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.
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 Denmark 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.


Contents

Contents iii
Preface v
Introduction vii

I Elliptic Curves 1

1 Arithmetic on Elliptic Curves 3
  1.1 General Definitions ...................................... 4
  1.2 The Group Law ........................................... 8

2 Elliptic Curves in Cryptography 13
  2.1 Cryptographic Protocols ................................. 13
  2.2 Elliptic Curves Recommended by NIST .................. 15

II Efficient Scalar Multiplication 17

3 Scalar Multiplication Methods 19
  3.1 Binary Methods ........................................ 20
  3.2 Methods using Signed Representations .................. 28
  3.3 Comparison and Conclusion ............................. 39

4 Coordinate Representations 41
  4.1 Fixed Representations .................................. 41
  4.2 Mixed Representations .................................. 44
  4.3 Comparison and Conclusion ............................. 60

5 Implementations 63
  5.1 Setup for Time Measurements ........................... 63
  5.2 IBM Test Implementation ................................ 64
  5.3 An Efficient Scheme ................................. 66
## Contents

5.4 Conclusion ........................................ 67

III Countermeasures against Power Analysis .... 69

6 Power Analysis ....................................... 71
   6.1 Simple Power Analysis ......................... 72
   6.2 Differential Power Analysis ................. 98

7 Securing an Implementation ..................... 113
   7.1 Combinations of Countermeasures .......... 113
   7.2 Comparison and Conclusion ................ 116

IV Conclusion ......................................... 121

8 Results and Recommendations .................. 123

V Appendix ........................................... 127

A Random Processes and Markov Chains .......... 129
   A.1 Basic Definitions and Results .............. 129
   A.2 Properties .................................. 132
   A.3 Asymptotic Behaviour ....................... 134

B Test Vectors ....................................... 135

C Source Code ........................................ 139
   C.1 Field Implementations ...................... 139
   C.2 Addition and Doubling ...................... 149
   C.3 Scalar Multiplication without SPA/DPA Countermeasures .... 164
   C.4 Scalar Multiplication with SPA Countermeasures .... 167
   C.5 Scalar Multiplication with DPA Countermeasures .... 172
   C.6 Scalar Multiplication with
       SPA & DPA Countermeasures ................ 177
   C.7 Auxiliary Methods ......................... 180

Bibliography .......................................... 187
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 ○ (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 ■.

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. Møesgaard 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
Preface

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 Adleman 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/.
Introduction

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 $\mathbb{F}_q^* \times \mathbb{F}_q$ 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.¹

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 [BSS91]). 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

¹It 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 $\omega_1$ and $\omega_2$ which are linearly independent over $\mathbb{R}$. These numbers, called periods, are used to define the lattice

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

A meromorphic function is given by

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

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].
Chapter 1. Arithmetic on Elliptic Curves

1.1 General Definitions

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

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

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

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

$$F(X, Y, Z) = \sum_{i,j,k} b_{i,j,k}X^iY^jZ^k, \quad b_{i,j,k} \in K.$$ 

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

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

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

$$F(X, Y, Z) = 0. \quad (1.2)$$

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

$$(X, Y, Z) \sim (X', Y', Z') \text{ if } \exists \lambda \in K^* : X = \lambda X' \land Y = \lambda Y' \land Z = \lambda Z'.$$

The quotient space $K/\sim$ is called the projective plane over $K$ and is denoted $\mathbb{P}^2(K)$ (or simply $\mathbb{P}^2$), while $K^2$ is called the affine plane over $K$ and is denoted $\mathbb{A}^2(K)$ (or simply $\mathbb{A}^2$). A point in $P \in \mathbb{P}^2(K)$ 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 $\left(\frac{X}{Z}, \frac{Y}{Z}\right) \in K^2$. 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,

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

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).
General Definitions

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

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

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

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

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

If the field \( L \) is apparent from the context, then \( C(L) \) 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 \( C \).

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 \( K \) 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 \( E \).

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 \( K \) can (using Zorn’s lemma) only be shown to be unique up to an isomorphism which fixes the elements of \( K \). We will disregard this and simply speak of the algebraic closure of \( K \).
Chapter 1. Arithmetic on Elliptic Curves

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

$$y^2 + a_1 xy + a_3 y = x^3 + a_2 x^2 + a_4 x + a_6. \quad (1.4)$$

This corresponds to the equation

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

with

$$f(x, y) = y^2 + a_1 xy + a_3 y - x^3 - a_2 x^2 - a_4 x - a_6.$$  

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 $\mathbb{K}$—rational points are:

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

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

$$(X : Y : Z) \leftrightarrow \left(\frac{x}{Z}, \frac{y}{Z}\right), \quad Z \neq 0$$

$$(0 : 1 : 0) \leftrightarrow \mathcal{O}.$$  

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

$$E : y^2 = x^3 + ax + b. \quad (1.6)$$
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
\[ E : Y^2Z = X^3 + aXZ + bZ^3. \]

So far, we have implicitly made the assumption that the variables in \( K[X, Y, Z] \) 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
\[ G(X, Y, Z) = \sum_{i,j} a_{i,j}X^iY^jZ^{2 - \deg(f) - 2i - 3j}. \tag{1.7} \]

If a point \( (\xi, \eta, \zeta) \in K^3 \) satisfies \( G(\xi, \eta, \zeta) = 0 \), then so will \( (\lambda^2 \xi, \lambda^3 \eta, \lambda \zeta) \) for any \( \lambda \in K^* \). This motivates the definition of yet another equivalence relation on \( K \). We define that
\[ (\xi, \eta, \zeta) \sim (\xi', \eta', \zeta') \text{ if } \exists \lambda \in K^* : \xi = \lambda^2 \xi' \land \eta = \lambda^3 \eta' \land \zeta = \lambda \zeta'. \]

The quotient space \( \bar{K}/ \sim \) is called the weighted projective plane over \( K \) with weights 2, 3 and 1. It is denoted \( \mathbb{P}^2_{(2,3,1)}(K) \). Points in \( \mathbb{P}^2_{(2,3,1)}(K) \) 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 \( \left( \frac{\xi}{\zeta}, \frac{\eta}{\zeta} : 1 \right) \), corresponding to the affine point \( \left( \frac{\xi}{\zeta}, \frac{\eta}{\zeta^2} \right) \). 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 \(\lambda := \frac{2}{\zeta}\) in the definition on page 7 (the definition of \(\mathbb{P}^2_{(2,3,1)}\) ensures that \(\xi \neq 0\)). Indeed, this gives \((\lambda^2 \xi, \lambda^3 \eta, 0) = (\xi^3, \xi^3, 0)\), 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 \(E(K)\) is given by

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

### 1.2 The Group Law

Let \(E\) be an elliptic curve over the field \(K\) defined by

\[
E: Y^2Z = X^3 + aXZ^2 + bZ^3.
\]

Let \(L\) be a field with \(K \subseteq L \subseteq \overline{K}\). The set \(E(L)\) of \(L\)-rational points on \(E\) has an interesting property. With a proper definition of a composition \(\oplus\), called **addition** on \(E(L)\), the pair \((E(L), \oplus)\) 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 \(E(L)\), it turns out that one has to distinguish between adding two distinct points and doubling a point. Let \(P, Q \in E(L)\) 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 \(L\)-rational. The cases \(R = P\) or \(R = Q\) are not excluded.

(ii) Let \(P\) be an \(L\)-rational point on \(E\). The tangent to \(E\) at \(P\) intersects \(E\) at exactly one further point \(R\), which is \(L\)-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 \(\mathbb{P}^2(L)\), the point \((0 : 1 : 0)\) is the only point at infinity on \(E\). Denote the third point of
The Group Law

intersection between \( E \) and the line through \( P \) and \( Q \) by \( P \ast Q \). Similarly, \( P \ast P \) denotes the other intersection point between \( E \) and the tangent to \( E \) at \( P \). The group law on \( E(L) \) 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

\[-P := (0:1:0) \ast P.\]

**Addition**: We know that \( P \ast Q \in E(L) \), and we define

\[P \oplus Q := -(P \ast Q).\]

**Doubling**: We know that \( P \ast P \in E(L) \), and we define

\[P \oplus P := -(P \ast P).\]

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 \( P \ast Q \) and taking the inverse of \( P \ast Q \). A doubling is done similarly, only with the line being a tangent to \( E \) at \( P \). The situation for \( L = \mathbb{R} \) 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 \( \mathbb{R} \).](image)

Using Max Noether’s theorem or direct calculation, one can prove that, with these definitions, \((E(L), \oplus)\) 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].
Chapter 1. Arithmetic on Elliptic Curves

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

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

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 \((E(L), \oplus)\) with neutral element \((0 : 1 : 0)\). Let \( P, Q \in E(L) \) 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:
Formulas for Addition and Doubling

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

- $O = O$
- $P \oplus O = P$
- $P - P = O$

When implementing the group law in affine coordinates, one must choose a suitable representation of $O$ 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 $(E(L), \oplus)$ with neutral element $(1 : 1 : 0)$. Let $P, Q \in E(L)$ 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
Chapter 1. Arithmetic on Elliptic Curves

\(-P = (\xi_1 : -\eta_1 : \zeta_1)\). 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 $k$. 

13
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 $h : E(\mathbb{F}_p) \rightarrow \mathbb{Z}/n\mathbb{Z}$. 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 “Strass’ 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\textsuperscript{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

\begin{align*}
    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.
\end{align*}

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].

\footnotesize{\textsuperscript{2}Federal Information Processing Standards Publication}
Chapter 2. Elliptic Curves in Cryptography

<table>
<thead>
<tr>
<th>$\mathbf{P-192}$:</th>
<th>$\mathbf{P-224}$:</th>
<th>$\mathbf{P-256}$:</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p = 2^{192} - 2^{64} - 1$</td>
<td>$p = 2^{224} - 2^{96} + 1$</td>
<td>$p = 2^{256} - 2^{128} + 2^{192} + 2^{96} - 1$</td>
</tr>
<tr>
<td>$a = -3$</td>
<td>$a = -3$</td>
<td>$a = -3$</td>
</tr>
<tr>
<td>$b = \text{0x 64210519 E59C80E7 0FA7E9AB 72243049 FEB6DEEC C1469B1}$</td>
<td>$b = \text{0x B4050A85 0C04B3AB F5413256 5044B0B7 D7FBF8BA 270B3943 2355FFB4}$</td>
<td>$b = \text{0x 5AC635D8 AA3A93E7 B3EBBD55 769886BC 651D06B0 CC530F6 3BEC3C3E 27D2604B}$</td>
</tr>
<tr>
<td>$n = \text{0x FFFFFFF FFFFFFF FFFFFFF 99DEF836 146BC9B1 B4D22831}$</td>
<td>$n = \text{0x FFFFFFF FFFFFFF FFFFFFF FFFFFFF FFFF16A2 E0B8F03E 13DD2945 5C5C2A3D}$</td>
<td>$n = \text{0x FFFFFFF 00000000 FFFFFFF FFFFFFF BCE6FAAD A7179E84 F3B9CAC2 FC632551}$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$\mathbf{P-384}$:</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$p = 2^{384} - 2^{128} - 2^{96} + 2^{32} - 1$</td>
<td></td>
</tr>
<tr>
<td>$a = -3$</td>
<td></td>
</tr>
<tr>
<td>$b = \text{0x B3312FA7 E23EE7E4 988E056B E3F82D19 181D9C6E FE814112 0314088F 5013875A C656398D 8A2ED19D 2A85C8ED D3EC2A3F}$</td>
<td></td>
</tr>
<tr>
<td>$n = \text{0x FFFFFFF FFFFFFF FFFFFFF FFFFFFF FFFFFFF FFFFFFF FFFFFFF F7634D81 F4372DDF 581A0D2B 48B0A77A ECEC196A CCC52973}$</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$\mathbf{P-521}$:</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$p = 2^{521} - 1$</td>
<td></td>
</tr>
<tr>
<td>$a = -3$</td>
<td></td>
</tr>
<tr>
<td>$b = \text{0x 00000051 953EB961 8E1C9A1F 929A21A0 B68540EE A2DA725B 99B315F3 B8B48991 8EF109E1 56193951 EC7E937B 1652C0BD 3BB1BF07 3573DF88 3D2C34F1 EF451FD4 6B503F00}$</td>
<td></td>
</tr>
<tr>
<td>$n = \text{0x FFFFFFF FFFFFFF FFFFFFF FFFFFFF FFFFFFF FFFFFFF FFFFFFF FFFFFFF FFFFFFF FFFFFFF FFFFFFF FFFFFFF FFFFFFF FFFFFFF FFFFFFF FFFFFFF}$</td>
<td></td>
</tr>
</tbody>
</table>

| $\text{Table 2.1: The table shows the five NIST curves over prime fields.}$ | 

16
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 $P \in E(F_p)$. Scalar multiplication in $E(F_p)$ 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 $F_p$. 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. 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 $t(\mathfrak{A}) = u \cdot \text{ECDBL} + v \cdot \text{ECADD}$ 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

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

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

$$[k]P \in E(F_p).$$

\footnote{An introduction to proving correctness of algorithms can be found in [CLRS01].}
Chapter 3. Scalar Multiplication Methods

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 \( \text{ECDEL} + (k - 2) \cdot \text{ECADD} \). 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

\[
\]

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. \textbf{while} \( i \geq 0 \) \textbf{do}
3. \( Q \leftarrow [2]Q; \)
4. \textbf{if} \( k_i = 1 \) \textbf{then}
5. \( Q \leftarrow P \oplus Q; \)
6. \textbf{end if}
7. \( i \leftarrow i - 1; \)
8. \textbf{end while}
9. \textbf{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

\( L: \text{At the start of each iteration of the while-loop in lines 2-8,} \)

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

\(^2\)In a general (multiplicatively written) group the algorithm is known as the square-and-multiply method and performs exponentiation.
The $2^w$-ary Method

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

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

(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 $\frac{1}{2}(l-1) \cdot \text{ECADD}$. One \text{ECDBL} per bit is always performed, so we get the following result:

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

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

**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

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

### 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

$$k = (110 001 110)_2.$$

This corresponds to the equality $k = (616)_2^w$.

