. *Differential Power Analysis.* In *Advances in Cryptology - Ctypto 99 Proceedings*, volume 1666 of *Lecture Notes in Computer Science, M. Wiener (ed.)*. Springer-Verlag, 1999.

[Kob87] N. Koblitz. *Elliptic curve cryptosystems.* In *Mathematics of Computation*, volume 48, pages 203–209, 1987.

[Kob94] N. Koblitz. *A Course in Number Theory and Cryptography.* Springer-Verlag, second edition, 1994.

[LV00] A. K. Lenstra and E. R. Verheul. *Selecting Cryptographic Key Sizes.* In *PKC '00: Proceedings of the Third International Workshop on Practice and Theory in Public Key Cryptography*, pages 446–465. Springer-Verlag, 2000.

[MDS99] T. Messerges, E.A. Dabbish, and RH. Sloan. *Investigation of Power Analysis Attacks on Smartcards.* USENIX Workshop Electronic Commerce, pages 151–161, 1999.

[Mil85] V. Miller. *Use of elliptic curves in cryptography.* In *Advances in Cryptology - CRYPTO '85*, Lecture Notes in Computer Science, pages 417–426. Springer-Verlag, 1985.

[MS04] J. A. Muir and D. R. Stinson. *Minimality and other properties of the width-w nonadjacent form.* Technical report, University of Warerloo, 2004. Combinatorics and Optimization Research report CORR 2004-08.

[NIS06] NIST. *Recommendation for Key Management*, 2006. Special Publication 800-57. Available online at `http://csrc.nist.gov/CryptoToolkit/tkkeymgmt.html`.

[OA01] E. Oswald and M. Aigner. *Randomized addition-subtraction chains as a countermeasure against power attacks.* In *Cryptographic Hardware and Embedded Systems - CHES 2001*, volume 2162 of *Lecture Notes in Computer Science*, pages 39–50. Springer-Verlag, 2001.

[OT03] K. Okeya and T. Takagi. *The Width-w NAF Method Provides Small Memory and Fast Elliptic Scalar Multiplications Secure against Side Channel Attacks.* In *CT-RSA*, pages 328–342. Springer-Verlag, 2003.

[P1300] IEEE Working Group P1363. *IEEE Standard Specifications for Public-Key Cryptography*, 2000.

[P1304] IEEE Working Group P1363a. *IEEE Standard Specifications for Public-Key Cryptography - Amendment 1: Additional Techniques*, 2004.

[RY97] M.J.B. Robshaw and Y.L. Yin. *Overview of Elliptic Curve Cryptosystems.* Technical report, RSA Laboratories, 1997.

[Sem04] O. Semay. *Efficiency analysis of window methods using Markov chains*, 2004. Diploma thesis.

[Sil92] J. H. Silverman. *Rational Points on Elliptic Curves.* Springer-Verlag, second edition, 1992.

[Sol99] J. A. Solinas. *Generalized Mersenne Numbers*, 1999. CACR.

[SST04] H. Sato, D. Schepers, and T. Takagi. *Exact Analysis of Montgomery Multiplication.* In *INDOCRYPT*, pages 290–304. Springer-Verlag, 2004.

[Wal04] C.D. Walter. *Security constraints on the Oswald-Aigner exponentiation algorithm.* In *Topics in Cryptology - CT-RSA 2003*, volume 2523, pages 391–402. Springer-Verlag, 2004.

[X9.98] ANSI X9.62. *Public Key Cryptography for The Financial Service Industry: The Elliptic Curve Digital Signature Algorithm (ECDSA)*, 1998.