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 $\sigma : \mathbb{N}_0 \rightarrow \mathbb{N} \times \mathbb{N}_0$ defined by

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

21
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:** $[k]P \in E(\mathbb{F}_p)$.

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

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

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 $(l - 1) \cdot \text{ECDBL}$ 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, $\frac{2^w - 1}{2w}$ of the $e_i$’s are non-zero, so the main loop performs

$$
(n - 1) \frac{2^w - 1}{2w} \cdot \text{ECADD} = \left( \left\lceil \frac{l}{w} \right\rceil - 1 \right) \cdot \frac{(2^w - 1)}{2^w} \cdot \text{ECADD}
$$
on average. The precomputations require one **ECDBL** and $(2^{w-1} - 1)\text{ECADD}$, so the average requirement of Algorithm 2 is:

22
The $2^w$-ary Method

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

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

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 $w \in [1, 10]$.

![Graphs showing the value of $\phi(w)$ for $l = 192$ and $l = 521$](image)

**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.

<table>
<thead>
<tr>
<th>$l$</th>
<th>$w$</th>
</tr>
</thead>
<tbody>
<tr>
<td>[70, 196]</td>
<td>4</td>
</tr>
<tr>
<td>[197, 520]</td>
<td>5</td>
</tr>
<tr>
<td>[521, 1452]</td>
<td>6</td>
</tr>
</tbody>
</table>

**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
\[
192 \cdot \text{ECDBL} + 51 \cdot \text{ECADD}.
\]
Compared to the double-and-add method, the \( 2^w \)-ary method saves \( 45 \text{ECADD} \) on average, while it uses an extra \( \text{ECDBL} \) 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 \( Q \):
\[
\]
Alternatively, one could compute
\[
\]
whereby one \( \text{ECADD} \) 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:
\[
(398)_{10} = (11\,000\,111\,0)_{2}.
\]
Skipping a zero can then be done by performing an \( \text{ECDBL} \). Algorithm 3 shows the method in general.

Remark 3.1. In lines 4-7, Algorithm 3 performs \( \text{ECDBL} \) 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 \([((k_i \cdots k_t)]P\) has been precomputed.
\[\circ\]
**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:** \([k]P \in E(\mathbb{F}_p)\).

2: \( Q \leftarrow P \); and \( i \leftarrow l - 2 \);
3: while \( i \geq 0 \) do
4: \hspace{1em} if \( k_i = 0 \) then
5: \hspace{2em} \( Q \leftarrow [2]Q \);
6: \hspace{2em} \( i \leftarrow i - 1 \);
7: \hspace{1em} else
8: \hspace{2em} \( s \leftarrow \max\{i - w + 1, 0\} \);
9: \hspace{2em} \( t \leftarrow \min\{j \in \mathbb{Z} \mid j \geq s \land k_j = 1\} \);
10: \hspace{2em} for \( h = 1 \) to \( i - t + 1 \) do
11: \hspace{3em} \( Q \leftarrow [2]Q \);
12: \hspace{2em} end for
13: \hspace{2em} \( u \leftarrow (k_i \cdots k_1)_2 \);
14: \hspace{2em} \( Q \leftarrow Q \oplus [u]P \);
15: \hspace{2em} for \( h = 1 \) to \( t - s \) do
16: \hspace{3em} \( Q \leftarrow [2]Q \);
17: \hspace{2em} end for
18: \hspace{2em} \( i \leftarrow s - 1 \);
19: \hspace{1em} 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

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

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

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

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

\[
Q = \left[ \left( \sum_{j=i+1}^{l-1} k_j 2^{j-s} \right) + 2^{l-s}u \right] P
\]

\[
= \left[ \left( \sum_{j=i+1}^{l-1} k_j 2^{j-s} \right) + 2^{l-s}(k_i 2^{i-j} + \ldots + k_t) \right] P
\]

\[
= \left[ \left( \sum_{j=i+1}^{l-1} k_j 2^{j-s} \right) + \sum_{j=t}^{i} k_j 2^{j-s} \right] P
\]

\[
= \left[ \left( \sum_{j=t}^{l-1} k_j 2^{j-s} \right) \right] P
\]

\[
= \left[ \left( \sum_{j=s}^{l-1} k_j 2^{j-s} \right) \right] P
\]

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 \texttt{ECDEL} for each bit in the binary representation
of \( k \) and that an \texttt{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 \texttt{ECADD} on average. We now see that Algorithm 3 requires

\[
\frac{1}{2} \cdot \frac{2(l-1)}{w+1} \cdot \text{ECADD} = \frac{l-1}{w+1} \cdot \text{ECADD}
\]

on average. Also counting the operations from the precomputations, one gets:
Proposition 3.3 (Requirement of the sliding-window method). One has
\[ t(\text{Algorithm 3}) = l \cdot \text{ECDBL} + \left( \frac{l - 1}{w + 1} + 2^{w-1} - 1 \right) \cdot \text{ECADD} \]
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 \(w \in [1, 10]\).](image)

<table>
<thead>
<tr>
<th>(l)</th>
<th>([25, 80])</th>
<th>([81, 240])</th>
<th>([241, 672])</th>
</tr>
</thead>
<tbody>
<tr>
<td>(w)</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

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
\[ 192 \cdot \text{ECDBL} + 45 \cdot \text{ECADD} \]
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 be 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 $(s-1) \cdot \text{ECDBL}$ and $(s-1) \cdot \text{ECADD}$. If one computes $[2^s-1]P$ as $[2^s]P \oplus (-P)$, the calculation only requires $s \cdot \text{ECDBL}$ 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,

$$23 = (110\bar{1})_2 = (11\overline{T}T1)_2,$$

where $\overline{T} = -1$. 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 $(d_{m-1} \cdots d_0)_{\text{NAF}}$.

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 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, $d \text{ mods } 4 = 1$, and in the latter case, $d \text{ mods } 4 = -1$, 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 $M = (X_n)_{n \in \mathbb{N}}$ with state space $S = \{0, *\}$, where * symbolizes 1 or $\overline{1}$. The conditional distribution of $X_{n+1}$ is
Chapter 3. Scalar Multiplication Methods

\[
P(X_{n+1} = 0 \mid X_n = 0) = \frac{1}{2}
\]
\[
P(X_{n+1} = \ast \mid X_n = 0) = \frac{1}{2}
\]
\[
P(X_{n+1} = 0 \mid X_n = \ast) = 1
\]
\[
P(X_{n+1} = \ast \mid X_n = \ast) = 0.
\]

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.

![Diagram](#)

**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 \( \pi = \left(\frac{2}{3}, \frac{1}{3}\right) \). As \( M \) is irreducible and aperiodic, Theorem A.3 in Appendix A implies that \( \mu^{(n)} \) converges to \( \pi \) 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 \( \nu \) 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,

\[
\nu \approx \frac{m}{3}.
\]
The Addition-subtraction Method

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 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 \( k = (d_l \cdots d_0)_\text{NAF} \).

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

1: \( Q \leftarrow \mathcal{O} \);
2: \( i \leftarrow l \);
3: \textbf{while} \( i \geq 0 \) \textbf{do}
4: \hspace{1em} \( Q \leftarrow [2]Q \);
5: \hspace{1em} \textbf{if} \( d_i \neq 0 \) \textbf{then}
6: \hspace{2em} \( Q \leftarrow Q \oplus [d_i]P \);
7: \hspace{1em} \textbf{end if}
8: \hspace{1em} \( i \leftarrow i - 1 \);
9: \textbf{end while}
10: \textbf{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 \( k = (k_{l-1} \cdots k_0)_2 = (d_l \cdots d_0)_\text{NAF} \), one has on average

\[
t(\text{Algorithm 6}) = l \cdot \text{ECDBL} + \frac{l}{3} \cdot \text{ECADD}.
\]

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

\[
192 \cdot \text{ECDBL} + 64 \cdot \text{ECADD}
\]

on average.

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

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

**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 \( k = (700000300001)_{\text{NAF}_w} \) with \( w = 4 \), we have \( k = 2^0(2^4(2^6 \cdot 7 - 3) + 1) \).

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:

\( \diamond \) Every integer has a unique NAF\(_ w \).

\( \diamond \) The NAF\(_ w \) of an integer \( k \) is at most one bit longer than the binary representation of \( k \).
Chapter 3. Scalar Multiplication Methods

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 recordings 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

\[ k = (d_l \cdots d_0)_{NAF_w} \]

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:** \( k = (d_l \cdots d_0)_{NAF_w} \).

1. \( i \leftarrow 0; \ d \leftarrow k; \)
2. while \( d > 0 \) do
3. \hspace{.5em} if \( d \) is odd then
4. \hspace{1em} \( d_i \leftarrow d \text{ mods } 2^w, \ // d \) is odd, so mods is well-defined.
5. \hspace{1em} \( d \leftarrow d - d_i; \)
6. \hspace{.5em} else
7. \hspace{1em} \( d_i \leftarrow 0; \)
8. \hspace{.5em} 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 \( d_i \leftarrow d \text{ mods } 2^w \) 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

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

The assignment \( d \leftarrow \frac{d}{2} \) gives

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

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.
The Width-w NAF Method

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:

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

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

\[
k = 2^i d + \sum_{j=0}^{i-1} d_j 2^j
\]

\[
= 2 \left( \frac{2^i (d - (d \text{ mods } 2^w))}{2} \right) + \sum_{j=0}^{i-1} d_j 2^j + 2^i (d \text{ mods } 2^w).
\]

(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 \( k = \sum_{j=0}^{i} d_j 2^j \), with the convention that \( d_j = 0 \) for \( j \) greater than or equal to the final value of \( i \).

\[
\square
\]

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 \( (X_n)_{n=0} \) with state space \( \mathcal{S} = \{s_1, s_2\} \), where

\[
s_1 = 0 \text{ (a single bit) and } s_2 = 0 \cdots 0*.
\]

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

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

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 $w - 1$ zeroes we know for sure to be in there), so $Y \in \mathbb{N}_0$. The event $Y = 0$ corresponds to $d$ having the form

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

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

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

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:

36
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.     $Q \leftarrow [2]Q$;
5.     if $d_i \neq 0$ then
6.         $Q \leftarrow Q \oplus [d_i]P$; / /If $d_i \neq 0$, it is odd, and $[d_i]P$ has been precomputed.
7.     end if
8.     $i \leftarrow i - 1$;
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

$\mathcal{L}$: 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

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

instead of equation (3.1).

Algorithm 7 performs one ECDEL 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 ECDEL in line 4 and the first ECADD in line 6 are not performed, as $Q = \mathcal{O}$. The
precomputations require one \texttt{ECDBL} and \((2^w - 2 - 1) \cdot \texttt{ECADD}\). 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

\[
    t(\text{Algorithm } 8) = (l + 1) \cdot \texttt{ECDBL} + \left(2^w - 1 + \frac{l}{w + 1}\right) \cdot \texttt{ECADD}
\]

on average. Algorithm 8 requires storage for \(2^w - 1\) 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 \texttt{ECADD} in the cases \(l = 192\) and \(l = 521\).

![Figure 3.4:](image)

The plots show the number of \texttt{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 \(w\).

<table>
<thead>
<tr>
<th>(l)</th>
<th>([41, 119])</th>
<th>([120, 335])</th>
<th>([336, 895])</th>
</tr>
</thead>
<tbody>
<tr>
<td>(w)</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

**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
\[
193 \cdot \text{ECDEL} + 39 \cdot \text{ECADD}
\]
on average. Compared to the sliding-window method, the NAF\(_{w}\) method saves 6 ECADD, while it introduces an extra ECDEL. 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 ECDEL 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 ECDEL 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 \( t(\text{ECDEL}) = 1.05 \cdot t(\text{ECADD}) \). The plots in Figure 3.5 correspond to an optimal choice of \( w \) where this value is used (in the 2\textsuperscript{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
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 $k = (d_l \cdots d_0)_{\text{NAF}_w}$ 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

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

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 $O$.

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:
Jacobian Coordinates

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 \( 2P = (\xi_4 : \eta_4 : \zeta_4 : a\zeta_4^4) \):

**Addition:**

Set \( A = \xi_1\zeta_2^2 \), \( B = \xi_2\zeta_1^2 \), \( C = \eta_1\zeta_3^3 \), \( D = \eta_2\zeta_3^3 \), \( E = B - A \) and \( F = D - C \).

Then, \( \zeta_3 = -E^3 - 2AE^2 + F^2 \), \( \eta_3 = -CE^3 + F(AE^2 - \zeta_3) \) and \( \zeta_3 = \zeta_1\zeta_2E \).

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

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 \( A \), \( P \), \( J \), \( J^c \) and \( J^m \) 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
Mixed Representations

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. \( J \)) or vice versa (e.g. \( P \)).

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 "\( \rightarrow \)" 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 \( A \) and express the result in \( J \), we do as follows:

\[
A \xrightarrow{(x, y)} J \xrightarrow{(x : y : 1)} [2](x : y : 1) = (\xi : \eta : \zeta) .
\]

(4.1)

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

\[
A \xrightarrow{(x, y)} J^c \xrightarrow{(x : y : 1 : 1)} J \xrightarrow{(x : y : 1 : 1 : 1)} (\xi : \eta : \zeta) \xrightarrow{J^c}
\]

\[
(\eta : \zeta) = (\xi' : \eta' : \zeta' : \zeta'^2 : \zeta'^3) .
\]

(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 \( A \) to \( P \) and from \( A \) to \( J \) are done by performing \((x, y) \rightarrow (x, y, 1)\), and conversions from \( J^c \) or \( J^m \) to \( J \) are done by disregarding one or more coordinates, conversions between \( P \) and \( J \) 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

\[
t(C^1 + C^2 = C^3)
\]
Chapter 4. Coordinate Representations

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

$$t(2C^1 = C^2)$$

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

<table>
<thead>
<tr>
<th>Doubling</th>
<th>Addition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fixed:</td>
<td>Fixed:</td>
</tr>
<tr>
<td>$t(2A)$</td>
<td>$I + 2M + 2S$</td>
</tr>
<tr>
<td>$t(2P)$</td>
<td>$7M + 5S$</td>
</tr>
<tr>
<td>$t(2J^c)$</td>
<td>$7M + 3S$</td>
</tr>
<tr>
<td>$t(2J^m)$</td>
<td>$4M + 4S$</td>
</tr>
<tr>
<td>$t(2J)$</td>
<td>$4M + 4S$</td>
</tr>
<tr>
<td>$t(2J^m = J^c)$</td>
<td>$4M + 5S$</td>
</tr>
<tr>
<td>$t(2A = J)$</td>
<td>$4M + 4S$</td>
</tr>
<tr>
<td>$t(2A = J^c)$</td>
<td>$4M + 3S$</td>
</tr>
<tr>
<td>$t(2J^m = J)$</td>
<td>$3M + 4S$</td>
</tr>
<tr>
<td>$t(2J^m = J^c)$</td>
<td>$2M + 4S$</td>
</tr>
<tr>
<td></td>
<td>$t(J^m + J^c) = J^m) = 11M + 5S$</td>
</tr>
<tr>
<td></td>
<td>$t(J + J) = 12M + 6S$</td>
</tr>
<tr>
<td></td>
<td>$t(J^m + J^m) = 12M + 4S$</td>
</tr>
<tr>
<td></td>
<td>$t(J^c + J^c) = 11M + 3S$</td>
</tr>
<tr>
<td></td>
<td>$t(J^c + J = J) = 10M + 3S$</td>
</tr>
<tr>
<td></td>
<td>$t(J^m + A = J^m) = 10M + 2S$</td>
</tr>
<tr>
<td></td>
<td>$t(J^c + A = J^c) = 8M + 3S$</td>
</tr>
<tr>
<td></td>
<td>$t(J + A = J) = 8M + 3S$</td>
</tr>
<tr>
<td></td>
<td>$t(J^m + A = J) = 8M + 3S$</td>
</tr>
<tr>
<td></td>
<td>$t(J^m + A = J^m) = 7M + 4S$</td>
</tr>
<tr>
<td></td>
<td>$t(A + J^c) = 7M + 2S$</td>
</tr>
<tr>
<td></td>
<td>$t(A + J^m) = 4M + 4S$</td>
</tr>
<tr>
<td></td>
<td>$t(A + A = J^c) = 4M + 2S$</td>
</tr>
<tr>
<td></td>
<td>$t(A + A = J) = 4M + 2S$</td>
</tr>
</tbody>
</table>

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 $[4]P \neq Q, O$. We wish to perform the following sequence of operations:
Mixed Representations

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

\[
2t(2\mathcal{A}) + t(\mathcal{A} + \mathcal{A}) = 2(I + 2M + 2S) + I + 2M + S \\
= 3I + 4M + 5S \\
= 57M.
\]

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

\[
t(2C^1 = C^2) + t(2C^2 = C^3) + t(C^3 + A = C^1) =
\min_{c^i, c^j, c^k \in C} \left( t(2C^i = C^j) + t(2C^j = C^k) + t(C^k + A = C^i) \right),
\]

(4.3)

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

\[
2t(2J) + t(J + A = J) = 2(4M + 4S) + 8M + 3S \\
= 16M + 11S \\
= 27M.
\]

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

\[\diamond\]

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

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

where

\[\diamond \ W_i \text{ is odd and } -2^{w-1} + 1 \leq W_i \leq 2^{w-1} - 1 \text{ for all } i.\]
\[ W_\nu > 0, \ k_0 \geq 0 \text{ and } k_i \geq w \text{ for all } i \geq 1. \]

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

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

i.e.

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

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

\[ w \cdot t(2C^1) + t(2C^1 = C^2) + t(C^2 + C^3 = C^1) \]

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

\[ T_w(C^1, C^2, C^3) = \frac{lw}{w+1} \cdot t(2C^1) + \frac{l}{w+1} \left( t(2C^1 = C^2) + t(C^2 + C^3 = C^1) \right) \]

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

The system \( C^3 \) is the one used for representing the precomputed points. As addition is the dominating operation involved in the precomputations, one should choose \( C^3 \) in a way such that \( t(C^3 + C^3) \) is as small as possible. From Table 4.1 one sees that both \( A \) and \( J^c \) 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 \( t(2C^1 = C^2) + t(C^2 + C^3 = C^1) \).

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

\[ t_1 = (w + 1)t(2J) + t(J + A = J) = (4w + 12)M + (4w + 7)S, \]
\[ t_2 = (w + 1)t(2J) + t(J + J^c = J) = (4w + 15)M + (4w + 7)S. \]
Mixed Representations

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

\[ t_3 = w \cdot t(2J^m) + t(2J^m = J) + t(J + A = J^m) = (4w + 11)M + (4w + 9)S, \]
\[ t_4 = w \cdot t(2J^m) + t(2J^m = J) + t(J + J^c = J^m) = (4w + 14)M + (4w + 9)S. \]

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

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


which requires one \texttt{ECDBL} and \((2^{w-2} - 1)\texttt{ECADD}\). 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 \textit{simultaneous inversion} in \(\mathbb{F}_p\). The method is shown in Algorithm 9.

\begin{algorithm}
\caption{Simultaneous inversion in \(\mathbb{F}_p\)}
\textbf{Input:} \(a_1, \ldots, a_j \in \mathbb{F}_p\) with \(a_i \neq 0\) for \(i = 1, \ldots, j\).
\textbf{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: \textbf{while} \(i \leq j\) \textbf{do}
4: \(c_i \leftarrow a_i c_{i-1};\)
5: \(i \leftarrow i + 1;\)
6: \textbf{end while}
7: \(u \leftarrow c_j^{-1};\)
8: \(i \leftarrow j;\)
9: \textbf{while} \(i \geq 2\) \textbf{do}
10: \(b_i \leftarrow uc_{i-1};\)
11: \(u \leftarrow ua_i;\)
12: \(i \leftarrow i - 1;\)
13: \textbf{end while}
14: \(b_1 \leftarrow u;\)
15: \textbf{return} \((b_1, \ldots, b_j)\)
\end{algorithm}

\textit{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

\[ \mathcal{L}: \text{At the beginning of the loop in lines 9-13 of algorithm 9,}\]

\[ u = a_i^{-1} \cdots a_1^{-1}. \]

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.

50
Efficient Precomputations

Assume that $L$ 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.

\[ \square \]

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:** $P, [3]P, \ldots, [2^{w-1}]P \in E(\mathbb{F}_p)$.

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

\[ \mathcal{L}:\text{At the beginning of the loop in lines 4-22 of Algorithm 10,} \]

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

\text{are the coordinates of} $P, [3]P, \ldots, [2^i - 1]P$ \text{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
Efficient Precomputations

inverted elements from line 12, lines 14-20 calculate

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

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

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

The ECDL 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 ECDL_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

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

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

\[(w - 1)I + (5 \cdot 2^{w-2} + 2w - 12)M + (2^{w-2} + 2w - 5)S.\]  \hspace{1cm} (4.5)

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

- \(w = 4\) : \(3I + 16M + 7S = 71M\)
- \(w = 5\) : \(4I + 38M + 13S = 115M\)
- \(w = 6\) : \(5I + 80M + 23S = 183M\),

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 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.
| $w = 4$ | 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 $[2^{\kappa W} \cdot W_\nu]P$ 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

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

This reduces $\kappa W \cdot ECDBL$ to $(\kappa W - w + 1) \cdot ECDBL$ and one ECADD. In general, one has, for $W_\nu$ 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 $W_\nu$. Assuming that $a_{l-1} = 1$, we have

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

To see this, notice that

$$2^{\kappa W - w + 1} = 2^{\kappa W + l - 1} + 2^{\kappa W} (a_{l-2} \cdot 2^{l-2} + \cdots + 1)$$

$$= 2^{\kappa W} \cdot (2^{l-1} + a_{l-2} \cdot 2^{l-2} + \cdots + 1)$$

$$= 2^{\kappa W} \cdot W_\nu.$$  

For $W_\nu \leq 15$ we have

- $W_\nu = 1: \quad [2^{\kappa W} P = [2^{\kappa W - w + 1}]([2^{w - 1} - 1]P + P)$
- $W_\nu = 3: \quad [2^{\kappa W} \cdot 3]P = [2^{\kappa W - w + 2}]([2^{w - 1} - 1]P + [2^{w-2} + 1]P)$
- $W_\nu = 5: \quad [2^{\kappa W} \cdot 5]P = [2^{\kappa W - w + 3}][2^{w - 1} - 1]P + [2^{w-3} + 1]P)$
- $W_\nu = 7: \quad [2^{\kappa W} \cdot 7]P = [2^{\kappa W - w + 3}][2^{w - 1} - 1]P + [3 \cdot 2^{w-3} + 1]P)$
- $W_\nu = 9: \quad [2^{\kappa W} \cdot 9]P = [2^{\kappa W - w + 4}][2^{w - 1} - 1]P + [2^{w-4} + 1]P)$
- $W_\nu = 11: \quad [2^{\kappa W} \cdot 11]P = [2^{\kappa W - w + 4}][2^{w - 1} - 1]P + [3 \cdot 2^{w-4} + 1]P)$
- $W_\nu = 13: \quad [2^{\kappa W} \cdot 13]P = [2^{\kappa W - w + 4}][2^{w - 1} - 1]P + [5 \cdot 2^{w-4} + 1]P)$
- $W_\nu = 15: \quad [2^{\kappa W} \cdot 15]P = [2^{\kappa W - w + 4}][2^{w - 1} - 1]P + [7 \cdot 2^{w-4} + 1]P)$

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 \((\kappa_\nu - w + l) \cdot \text{ECDBL}\) and introduces an additional \text{ECADD}. From Proposition 3.7 we know that \( \kappa_\nu = w + 1 \) on average. Therefore, one saves

\[
\frac{1 \cdot 2 + 1 \cdot 3 + 2 \cdot 4 + 4 \cdot 5 + \cdots + (2^{w-3} - 1)w}{2^{w-2}} \cdot \text{ECDBL} =
\]

\[
\frac{2 - w + \sum_{i=0}^{w-3} 2^i(i + 3)}{2^{w-2}} \cdot \text{ECDBL} =
\]

\[
(2^{-w}(-4w + 4) + w - 1) \cdot \text{ECDBL}
\]

on average by using equation (4.6). An extra

\[
\frac{2^{w-2}}{\frac{1 + 1 + \cdots + 1 + 0}{2^{w-2}}} \cdot \text{ECADD} =
\]

\[
\frac{2^{w-2} - 1}{2^{w-2}} \cdot \text{ECADD} =
\]

\[
(1 - 2^{2-w}) \cdot \text{ECADD}
\]

is needed on average.

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

\[
\sum_{i=0}^{w-2} 2^i(i + 2) - w \left(\frac{2^{w-1} - 1}{w - 1}\right) \cdot \text{ECDBL} =
\]

\[
\left(\frac{w - 1}{2^{w-1} - 1}\right) \cdot \text{ECDBL},
\]

and introduces an extra

\[
\frac{2^{w-1} - 1}{\frac{1 + 1 + \cdots + 1 + 0}{2^{w-1} - 1}} \cdot \text{ECADD} =
\]

\[
\frac{2^{w-1} - 2}{2^{w-1} - 1} \cdot \text{ECADD},
\]

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

$$FS^1(s) = t(\mathcal{A} + \mathcal{A} = \mathcal{J}) + (s + 1)t(2\mathcal{J})$$

$$= (4s + 8)M + (4s + 6)S,$$

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

$$FS^2_w = t(2\mathcal{A} = \mathcal{J}) + w \cdot t(2\mathcal{J})$$

$$= (4w + 2)M + 4(w + 1)S$$

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

$$FS_w = \frac{FS^1(1) + FS^2_w - FS^1(w - 1) + \sum_{i=0}^{w-3} 2^i \cdot FS^1(i + 2)}{2^{w-2}}$$

$$= (2^{3-w} + 4w)M + (3 \cdot 2^{3-w} + 4w - 2)S$$

on average.

For the last stage (LS) of doublings ($2^{\omega_0}(Q + [W_0]P)$), 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 $\kappa_0$ 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 $\uparrow$ in equation (4.7) below.

$$k = \overbrace{W_0 \cdots 0 W_{\nu \downarrow - 1} \cdots 0 W_1 \cdots 0 W_0 \cdots 0}_{t+1} (N_{AF_w}).$$

(4.7)
Chapter 4. Coordinate Representations

Assuming that $\kappa_0 = 1$ and $\kappa_w = w + 1$, there are $m := l - w - 1$ bits marked with $\dagger$. 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 $\dagger$ corresponds to an addition. Therefore, the average number of field operations is

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

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.

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

(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

$$PRE_w = t(2\mathcal{A} = \mathcal{J}^c) + t(\mathcal{J}^c + \mathcal{A} = \mathcal{J}^c) + (2^{w-2} - 2) t(\mathcal{J}^c + \mathcal{J}^c)$$

$$= (11 \cdot 2^{w-2} - 10) M + 3 \cdot 2^{w-2} S.$$ 

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

$$FS^1 = t(\mathcal{A} + \mathcal{J}^c = \mathcal{J}) + 2 \cdot t(2\mathcal{J})$$

$$= 15 M + 10 S.$$

If $1 < W_w < 2^{w-1} - 1$, we get

$$FS^2(s) = t(\mathcal{J}^c + \mathcal{J}^c = \mathcal{J}) + (s + 1) t(2\mathcal{J})$$

$$= (4s + 14) M + (4s + 6) S,$$
where $s$ is the binary length of $W_v$. In the case of $W_v = 2^{w-1} - 1$, the requirement is

$$FS_w^3 = t(2\mathcal{J}^c = \mathcal{J}) + w \cdot t(2\mathcal{J})$$

$$= 4(w + 1)M + (4w + 3)S$$

on average. This makes an average cost of

$$FS_w = \frac{FS_1^1 + FS_3^3 - FS_2^2(w - 1) + \sum_{i=0}^{w-3} 2^i \cdot FS_2^2(i + 2)}{2^{w-2}}$$

$$= (-5 \cdot 2^{2-w} + 4w + 6)M + (5 \cdot 2^{2-w} + 4w - 2)S$$

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 $\frac{1}{2w}$. We define the map $\psi$ 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,

$$T_w(\mathcal{J}, \mathcal{J}, \mathcal{J}^c) =$$

$$I + \left(11 \cdot 2^{w-2} - 5 \cdot 2^{2-w} + \frac{(11 - 3 \cdot 2^{2-w})m}{w + 1} + 4l - 5\right) M +$$

$$\left(3 \cdot 2^{w-2} + 5 \cdot 2^{2-w} + \frac{3m}{w + 1} + 4l - 5\right) S.$$  \hfill (4.9)
4.3 Comparison and Conclusion

As mentioned in Section 4.2, the choice between $C^3 = A$ and $C^3 = J^c$ 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 $T_w(J,J,A)$ and $T_w(J,J,J^c)$ are determined for these values of $l$. When performing these calculations, $w$ should be chosen optimally. Figure 4.1 shows $T_w(J,J,J^c)$ for $w \in [1, 10]$. One might suspect that $w = 5$ and $w = 6$ are optimal values in the two cases, and, indeed, one finds that for $l = 192$:

\[
T_4(J,J,J^c) = 2120M, \\
T_5(J,J,J^c) = 2084M, \\
T_6(J,J,J^c) = 2139M.
\]

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

Similarly, for $l = 521$:

\[
T_5(J,J,J^c) = 5463M, \\
T_6(J,J,J^c) = 5420M, \\
T_7(J,J,J^c) = 5522M.
\]

Let $T_w(C^1,C^2,C^3)$ denote the value of $T_w(C^1,C^2,C^3)$ for a fixed value of $l$. Determining when to use $A$ instead of $J^c$ for the precomputed points boils down to
solving the inequalities
\[
\min_w(T_w^l(J, J, A)) < \min_w(T_w^l(J, J, J^c)), \quad l \in \{192, 224, 256, 384, 521\}
\]
with respect to I/M.

![Graph](image)

**Figure 4.2:** The plots show the value of \(\min_w(T_w^l(J, J, A))\) and \(\min_w(T_w^l(J, J, J^c))\) for \(l = 192\) (left) and \(l = 521\) (right).

<table>
<thead>
<tr>
<th>I/M (\in)</th>
<th>(l = 192)</th>
<th>(l = 224)</th>
<th>(l = 256)</th>
<th>(l = 384)</th>
<th>(l = 521)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0, 34)</td>
<td>(0, 37)</td>
<td>(0, 41)</td>
<td>(0, 53)</td>
<td>(0, 64)</td>
<td></td>
</tr>
</tbody>
</table>

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

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

We now draw conclusions based on our observations. In Sections 4.1.1-4.1.3, formulas for the operations ECDEL and ECADD using the coordinate representations \(P, A, J, J^c\) and \(J^m\) 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:

```java
    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:
Chapter 5. Implementations

\[
n := 0;
limit := 2000 + (\text{start} := \text{System.time}());
\]

while \((\text{end} := \text{System.time}()) < \text{limit}\) do
\[
\text{IMPL();}
\]
\[
n := n+1;
\]
end while
\[
t := (\text{end}-\text{start})/n;
\]
\[
\text{return } t;
\]

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 $\mathbb{F}_{p^{192}}, \mathbb{F}_{p^{224}}, \mathbb{F}_{p^{256}}, \mathbb{F}_{p^{384}}$ and $\mathbb{F}_{p^{521}}$. 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 $S = M$ 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 [BHL01], 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.

<table>
<thead>
<tr>
<th>Operation</th>
<th>$\mathbb{F}_{p^{192}}$ Time</th>
<th>$\mathbb{F}_{p^{224}}$ Time</th>
<th>$\mathbb{F}_{p^{256}}$ Time</th>
<th>$\mathbb{F}_{p^{384}}$ Time</th>
<th>$\mathbb{F}_{p^{521}}$ Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addition</td>
<td>266 ns</td>
<td>276 ns</td>
<td>291 ns</td>
<td>338 ns</td>
<td>400 ns</td>
</tr>
<tr>
<td>Subtraction</td>
<td>248 ns</td>
<td>439 ns</td>
<td>273 ns</td>
<td>323 ns</td>
<td>686 ns</td>
</tr>
<tr>
<td>Multiplication (M)</td>
<td>3184 ns</td>
<td>3856 ns</td>
<td>4746 ns</td>
<td>8 µs</td>
<td>16 µs</td>
</tr>
<tr>
<td>Squaring (S)</td>
<td>3318 ns</td>
<td>4149 ns</td>
<td>4950 ns</td>
<td>8 µs</td>
<td>15 µs</td>
</tr>
<tr>
<td>Inversion (I)</td>
<td>51 ms</td>
<td>59 ms</td>
<td>80 ms</td>
<td>136 ms</td>
<td>223 ms</td>
</tr>
</tbody>
</table>

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

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

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

65
Chapter 5. Implementations

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

$$T_{IBM} := l \cdot t(2A) + \frac{l}{3} t(A + A)$$

$$= \frac{4l}{3} I + \frac{8l}{3} M + \frac{7l}{3} S$$

$$= \frac{79l}{3} M.$$

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

<table>
<thead>
<tr>
<th>$T_{IBM}$</th>
<th>$l = 192$</th>
<th>$l = 224$</th>
<th>$l = 256$</th>
<th>$l = 382$</th>
<th>$l = 521$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td>5056M</td>
<td>5899M</td>
<td>6741M</td>
<td>10112M</td>
<td>13720M</td>
</tr>
<tr>
<td></td>
<td>15625µs</td>
<td>22222µs</td>
<td>30769µs</td>
<td>80 ms</td>
<td>175 ms</td>
</tr>
</tbody>
</table>

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 $(C^1, C^2, C^3) = (J, J, A)$ (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 $T_{Optimized} := \min_w T_w(J, J, A)$ (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

<table>
<thead>
<tr>
<th>$T_{Optimized}$</th>
<th>$l = 192$</th>
<th>$l = 224$</th>
<th>$l = 256$</th>
<th>$l = 382$</th>
<th>$l = 521$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reduction</td>
<td>2011M</td>
<td>2326M</td>
<td>2640M</td>
<td>3866M</td>
<td>5177M</td>
</tr>
<tr>
<td></td>
<td>60.0%</td>
<td>60.6%</td>
<td>60.8%</td>
<td>61.8%</td>
<td>62.3%.</td>
</tr>
</tbody>
</table>

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.
Conclusion

<table>
<thead>
<tr>
<th>Time</th>
<th>( l = 192 )</th>
<th>( l = 224 )</th>
<th>( l = 256 )</th>
<th>( l = 382 )</th>
<th>( l = 521 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( 7352 \mu s )</td>
<td>( 10050 \mu s )</td>
<td>( 13698 \mu s )</td>
<td>( 34482 \mu s )</td>
<td>( 81 ) ms</td>
<td></td>
</tr>
</tbody>
</table>

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 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 ECDELB. Each type of field operation has its own unique power consumption trace. Therefore, an ECADD and an ECDELB 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 ECDELB 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 ECDELB and the other sub-trace as an ECADD. Knowing that an ECDELB corresponds to a zero-bit in the scalar $k$ and that an ECDELB 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.

72
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:** $P \in E(F_p)$ and $k = (k_{l-1} \cdots k_0)_2$

**Output:** $[k]P \in E(F_p)$.

1. $Q_0 \leftarrow P; Q_1 \leftarrow O; i \leftarrow l - 2$;
2. while $i \geq 0$ do
3.   $Q_0 \leftarrow [2]Q_0$;
4.   $Q_{1-k_i} \leftarrow Q_{1-k_i} \oplus P$;
5.   $i \leftarrow i - 1$;
6. end while
7. 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 $(l - 1)(\text{ECDBL} + \text{ECADD})$.

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 \( k_1 = 0 \), the operation in line 4 is a dummy. If the calculation \( Q_{1-k_1} \leftarrow Q_{1-k_1} \oplus P \) 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

\[
T_{\text{Alg,11}} = (l-1)(t(2\mathcal{J}) + t(\mathcal{J} + \mathcal{A} = \mathcal{J}))) + C
\]

\[
= I + (12l - 9)M + (7l - 6)S.
\]

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
Algorithms with Uniform Behaviour

<table>
<thead>
<tr>
<th>$T_{\text{Alg.11}}$</th>
<th>$l = 192$</th>
<th>$l = 224$</th>
<th>$l = 256$</th>
<th>$l = 384$</th>
<th>$l = 521$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overhead</td>
<td>81.5%</td>
<td>83.0%</td>
<td>84.2%</td>
<td>88.7%</td>
<td>91.2%</td>
</tr>
</tbody>
</table>

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

$$k = (U_d 0 \cdots 0 U_{d-1} 0 \cdots 0 U_0 0 \cdots 0)_2,$$

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:** $U_{\lceil \frac{d}{w} \rceil}, \ldots, U_0$ such that $k = (U_{\lceil \frac{d}{w} \rceil} \cdots U_0)_{\text{NAF}_w^*}$.

1: $d \leftarrow \lceil \frac{l}{w} \rceil$;
2: $i \leftarrow 0$;
3: while $i \leq d$ do
4: $U_i \leftarrow k \mod 2^w$;
5: $k \leftarrow k - U_i$;
6: $k \leftarrow \frac{k}{2^w}$;
7: $i \leftarrow i + 1$;
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: \( P \in E(\mathbb{F}_p) \), \( w \geq 1 \) and \( k = (U_d \cdots U_0)_{N_{AF_e}} \).

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

\[
\text{ECDBL, \ldots, ECDBL, ECADD, \ldots, ECDBL, ECADD, \ldots}
\]

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 \( \mathcal{A} \), \( w > 1 \).


1. \((x_1, y_1) \leftarrow P;\)
2. \((x_2, y_2) \leftarrow \text{ECDBL}(P);\)
3. \(i \leftarrow 1;\)
4. **while** \( i \leq w - 2 \) **do**
   5. \((d_1, \ldots, d_{2^i}) \leftarrow (x_{2^i} - x_1, x_{2^i} - x_2, \ldots, x_{2^i} - x_{2^{i-1}}, 2y_{2^i});\)
   6. \((\delta_{2^{i+1}}, \delta_{2^{i+2}}, \ldots, \delta_{2^{i+1-1}}, \delta_{2^{i+1}}) \leftarrow \text{SIMINV}(d_1, \ldots, d_{2^i}); /\text{SIMINV is an implementation of Algorithm 9.}\)
   7. \(j \leftarrow 2^i + 1;\)
   8. **while** \( j \leq 2^{i+1} - 1 \) **do**
   9. \((x_j, y_j) \leftarrow \text{ECADD_NI}((x_{j-2^i}, y_{j-2^i}), (x_{2^i}, y_{2^i}), \delta_j);\)
   10. \(j \leftarrow j + 1;\)
   11. **end while**
   12. \((x_{2^{i+1}}, y_{2^{i+1}}) \leftarrow \text{ECDBL_NI}((x_{2^i}, y_{2^i}), \delta_{2^{i+1}});\)
   13. \(i \leftarrow i + 1;\)
14. **end while**
15. **return** \( ((x_1, y_1), (x_2, y_2), \ldots, (x_{2^{w-1}}, y_{2^{w-1}})) \)

The **ECDBL** requires \( I + 2M + 2S \). The \( w - 2 \) iterations of the main loop each requires

- One simultaneous inversion of \( 2^i \) elements.
- \((2^i - 1)\text{ECADD_NI}.\)
- One **ECDBL_NI**.

This makes a total requirement of

\[
\text{PRE}_w^A := I + 2M + 2S + \sum_{i=1}^{w-2}(I + 2^i(5M + S) - 3M + S) = \\
(w - 1)I + (5 \cdot 2^{w-1} - 3w - 2)M + (2^{w-1} + w - 2)S
\]

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

\[
\text{PRE}_w^{\mathcal{J}^c} = t(2A = \mathcal{J}^c) + t(A + \mathcal{J}^c = \mathcal{J}^c) + (2^{w-1} - 4) \cdot t(\mathcal{J}^c + \mathcal{J}^c) = \\
(11 \cdot 2^{w-1} - 32)M + (3 \cdot 2^{w-1} - 6)S.
\]

For the fist stage of doublings \( [2^w U_d]P \) 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
Chapter 6. Power Analysis

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 vary according to the value of $U_d$. Because of this, we refrain from using the modification.

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

$$T^A_w := \text{PRE}_w^A + t(2A = J) + \left( w \cdot \left\lceil \frac{l}{w} \right\rceil - 1 \right) \cdot t(2J) + \left( \left\lceil \frac{l}{w} \right\rceil - 1 \right) \cdot t(A + J = J) + C,$$

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

$$T^{J^c}_w := \text{PRE}_w^J + t(2J^c = J) + \left( w \cdot \left\lceil \frac{l}{w} \right\rceil - 1 \right) \cdot t(2J) + \left( \left\lceil \frac{l}{w} \right\rceil - 1 \right) \cdot t(J^c + J = J) + C.$$

Using the values from Table 4.1, we get

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

and

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

The values of $\min_w(T^A_w)$ and $\min_w(T^{J^c}_w)$ are shown in Figure 6.2 for $l = 192$ and $l = 521$ respectively.
Algorithms with Uniform Behaviour

Figure 6.2: The plots show the number of field multiplications in min_w(T^A_w) and min_w(T^J_w) for l = 192 (left) and l = 521 (right).

As the figure indicates, we should choose 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 A for

<table>
<thead>
<tr>
<th>l</th>
<th>192</th>
<th>224</th>
<th>256</th>
<th>384</th>
<th>521</th>
</tr>
</thead>
<tbody>
<tr>
<td>v</td>
<td>55</td>
<td>61</td>
<td>67</td>
<td>82</td>
<td>98</td>
</tr>
</tbody>
</table>

Table 6.2: For different values of l, the table shows a value v which satisfies that 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

$$l \cdot \text{ECDBL} + (l - 1) \cdot \text{ECADD}.$$
Chapter 6. Power Analysis

<table>
<thead>
<tr>
<th>$l$</th>
<th>$T_w^A$</th>
<th>$T_w^B$</th>
<th>$T_w^C$</th>
<th>$T_w^D$</th>
<th>$T_w^E$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$l = 192$</td>
<td>2142M</td>
<td>2448M</td>
<td>2800M</td>
<td>4039M</td>
<td>5396M</td>
</tr>
<tr>
<td>$l = 224$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$l = 256$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$l = 384$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$l = 512$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

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

1. $P_1 \leftarrow P$; $P_2 \leftarrow [2]P$;
2. $i \leftarrow l - 2$;
3. while $i \geq 0$ do
4.   if $k_i = 0$ then
5.      $P_2 \leftarrow P_1 + P_2$; $P_1 \leftarrow [2]P_1$;
6.   else
7.      $P_1 \leftarrow P_1 + P_2$; $P_2 \leftarrow [2]P_2$;
8. end if
9. $i \leftarrow i - 1$;
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 $a = \frac{1}{3^2} - \frac{A^2}{3^3}$ and $b = -\frac{A^3}{27B^2} - a\frac{A}{3B}$, but the converse is not true.

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

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

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

80
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 $\alpha$ 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 $K$ be a field with $\text{char}(K) \neq 2, 3$, and let $E : y^2 = x^3 + ax + b$ be an elliptic curve over $K$. Let $P = (x_1, y_1)$ and $Q = (x_2, y_2)$ be given such that $P, Q \in E(K) \setminus \{O\}$ 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

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

(6.2)

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

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

(6.3)

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

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

(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 $P, Q \neq O$ 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 $X_i, Z_i, Z \in K$, $i = 1, 2, 3$, with $Z_1 Z_2 Z \neq 0$. Substituting into equation (6.2) gives

$$\frac{X}{Z} = \frac{(\frac{X_1 X_2}{X_1 Z_2} - a)^2 - 4b(\frac{X_1}{Z_1} + \frac{X_2}{Z_2})}{x_3(\frac{X_1}{Z_1} - \frac{X_2}{Z_2})^2} = \frac{(X_1 X_2 - a Z_1 Z_2)^2 - 4b Z_1 Z_2 (X_1 Z_2 + X_2 Z_1)}{x_3 (X_1 Z_2 - X_2 Z_1)^2}.$$
From this we see that formulas for $X$ and $Z$ are

$$X = (X_1X_2 - aZ_1Z_2)^2 - 4bZ_1Z_2(X_1Z_2 + X_2Z_1)$$

$$Z = x_3(X_1Z_2 - X_2Z_1)^2.$$  

(6.5)

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

$$X = (X_1^2 - aZ_1^2)^2 - 8bX_1Z_1^3$$

$$Z = 4Z_1(X_1^3 + aX_1Z_1^2 + bZ_1^3).$$  

(6.6)

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

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

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

$$T_1 := 2M + 2S + (l - 1) \cdot (12M + 5S)$$

$$= (12l - 10)M + (5l - 3)S.$$

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

$$T_{Montgomery} := T_1 + T_2$$

$$= 3I + (12l - 4)M + (5l - 2)S.$$

This gives the values in table 6.4.

<table>
<thead>
<tr>
<th>$T_{Montgomery}$</th>
<th>$l = 192$</th>
<th>$l = 224$</th>
<th>$l = 256$</th>
<th>$l = 384$</th>
<th>$l = 521$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overhead</td>
<td>64.4%</td>
<td>65.5%</td>
<td>66.4%</td>
<td>70.0%</td>
<td>71.9%</td>
</tr>
</tbody>
</table>

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 \(K\) with \(\text{char}(K) \neq 2, 3\). Let \(P = (x_1, y_1)\) and \(Q = (x_2, y_2)\) be \(K\)-rational points on \(E\) with \(P, Q \neq \mathcal{O}\) and \(P \neq -Q\). Then, \(P \oplus Q = (x_3, y_3)\) with

\[
x_3 = \lambda^2 - x_1 - x_2, \quad y_3 = \lambda(x_1 - x_3) - y_1,
\]

where

\[
\lambda = \frac{x_1^2 + x_1x_2 + x_2^2 + a}{y_1 + y_2}.
\]

If we recall that \(x_1^2 + x_1x_2 + x_2^2 = (x_1 + x_2)^2 - x_1x_2\), when \(\text{char}(K) \neq 2\), 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

\[
\lambda = \frac{(x_1 + x_2)^2 - x_1x_2 + a}{y_1 + y_2}
\]

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

\[
2y_3 = \lambda(x_1 + x_2 - 2x_3) - (y_1 + y_2).
\]
Chapter 6. Power Analysis

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

\[
x_3 = \frac{X_3}{Z_3} = \frac{[(X_1Z_2 + X_2Z_1)^2 - X_1Z_2X_2Z_1 + a(Z_1Z_2)^2]}{(Z_1Z_2)^2(Y_1Z_2 + Y_2Z_1)^2} - \frac{Z_1Z_2(Y_1Z_2 + Y_2Z_1)^2(X_1Z_2 + X_2Z_1)}{(Z_1Z_2)^2(Y_1Z_2 + Y_2Z_1)^2}
\]

and

\[
2y_3 = \frac{2Y_3}{Z_3} = \frac{3[(X_1Z_2 + X_2Z_1)^2 - X_1Z_2X_2Z_1 + a(Z_1Z_2)^2]}{(Z_1Z_2)^2} (X_1Z_2 + X_2Z_1)Z_2(Y_1Z_2 + Y_2Z_1)^2 - \frac{2[(X_1Z_2 + X_2Z_1)^2 - X_1Z_2X_2Z_1 + a(Z_1Z_2)^2]^3 + (Y_1Z_2 + Y_2Z_1)^4(Z_1Z_2)^2}{(Z_1Z_2)^2(Y_1Z_2 + Y_2Z_1)^2}.
\]

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

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

\[
A = Z_1Z_2, \quad B = X_1Z_2, \quad C = X_2Z_1, \quad D = Y_1Z_2, \quad E = Y_2Z_1, \\
F = B + C, \quad G = D + E, \quad H = F^2 - BC + aA^2, \quad J = AG, \\
K = GJ, \quad L = FK, \quad M = H^2 - L.
\]  

(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 \( \text{NAF}_w \) is given, so we construct one. Algorithm 16 shows the result, in which \( \delta \) and \( \varphi \) 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 $k = (d_i \cdots d_0)_{N_{AF_w}}$.

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

2: $(R_1, R_3, \ldots, R_{2^{w-1} - 1}) \leftarrow (P, [3]P, \ldots, [2^{w-1} - 1]P)$;
3: $R_0 \leftarrow [d_0]P$;
4: $i \leftarrow l - 1$; $\sigma \leftarrow 0$;
5: **while** $i \geq 0$ **do**
6: $R_0 \leftarrow R_0 \oplus R_{\sigma}$; /*Use unified addition and doubling.**
7: $\sigma \leftarrow \varphi(\sigma, d_i)$;
8: $i \leftarrow i + \delta(d_i) - 1$;
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 ECDEL 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 ECDEL 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

$$PRE_w^P = t(2\mathcal{A} = \mathcal{P}) + t(\mathcal{A} + \mathcal{P} = \mathcal{P}) + (2^{w-2} - 2)t(\mathcal{P} + \mathcal{P})$$

$$= (3 \cdot 2^w - 11)M + (2^{w-1} + 2)S$$

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

$$T_w^P = PRE_w^P + \left(l + \frac{l}{w+1}\right) \cdot (12M + 5S) + C$$

$$= I + \left(3 \cdot 2^w + 12l \left(1 + \frac{1}{w+1}\right) - 9\right)M +$$

$$\left(2^{w-1} + 5l \left(1 + \frac{1}{w+1}\right) + 2\right)S$$

on average.

Precomputations in $\mathcal{A}$ require

$$PRE_w^A = (w - 1)I + (5 \cdot 2^{w-2} + 2w - 12)M + (5 \cdot 2^{w-2} + 2w - 5)S.$$
The total average requirement, when using $\mathcal{A}$ for precomputation, is

$$T^A_w = PRE^A_w + \left( l + \frac{l}{w+1} \right) \cdot (I + 3M + 2S) = \left( w - 1 + l + \frac{l}{w+1} \right) I + \left( 5 \cdot 2^{w-2} + 2w + 3l \left( 1 + \frac{1}{w+1} \right) - 12 \right) M + \left( 2^{w-2} + 2w + 2l \left( 1 + \frac{1}{w+1} \right) - 5 \right) S.$$

The values of $\min_w(T^A_w)$ and $\min_w(T^P_w)$ 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 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]).

![Figure 6.3: The plots show the number of field multiplications in $\min_w(T^A_w)$ and $\min_w(T^P_w)$ for $l = 192$ (left) and $l = 521$ (right) respectively.](image)
Algorithms with Dummy Field Operations

<table>
<thead>
<tr>
<th>$T^{P}_{w}$</th>
<th>$l = 192$</th>
<th>$l = 224$</th>
<th>$l = 256$</th>
<th>$l = 384$</th>
<th>$l = 521$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$w = 5$</td>
<td>3929M</td>
<td>4564M</td>
<td>5198M</td>
<td>7694M</td>
<td>10355M</td>
</tr>
<tr>
<td>$w = 5$</td>
<td>5198M</td>
<td>7694M</td>
<td>10355M</td>
<td>10355M</td>
<td></td>
</tr>
<tr>
<td>$w = 6$</td>
<td>7694M</td>
<td>10355M</td>
<td>10355M</td>
<td>10355M</td>
<td></td>
</tr>
<tr>
<td>$w = 6$</td>
<td>10355M</td>
<td>10355M</td>
<td>10355M</td>
<td>10355M</td>
<td></td>
</tr>
</tbody>
</table>

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.
Chapter 6. Power Analysis

Section 6.1.3 is excluded from this version.
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.
Chapter 6. Power Analysis

Section 6.1.3 is excluded from this version.
Section 6.1.3 is excluded from this version.
Chapter 6. Power Analysis

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.

Section 6.1.3 is excluded from this version.
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.
Comparison and Conclusion

### Field multiplications:

<table>
<thead>
<tr>
<th>Countermeasure</th>
<th>( l = 192 )</th>
<th>( l = 224 )</th>
<th>( l = 226 )</th>
<th>( l = 384 )</th>
<th>( l = 521 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>None (algorithm 8)</td>
<td>( M )</td>
<td>( z )</td>
<td>( M )</td>
<td>( z )</td>
<td>( M )</td>
</tr>
<tr>
<td>Double-and-add always (algorithm 11)</td>
<td>2011 7 2326 7 3640 7 3866 15 5177 15</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>W-double-one-add always (algorithm 13)</td>
<td>2142 14 2448 14 2800 30 4039 30 5396 30</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Montgomery’s ladder algorithm (algorithm 15)*</td>
<td>3306 0 3850 0 4394 0 6570 0 8899 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Unified addition (algorithm 16)*</td>
<td>3929 7 4564 7 5198 7 7694 15 10355 15</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Side channel atomicity</td>
<td>2023 7 2338 7 2652 7 3878 15 5190 15</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

* Secure against FI attacks.

### Timings:

<table>
<thead>
<tr>
<th>Countermeasure</th>
<th>( l = 192 )</th>
<th>( l = 224 )</th>
<th>( l = 226 )</th>
<th>( l = 384 )</th>
<th>( l = 521 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>None (algorithm 8)</td>
<td>Time</td>
<td>Time</td>
<td>Time</td>
<td>Time</td>
<td>Time</td>
</tr>
<tr>
<td>Double-and-add always (algorithm 11)</td>
<td>7352 ( \mu s )</td>
<td>10050 ( \mu s )</td>
<td>13698 ( \mu s )</td>
<td>34482 ( \mu s )</td>
<td>81 ( ms )</td>
</tr>
<tr>
<td>W-double-one-add always (algorithm 13)</td>
<td>13513 ( \mu s )</td>
<td>18691 ( \mu s )</td>
<td>25641 ( \mu s )</td>
<td>68 ( ms )</td>
<td>166 ( ms )</td>
</tr>
<tr>
<td>Montgomery’s ladder algorithm (algorithm 15)*</td>
<td>8368 ( \mu s )</td>
<td>11235 ( \mu s )</td>
<td>15503 ( \mu s )</td>
<td>38769 ( \mu s )</td>
<td>92 ( ms )</td>
</tr>
<tr>
<td>Unified addition (algorithm 16)*</td>
<td>12048 ( \mu s )</td>
<td>16806 ( \mu s )</td>
<td>22988 ( \mu s )</td>
<td>61 ( ms )</td>
<td>149 ( ms )</td>
</tr>
<tr>
<td>Side channel atomicity</td>
<td>12987 ( \mu s )</td>
<td>18018 ( \mu s )</td>
<td>24888 ( \mu s )</td>
<td>66 ( ms )</td>
<td>162 ( ms )</td>
</tr>
</tbody>
</table>

* Secure against FI attacks.

**Table 6.6:** The tables show the average number of field multiplications required \( (M) \), the number of precomputed points \( (z) \) 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 \cite{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 \( t_{\text{max}} \) be the maximum number of time units, e.g. \( \text{ns} \) or \( \mu\text{s} \), required to execute \( \mathfrak{A} \). Let

\[
f(g, t), \quad g \in G, \ t \in [0, t_{\text{max}}]
\]
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 \textit{external noise} (generated by some external object), \textit{intrinsic noise} (generated by certain random movements within conductors in the hardware), \textit{quantification noise} (from the quantizer in the analog-to-digital converter used to sample the power signals) and \textit{algorithmic noise} (due to the random data being processed by the hardware). For details of noise characteristics see \cite{MDS99}.

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

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

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 \cite{MDS99} and \cite{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...
Differential Power Analysis

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

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

2) He/she computes

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

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 a concrete situation. We will assume that the Hamming weight $\nu$ of the representation $\text{rep}(P_i)$ of a point $P_i$ influences the power consumption in the system. We define $\Phi$ as

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

for some fixed value $\nu_0$. The map $\Phi$ is used to construct the sets

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

This construction can be done on a separate device.

4) With a partitioning

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

\footnote{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.}
Chapter 6. Power Analysis

of \([0, t_{\text{max}}]\), 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 \(\Delta_j\) during the calculation of \([k]P_i\).

He/she sets
\[
A_0(\Delta_j) := \frac{1}{|S_0|} \sum_{i \in S_0} F_i(\Delta_j), \quad A_1(\Delta_j) := \frac{1}{|S_1|} \sum_{i \in S_1} F_i(\Delta_j)
\]
for \(j = 1, \ldots, d\).

The collection of power measurements
\[
F_i(\Delta_j), \quad i = 1, \ldots, n, \quad j = 1, \ldots, d
\]
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
\[
\max_{1 \leq j \leq d} |A_0(\Delta_j) - A_1(\Delta_j)| \approx 0,
\]
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 \(k_{i-s-1}\) is \(\neg \kappa\). If
\[
\max_{1 \leq j \leq d} |A_0(\Delta_j) - A_1(\Delta_j)| > 0,
\]
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.
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 \textit{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.

\textbf{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 $\varphi$ of the scalar $k$ is given by $\varphi(k) = k + s\sigma$, where $s$ is a random positive integer. The map $\varphi$ should be applied every time a scalar multiplication is performed. Algorithm 17 shows the general method, where $\text{ECMUL}_T$ is any scalar multiplication algorithm.

<table>
<thead>
<tr>
<th>Algorithm 17 Scalar multiplication with randomized scalar</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input:</strong> $P \in E(\mathbb{F}_p)$ and $k \in \mathbb{Z}$, $k \geq 1$.</td>
</tr>
<tr>
<td><strong>Output:</strong> $[k]P \in E(\mathbb{F}_p)$.</td>
</tr>
<tr>
<td>1: $k' \leftarrow \varphi(k)$;</td>
</tr>
<tr>
<td>2: $Q \leftarrow \text{ECMUL}_T(P, k')$;</td>
</tr>
<tr>
<td>3: return $Q$</td>
</tr>
</tbody>
</table>

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

$$[\log_2(s)] + [\log_2(\sigma)] \leq [\log_2(k')] \leq [\log_2(s)] + [\log_2(\sigma)] + 1,$$

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 $r = [\log_2(s)]$, one finds that

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

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 $\text{ECMUL}_T$ in Algorithm 17, so we choose Algorithm 8 (with the modifications discussed in Sections 4.2.1 and 4.2.2). We have $[\log_2(\sigma)] = 192, 224, 256, 384$ and 521 for P-192, P-224, P-256, P-384 and P-521 respectively. Assume that $[\log_2(k + s\sigma)] = [\log_2(s)] + [\log_2(\sigma)] + 1$, 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.
Scalar randomization by variation

<table>
<thead>
<tr>
<th>Field multiplications</th>
<th>$l = 192$</th>
<th>$l = 224$</th>
<th>$l = 256$</th>
<th>$l = 384$</th>
<th>$l = 521$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Optimal value of $w$</td>
<td>$2336M$</td>
<td>$2650M$</td>
<td>$2957M$</td>
<td>$4182M$</td>
<td>$5493M$</td>
</tr>
<tr>
<td>Overhead</td>
<td>$w = 5$</td>
<td>$w = 5$</td>
<td>$w = 6$</td>
<td>$w = 6$</td>
<td>$w = 6$</td>
</tr>
<tr>
<td></td>
<td>$16.2%$</td>
<td>$14.0%$</td>
<td>$12.0%$</td>
<td>$8.2%$</td>
<td>$6.1%$</td>
</tr>
</tbody>
</table>

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 $k_{i-1}, \ldots, k_{i+1}$ are known, the attacker makes a guess of $k_i = 0$ or $k_i = 1$ and defines the point

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

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

$$C = \left\{ \left( t, \frac{1}{n} \sum_{j=1}^{n} F_j(t) \right) \bigg| 0 \leq t \leq t_{max} \right\}.$$  

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 $k - b \cdot |E(F_p)|$ (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

$$\mathcal{R} = \{(Q_i, [k](-Q_i)) \mid i = 1, \ldots, n\}$$

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

$$T_{Add} = t(A + A) + t(J + A = J)$$
$$= I + 10M + 4S.$$
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: $(Q, [k](-Q)) \leftarrow (Q_{\text{iv}}, [k](-Q_{\text{iv}})) \in \mathcal{R}$; //Randomly chosen
2: $R \leftarrow P \oplus Q$;
3: $R \leftarrow \text{ECMUL}(R, k)$;
4: $R \leftarrow R \oplus [k](-Q)$;
5: return $R$

<table>
<thead>
<tr>
<th>Field multiplications</th>
<th>$l = 192$</th>
<th>$l = 224$</th>
<th>$l = 256$</th>
<th>$l = 384$</th>
<th>$l = 521$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2041M</td>
<td>2356M</td>
<td>2670M</td>
<td>3896M</td>
<td>5207M</td>
</tr>
<tr>
<td>Overhead</td>
<td>1.5%</td>
<td>1.3%</td>
<td>1.1%</td>
<td>0.8%</td>
<td>0.6%</td>
</tr>
</tbody>
</table>

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 \( k = (d_l \cdots d_0)_{\text{NAF}_w} \).

**Output:** \([k]P \in E(\mathbb{F}_p)\).  

1. Compute the odd multiples \([\pm 3]P, [\pm 5]P, \ldots, [\pm (2^{w-1})]P\). //Use the modification from section 4.2.1.  
2. \((\xi : \eta : 1) \leftarrow [d_l]P; //\text{Precomputed points are represented in } \mathcal{A}\).  
3. Randomly choose \( \lambda \in \mathbb{F}_p^* \).  
4. \( Q \leftarrow (\lambda^2 \xi : \lambda^3 \eta : \lambda); ///\text{Redundant representation of } [d_l]P.\)  
5. \( i \leftarrow l - 1; \)
6. while \( i \geq 0 \) do  
7. \( Q \leftarrow [2]Q; \)
8. if \( d_i \neq 0 \) then  
9. \( Q \leftarrow Q \oplus [d_i]P; \)
10. end if  
11. \( i \leftarrow i - 1; \)
12. end while  
13. return \( Q \) in \( \mathcal{A} \)

<table>
<thead>
<tr>
<th>Field multiplications</th>
<th>( l = 192 )</th>
<th>( l = 224 )</th>
<th>( l = 256 )</th>
<th>( l = 384 )</th>
<th>( l = 521 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overhead</td>
<td>2016M</td>
<td>2331M</td>
<td>2645M</td>
<td>3871M</td>
<td>5182M</td>
</tr>
<tr>
<td></td>
<td>0.24%</td>
<td>0.21%</td>
<td>0.19%</td>
<td>0.13%</td>
<td>0.10%</td>
</tr>
</tbody>
</table>

**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 \( \tilde{E} \) for which there exists an isomorphism

\[
\psi : E(\mathbb{F}_p) \rightarrow \tilde{E}(\mathbb{F}_p)
\]

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:
Curve randomization by curve isomorphisms

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

\[ \psi \downarrow \quad \uparrow \psi^{-1} \]

\[ \tilde{P} \in \tilde{E}(\mathbb{F}_p) \rightarrow [k]\tilde{P} \in \tilde{E}(\mathbb{F}_p) \]

Figure 6.5: Calculation of \([k]P\) by using the isomorphism \(\psi\).

**Proposition 6.4.** Let \(K\) be a field with \(\text{char}(K) \neq 2, 3\). Let \(E\) and \(\tilde{E}\) be elliptic curves over \(K\) given by

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

The curves \(E\) and \(\tilde{E}\) are isomorphic if, and only if, there exists \(u \in K^*\) such that \(\tilde{a} = u^{-4}a\) and \(\tilde{b} = u^{-6}b\). In the affirmative, an isomorphism \(\psi : E(K) \rightarrow \tilde{E}(K)\) is given by

\[ \psi(P) = \begin{cases} (u^{-2}x, u^{-3}y), & P \neq \mathcal{O} \\ \mathcal{O}, & P = \mathcal{O}. \end{cases} \]  \hfill (6.9)

The inverse \(\psi^{-1} : \tilde{E}(K) \rightarrow E(K)\) is given by

\[ \psi^{-1}(P) = \begin{cases} (u^2x, u^3y), & P \neq \mathcal{O} \\ \mathcal{O}, & P = \mathcal{O}. \end{cases} \] \hfill (6.10)

Applying Proposition 6.4 to the case of the NIST curves, we see that a curve

\[ E : y^2 = x^3 - 3x + b \]  \hfill (6.11)

is isomorphic to a curve

\[ \tilde{E} : y^2 = x^3 + \tilde{a}x + \tilde{b} \]  \hfill (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. \( \tilde{P} \leftarrow (u^{-2}x, u^{-3}y); \)
4. \( (\tilde{x}, \tilde{y}) \leftarrow \text{ECMULT}(\tilde{P}, k, \tilde{a}); \)
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 \texttt{ECDEL} and \texttt{ECADD}. The element \( b \) is never used.

Apart from the cost of \texttt{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 \((\kappa_1 - 1)\) 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 \( \frac{1}{p-1} \), 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 \( t(\text{ECMULT}_1) \) 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 \( t(\text{ECMULT}_2) \) denote the average requirement of
Algorithm 8 with
\[(\mathcal{C}^1, \mathcal{C}^2, \mathcal{C}^3) = (\mathcal{J}, \mathcal{J}, A)\]
(the case \(\tilde{a} = -3\)). The average requirement of Algorithm 20 is
\[T_{\text{Alg. 20}} = I + 6M + 3S + \left(1 - \frac{r}{p - 1}\right) \cdot t(\text{ECMULT}_1) + \frac{r}{p - 1} \cdot t(\text{ECMULT}_2),\]
where \(r = 4\) for \(p = p_{224}\) and \(r = 2\) otherwise. This gives the values shown in table 6.10.

<table>
<thead>
<tr>
<th>Field multiplications</th>
<th>(l = 192)</th>
<th>(l = 224)</th>
<th>(l = 256)</th>
<th>(l = 384)</th>
<th>(l = 521)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overhead</td>
<td>1.4%</td>
<td>1.2%</td>
<td>1.0%</td>
<td>0.7%</td>
<td>0.5%</td>
</tr>
</tbody>
</table>

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.
Comparison and conclusion

### Field multiplications:

<table>
<thead>
<tr>
<th>Countermeasure</th>
<th>$l = 192$</th>
<th>$l = 224$</th>
<th>$l = 256$</th>
<th>$l = 384$</th>
<th>$l = 521$</th>
</tr>
</thead>
<tbody>
<tr>
<td>None (algorithm 8)</td>
<td>$M\not\geq 7$</td>
<td>2011</td>
<td>7</td>
<td>2326</td>
<td>7</td>
</tr>
<tr>
<td>Scalar randomization (algorithm 17)*</td>
<td>2336</td>
<td>7</td>
<td>2650</td>
<td>7</td>
<td>2957</td>
</tr>
<tr>
<td>Point randomization by blinding (algorithm 18)*</td>
<td>2031</td>
<td>7</td>
<td>2346</td>
<td>7</td>
<td>2661</td>
</tr>
<tr>
<td>Point randomization by redundancy (algorithm 19)</td>
<td>2006</td>
<td>7</td>
<td>2321</td>
<td>7</td>
<td>2636</td>
</tr>
<tr>
<td>Curve randomization by isomorphism (algorithm 20)</td>
<td>2029</td>
<td>7</td>
<td>2344</td>
<td>7</td>
<td>2658</td>
</tr>
</tbody>
</table>

* Secure against Goubin-type attacks.

### Timings:

<table>
<thead>
<tr>
<th>Countermeasure</th>
<th>$l = 192$</th>
<th>$l = 224$</th>
<th>$l = 256$</th>
<th>$l = 384$</th>
<th>$l = 521$</th>
</tr>
</thead>
<tbody>
<tr>
<td>None (algorithm 8)</td>
<td>7352 $\mu$s</td>
<td>10052 $\mu$s</td>
<td>13698 $\mu$s</td>
<td>34482 $\mu$s</td>
<td>81 ms</td>
</tr>
<tr>
<td>Scalar randomization (algorithm 17)*</td>
<td>8810 $\mu$s</td>
<td>11764 $\mu$s</td>
<td>16250 $\mu$s</td>
<td>38048 $\mu$s</td>
<td>87 ms</td>
</tr>
<tr>
<td>Point randomization by blinding (algorithm 18)*</td>
<td>7490 $\mu$s</td>
<td>10294 $\mu$s</td>
<td>13888 $\mu$s</td>
<td>35087 $\mu$s</td>
<td>82 ms</td>
</tr>
<tr>
<td>Point randomization by redundancy (algorithm 19)</td>
<td>7352 $\mu$s</td>
<td>10101 $\mu$s</td>
<td>13698 $\mu$s</td>
<td>34741 $\mu$s</td>
<td>81 ms</td>
</tr>
<tr>
<td>Curve randomization by isomorphism (algorithm 20)</td>
<td>7380 $\mu$s</td>
<td>10101 $\mu$s</td>
<td>13793 $\mu$s</td>
<td>35087 $\mu$s</td>
<td>81 ms</td>
</tr>
</tbody>
</table>

* Secure against Goubin-type attacks.

Table 6.11: The table shows the average number of field multiplications required ($M$), number of points stored ($\not\geq$) 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 from 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
Chapter 7. Securing an Implementation

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:

\[ \text{DA} := \text{Double-and-add always (Algorithm 11)} \]
\[ \text{WD} := \text{W-double-and-add always (Algorithm 13)} \]
\[ \text{MG} := \text{Montgomery’s ladder algorithm (Algorithm 15)} \]
\[ \text{UA} := \text{Unified addition (Algorithm 16)} \]
\[ \text{AT} := \text{Side channel atomicity} \]
\[ \text{SR} := \text{Scalar randomization (Algorithm 17)} \]
\[ \text{PB} := \text{Point randomization by blinding (Algorithm 18)} \]
\[ \text{PR} := \text{Point randomization by redundancy (Algorithm 19)} \]
\[ \text{CR} := \text{Curve randomization (Algorithm 20)} \]

\((A,1,1)\): We assume unlimited storage available and want security against both FI attacks and Goubin-type attacks. The straightforward choice is \((M_1, M_2) = (\text{MG, PB})\), 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 \(\text{SR}\) is a DPA countermeasure which provides security against both FI attacks \textit{and} Goubin-type attacks. Therefore, \((M_1, M_2) = (\text{AT, SR})\) is the optimal choice. Table 7.1 shows the average number of field operations required by this combined countermeasure.

<table>
<thead>
<tr>
<th></th>
<th>(l = 192)</th>
<th>(l = 224)</th>
<th>(l = 256)</th>
<th>(l = 384)</th>
<th>(l = 521)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Field multiplications</td>
<td>2348M</td>
<td>2662M</td>
<td>2969M</td>
<td>4194M</td>
<td>5505M</td>
</tr>
<tr>
<td>Overhead</td>
<td>16.8%</td>
<td>14.4%</td>
<td>12.5%</td>
<td>8.5%</td>
<td>6.3%</td>
</tr>
</tbody>
</table>

\textbf{Table 7.1}: The table shows the average number of required field multiplications for scalar multiplication with countermeasures \((M_1, M_2) = (\text{AT, SR})\) 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 \((M_1, M_2) = (\text{MG, PR})\). However, the
Combinations of Countermeasures

large overhead of MG makes the combination inferior to \((M_1, M_2) = (AT, SR)\), 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 \(M_1 = AT\). The most efficient DPA countermeasure with security against Goubin-type attacks is PB. Therefore, we get \((M_1, M_2) = (AT, PB)\) to be optimal. Table 7.2 shows the average number of required field multiplications.

<table>
<thead>
<tr>
<th>Field multiplications</th>
<th>(l = 192)</th>
<th>(l = 224)</th>
<th>(l = 256)</th>
<th>(l = 384)</th>
<th>(l = 521)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overhead</td>
<td>2.1%</td>
<td>1.8%</td>
<td>1.6%</td>
<td>1.1%</td>
<td>1.0%</td>
</tr>
</tbody>
</table>

Table 7.2: The table shows the average number of required field multiplications for scalar multiplication with countermeasures \((M_1, M_2) = (AT, PB)\) 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 \((M_1, M_2) = (AT, PR)\). Table 7.3 shows the average number of required field multiplications.

<table>
<thead>
<tr>
<th>Field multiplications</th>
<th>(l = 192)</th>
<th>(l = 224)</th>
<th>(l = 256)</th>
<th>(l = 384)</th>
<th>(l = 521)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overhead</td>
<td>0.85%</td>
<td>0.73%</td>
<td>0.64%</td>
<td>0.44%</td>
<td>0.35%</td>
</tr>
</tbody>
</table>

Table 7.3: The table shows the average number of required field multiplications for scalar multiplication with countermeasures \((M_1, M_2) = (AT, PR)\) and overhead compared to the efficient, non-secure 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 \(M_1 = AT\) or \(M_1 = WD\), because of the need to store the matrix and extra precomputed points respectively. Similarly, we cannot use \(M_2 = PB\), because of the need to store the table of points. Therefore, the optimal choice is \((M_1, M_2) = (MG, SR)\). 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,
we have \( M_1 \neq \text{AT, WD} \), because of the storage requirements. The optimal choice is \((M_1, M_2) = (\text{MG, PR})\). Table 7.5 shows the number of required field multiplications.

<table>
<thead>
<tr>
<th>Field multiplications</th>
<th>(l = 192)</th>
<th>(l = 224)</th>
<th>(l = 256)</th>
<th>(l = 384)</th>
<th>(l = 521)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>3311M</td>
<td>3855M</td>
<td>4399M</td>
<td>6575M</td>
<td>8904M</td>
</tr>
<tr>
<td>Overhead</td>
<td>64.6%</td>
<td>65.7%</td>
<td>66.6%</td>
<td>70.1%</td>
<td>72.0%</td>
</tr>
</tbody>
</table>

**Table 7.5:** The table shows the number of required field multiplications for scalar multiplication with countermeasures \((M_1, M_2) = (\text{MG, PR})\) 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 \((M_1, M_2) = (\text{MG, SR})\). 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 \(M_1 = \text{AT, WD} \). Therefore, the optimal choice is \((M_1, M_2) = (\text{MG, PR})\). 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:

\[
(M_1, M_2) = \begin{cases} 
  \text{(AT, SR)} \\
  \text{(AT, PB)} \\
  \text{(AT, PR)} \\
  \text{(MG, SR)} \\
  \text{(MG, PR)} 
\end{cases}
\]
Comparison and Conclusion

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:**

<table>
<thead>
<tr>
<th>Countermeasure</th>
<th>$l = 192$</th>
<th>$l = 224$</th>
<th>$l = 256$</th>
<th>$l = 384$</th>
<th>$l = 512$</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>$M \star \parallel M \star \parallel M \star \parallel M \star \parallel M \star \parallel$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$(AT,SR)^*$</td>
<td>2011 7 2326 7 2969 15 3866 15 5177 15</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$(AT,PR)^*$</td>
<td>2053 7 2662 7 2982 7 3908 15 5220 15</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$(MG,SR)$</td>
<td>3867 0 4411 0 4955 0 7131 0 9460 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$(MG,PR)$</td>
<td>3311 0 3855 0 4399 0 6575 0 8904 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

* Needs storage for matrix.
** Needs storage for matrix and table of points.

**Overhead:**

<table>
<thead>
<tr>
<th>Countermeasure</th>
<th>$l = 192$</th>
<th>$l = 224$</th>
<th>$l = 256$</th>
<th>$l = 384$</th>
<th>$l = 512$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(AT,SR)$</td>
<td>16.8% 14.4% 12.5% 8.5% 6.3%</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$(AT,PR)$</td>
<td>2.1% 1.8% 1.6% 1.1% 1.0%</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$(MG,SR)$</td>
<td>92.3% 89.6% 87.7% 84.5% 82.7%</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$(MG,PR)$</td>
<td>64.6% 65.7% 66.6% 70.1% 72.3%</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Timings:**

<table>
<thead>
<tr>
<th>Countermeasure</th>
<th>$l = 192$</th>
<th>$l = 224$</th>
<th>$l = 256$</th>
<th>$l = 384$</th>
<th>$l = 512$</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>7352 $\mu$s 10052 $\mu$s 13698 $\mu$s 34482 $\mu$s 81 ms</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$(AT,SR)$</td>
<td>9803 $\mu$s 13333 $\mu$s 17391 $\mu$s 41 ms 93 ms</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$(AT,PR)$</td>
<td>8333 $\mu$s 11299 $\mu$s 15267 $\mu$s 38037 $\mu$s 87 ms</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$(MG,SR)$</td>
<td>8196 $\mu$s 11173 $\mu$s 15037 $\mu$s 37735 $\mu$s 87 ms</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$(MG,PR)$</td>
<td>14285 $\mu$s 19230 $\mu$s 25974 $\mu$s 67 ms 159 ms</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 7.6: The tables show the average number of field multiplications ($M$), number of precomputed points ($\parallel$) 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 \((M_1, M_2) = (\text{MG, SR})\) if storage is limited and \((M_1, M_2) = (\text{AT, SR})\) 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, \(T_{(M_1, M_2)}\) denotes the number of field multiplications required on average by our scalar multiplication scheme with countermeasures \(M_1\) and \(M_2\), while \(T_{\text{IBM}}\) and \(T_{\text{Efficient}}\) 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 implementation 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 support 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 counter-

<table>
<thead>
<tr>
<th>(l = 192)</th>
<th>(l = 224)</th>
<th>(l = 226)</th>
<th>(l = 384)</th>
<th>(l = 521)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(T_{\text{IBM}})</td>
<td>(T_{\text{Efficient}})</td>
<td>(T_{(\text{MG, SR})})</td>
<td>(T_{(\text{AT, SR})})</td>
<td></td>
</tr>
<tr>
<td>(M/\text{Time})</td>
<td>(M/\text{Time})</td>
<td>(M/\text{Time})</td>
<td>(M/\text{Time})</td>
<td></td>
</tr>
<tr>
<td>5056M/15625(\mu s)</td>
<td>5899M/22222(\mu s)</td>
<td>6741M/30769(\mu s)</td>
<td>10112M/80(ms)</td>
<td>13720M/175(ms)</td>
</tr>
<tr>
<td>2011M/7352(\mu s)</td>
<td>2326M/10052(\mu s)</td>
<td>2640M/13698(\mu s)</td>
<td>3866M/34482(\mu s)</td>
<td>5177M/81(ms)</td>
</tr>
<tr>
<td>3867M/14285(\mu s)</td>
<td>4411M/19230(\mu s)</td>
<td>4955M/25974(\mu s)</td>
<td>7131M/67(ms)</td>
<td>9460M/159(ms)</td>
</tr>
<tr>
<td>2348M/9803(\mu s)</td>
<td>2662M/13333(\mu s)</td>
<td>2969M/17391(\mu s)</td>
<td>4194M/41(ms)</td>
<td>5505M/93(ms)</td>
</tr>
</tbody>
</table>

Table 7.7: The table shows the average number of field multiplications and timings of implementations.
Comparison and Conclusion

measure 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 NAFₙ 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
Chapter 8. Results and Recommendations

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 ECDEL 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 \((X_n)_{n=0}^{\infty}\) 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 \((X_n)_{n=0}^{\infty}\) is known as a random process with state space \(\mathcal{S}\).

**Definition A.1** (Memoryless process). The random process \((X_n)_{n=0}^{\infty}\) is a memoryless process if

\[
\begin{align*}
\forall n \in \mathbb{N}_0 \forall i_0, \ldots, i_{n-1} \in \{1, \ldots, k\} \forall i, j \in \{1, \ldots, k\} : \\
P(X_{n+1} = s_j \mid X_0 = s_{i_0}, \ldots, X_{n-1} = s_{i_{n-1}}, X_n = s_i) = \\
P(X_{n-1} = s_j \mid X_n = s_i).
\end{align*}
\]

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 \((X_n)_{n=0}^{\infty}\) is homogeneous if

\[
\begin{align*}
\forall n, n' \in \mathbb{N}_0 \forall i, j \in \{1, \ldots, k\} : \\
P(X_{n+1} = s_j \mid X_n = s_i) = P(X_{n'+1} = s_j \mid X_{n'} = s_i).
\end{align*}
\]

If \(n\) denotes steps in time, we speak of time homogeneity.
Appendix A. Random Processes and Markov Chains

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 \(\mathcal{S} = \{\text{“The car starts”}, \text{“The car doesn’t start”}\}\). 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:

\[
P(X_{n+1} = \text{“The car starts”} \mid X_n = \text{“The car starts”}) = \frac{7}{10}
\]

\[
P(X_{n+1} = \text{“The car doesn’t start”} \mid X_n = \text{“The car starts”}) = \frac{3}{10}
\]

\[
P(X_{n+1} = \text{“The car starts”} \mid X_n = \text{“The car doesn’t start”}) = \frac{4}{10}
\]

\[
P(X_{n+1} = \text{“The car doesn’t start”} \mid X_n = \text{“The car doesn’t start”}) = \frac{6}{10}
\]

Let \(s_1 = \text{“The car starts”}\) and \(s_2 = \text{“The car doesn’t start”}\). 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 \(M = (X_n)_{n \in \mathbb{N}_0}\) 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) \(\forall i \in \{1, \ldots , k\} : \sum_{j=1}^{k} T_{ij} = 1\).
Basic Definitions and Results

\[ \begin{align*} 4 \quad \rightarrow \\ s_1 \quad 7 \quad \rightarrow \\ 3 \quad \rightarrow \\ 6 \quad \rightarrow \\ s_2 \end{align*} \]

**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

\[ \forall n_i, n_j \in N : (n_j, n_i) \in V \iff T_{ij} > 0. \]

**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 \( (X_n)_{n \geq 0} \) with state space \( S = \{ s_1, \ldots, s_k \} \) 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

\[ \mu^{(n)} = (\mu_1^{(n)}, \ldots, \mu_k^{(n)}) = (P(X_n = s_1), \ldots, P(X_n = s_k)), \quad n \in \mathbb{N}, \]

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 \( T \):
Appendix A. Random Processes and Markov Chains

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

$$\mu^{(n)} = \mu^{(0)} T^n. \quad (A.1)$$

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

$$\mu^{(0)} T^n = (\mu^{(0)} T^{n-1}) T$$
$$= \mu^{(n-1)} T$$
$$= \mu^{(n)},$$

because for each $j = 1, \ldots, k$ we have

$$\mu_j^{(n)} = P(X_{n-1} = s_1) P(X_n = s_j \mid X_n = s_1) + \cdots + P(X_{n-1} = s_k) P(X_n = s_j \mid X_n = s_k)$$
$$= \mu_j^{(n-1)} T_{ij} + \cdots + \mu_k^{(n-1)} T_{kj}.$$


\begin{corollary}
The probability of being in state $s_j$ at time $n$ when starting in state $s_i$ is

$$P(X_n = s_j \mid X_0 = s_i) = (T^n)_{ij}.$$

\end{corollary}

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

$$\mu^{(1)} = \mu^{(0)} T$$
$$= (1, 0) \begin{pmatrix} 7 & 3 \\ 0 & 3 \end{pmatrix}$$
$$= \begin{pmatrix} 7/10 & 3/10 \end{pmatrix},$$

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 $s_i \rightarrow s_j$ 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 $s_i$.  

132
Properties

**Definition A.5** (Irreducible chain). A Markov chain with state space \( S = \{s_1, \ldots, s_k\} \) is *irreducible* if \( s_i \rightarrow s_j \) for all \( s_i, s_j \in S \). 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 \( S \) and transition matrix \( T \). The *period* \( d(s_i) \) of a state \( s_i \in S \) is defined as

\[
d(s_i) = \gcd\{n \geq 1 \mid (T^n)_{ii} > 0\}.
\]

If \( d(s_i) = 1 \), we say that \( s_i \) is *aperiodic*. \( M \) is said to be aperiodic if all states in \( S \) 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

\[
1 \in \{n \geq 1 \mid (T^n)_{ii} > 0\} \text{ for } i = 1, 2.
\]

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

\begin{enumerate}
  \item \( \pi_i > 0, i = 1, \ldots, k \) and \( \sum_{i=1}^k \pi_i = 1 \).
  \item \( \pi T = \pi \).
\end{enumerate}

This implies, that if \( \pi \) 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 \( \pi = (\frac{1}{7}, \frac{3}{7}) \) 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

$$V(P, Q) = \sum_{a \in \mathcal{A}} |P(a) - Q(a)|.$$ 

Let $(P_n)_n$ 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 $P_n \overset{V}{\to} Q$.

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 $\pi$ for $M$, and $\mu^{(n)} \overset{V}{\to} \pi$.

**Example A.8.** The Markov chain in the car example is irreducible, aperiodic and has a finite state space. Theorem A.3 says, that

$$\mu^{(n)} \overset{V}{\to} \pi = \left( \frac{4}{7}, \frac{3}{7} \right),$$

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.

<table>
<thead>
<tr>
<th>P-192</th>
</tr>
</thead>
</table>
| \( k \) & 0x 7FFFFFFF FFFFFFFF FFFFFFFF CCEF7C1B 0A35E4D8 DA691418  
| \( x_1 \) & 0x 188DA80E B03090F6 7CBF20EB 43A18800 F4FF0AFD 82FF1012  
| \( y_1 \) & 0x 07192B95 FFC8DA78 631011ED 6E24CDD5 73F977A1 1E794811  
| \( x_2 \) & 0x 7B4603CC 4AC84726 4022B071 44C25277 F2AD8FF8 9224728F  
| \( y_2 \) & 0x 7890050B B4048924 0DEBEC68 5E5B68A9 FE531D5E 9F92E5A2  

Table B.1: Scalar \( k \), base point \( P = (x_1, y_1) \) and value of \([k]P = (x_2, y_2)\) for P-192

<table>
<thead>
<tr>
<th>P-224</th>
</tr>
</thead>
</table>
| \( k \) & 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 EE519EFA 8DA70F8F F7292C0D F5E20B89 668CDDDA  
| \( y_2 \) & 0x D8DF2DB A3C1E407 6EF19DC7 F0DCA56B A5BA9A1E A7FCEA26 CF983DEC  

Table B.2: Scalar \( k \), base point \( P = (x_1, y_1) \) and value of \([k]P = (x_2, y_2)\) for P-224
Appendix B. Test Vectors

### P-256

<table>
<thead>
<tr>
<th>$k$</th>
<th>Ox 7FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF 79DC5E61 7E3192A8</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x_1$</td>
<td>Ox 6B17D1F2 E12C4247 F8BCE6E5 63A440F2 77037D81 2DEB33A0 F4A13945 D898C296</td>
</tr>
<tr>
<td>$y_1$</td>
<td>Ox 4FE342E2 FE1A7F9B 8EE7EB4A 7C0F9E16 2BCE3357 6B315ECE CBB64068 37BF51F5</td>
</tr>
<tr>
<td>$x_2$</td>
<td>Ox 2AFA386B 3F2EDCDB 83F4D83F 8FA3874D 7B74DCB4 54BD644F DD68F3D1 F2DA8DB6</td>
</tr>
<tr>
<td>$y_2$</td>
<td>Ox 72184BE1 CA85634 62B536F1 0852D665 AE8A64FD F1EB8D4C 946AD589 796F729C</td>
</tr>
</tbody>
</table>

**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

<table>
<thead>
<tr>
<th>$k$</th>
<th>Ox 7FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF E3B1A6C0 FA1B9E6F AC0D6D9 245853ED 76760CB5 666294B9</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x_1$</td>
<td>Ox A87CA22 BE8B537 8EB1C71E F320AD74 6E1D3B62 8BA79B98 59F741E0 82542A38 5502F25D BF55296C 3A545E38 72760AB7</td>
</tr>
<tr>
<td>$y_1$</td>
<td>Ox 3617DE4A 96262C6F 5D9E98BF 9292DC29 F8F41DBD 289A147C E9DA3113 B5FCB8C0 A6C81CE 1D7E819D 7A431D7C 90EACE5F</td>
</tr>
<tr>
<td>$x_2$</td>
<td>Ox D36FED39 CA71063A 5163E811 9A37AFF1 0F6B86D5 0FC2F1D3 24238D2B 090D8067 08495505 66396FF5 778738C0 B39B107A</td>
</tr>
<tr>
<td>$y_2$</td>
<td>Ox 46C3E562 B85B82F0D DFACE8F5 32101B4B 82E07DB1 C8FDC36D 1F572843 416840AC DCF2BC1C ED532667 81FCFBA9 739AAE51</td>
</tr>
</tbody>
</table>

**Table B.4:** Scalar $k$, base point $P = (x_1, y_1)$ and value of $[k]P = (x_2, y_2)$ for P-384
<table>
<thead>
<tr>
<th>$k$</th>
<th>$0x$ 000000FF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF 0x 0000C6 85E06B7 0404E9CD 9E3ECE66 2395B442 9C648139 053FB521 F828AF60 6B4D3DBA A14B5E77 EFE75928 FE1DC127 A2FFA8DE 3348B3C1 856A429B F97E7E31 C2E5BD66</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x_1$</td>
<td>$0x$ 00000018 39296A78 9A3BC004 5C8A5FB4 2C7D1BD9 98F54449 579B4468 17AFB017 273E662C 97EE7299 5EF42640 C550B901 3FAD0761 353C7086 A272C240 88BE9476 9FD16650</td>
</tr>
<tr>
<td>$y_1$</td>
<td>$0x$ 0000007C 1BB67BC4 F1A47A2C AB98F683 2FD9681F D803A639 451943B3 5EEB82B7 05FD4132 7338840F 7B531313 F188DE7E 42B46B6 8E0FA5CB 05B53558 C1CA8E31 D783223F</td>
</tr>
<tr>
<td>$x_2$</td>
<td>$0x$ 00000080 F5C012BC C94FE001 953F1E6F 96550AE0 EC2D9950 D5014995 8EE25F55 A0D30EAF 239F0274 00854830 6FCE7EFB 146970BC 87DAC12 D98D9376 DD2E3EBA 550A9CBF</td>
</tr>
<tr>
<td>$y_2$</td>
<td>$0x$ 00000000 F5C012BC C94FE001 953F1E6F 96550AE0 EC2D9950 D5014995 8EE25F55 A0D30EAF 239F0274 00854830 6FCE7EFB 146970BC 87DAC12 D98D9376 DD2E3EBA 550A9CBF</td>
</tr>
</tbody>
</table>

**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

```java
1  interface IFIELD_ELEMENT{
2      public IFIELD_ELEMENT add(IFIELD_ELEMENT val);
3      public int compareTo(IFIELD_ELEMENT val);
4      public boolean equals(Object pObj);
5      public IFIELD_ELEMENT inv();
6      public IFIELD_ELEMENT mul(int n);
7      public IFIELD_ELEMENT mul(IFIELD_ELEMENT val);
8      public IFIELD_ELEMENT negate();
9      public IFIELD_ELEMENT pow(int exp);
10     public IFIELD_ELEMENT shl(int val);
11     public IFIELD_ELEMENT shr(int val);
12     public IFIELD_ELEMENT sub(IFIELD_ELEMENT val);
13     public java.math.BigInteger toBigInteger();
14     public java.lang.String toString();
15  }
```

C.1.2 Implementation of $\mathbb{F}_{p^{192}}$

```java
1  import java.math.BigInteger;
2  import java.lang.Math;
3  import java.util.Random;
4  public final class P192Impl implements IFIELD_ELEMENT{
5      private BigInteger n;
6      private static final BigInteger p192 =
7          new BigInteger("627710173538668076383578942320766616083908700390324961279 ");
8  
9  /**
10   * Constructor
11   *
12   * @param a
13   */
14  public P192Impl(BigInteger a){
15      n = a;
16  }
17  
18  /**
19   * Addition
20   *
21   * @param val
22   */
23  public P192Impl add(IFIELD_ELEMENT val){
24      BigInteger b = val.toBigInteger();
25      BigInteger c = modularAdd(n, b);
26      return new P192Impl(c);
27  }
28  
29  /**
30   * Compare
31   *
32   * @param val
33   */
34  public int compareTo(IFIELD_ELEMENT val){
35      return n.compareTo(val.toBigInteger());
36  }
37  
38  /** Equality testing
39   *
40   * @param pObj
41   */
42  ```
*/
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));
}
/**
 * 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 $\mathbb{F}_{224}$

import java.math.BigInteger;

public final class P224Impl implements IFieldElement{
    private BigInteger n;

    private static final BigInteger p224 =
        new BigInteger("2695994666715063979467015063967557916260026308143510066298811");

    /**
     * Constructor
     * @param a
     */
    public P224Impl(BigInteger a){
        n = a;
    }

    /**
     * Addition
     * @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(IGFieldElement val)
{
    return new P224Impl(n.multiply(val.toBigInt
        eger()).mod(p224));
}

/**
 * Negation
 */
public P224Impl negate()
{
    return new P224Impl(n.negative().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(IGFieldElement val)
{
    BigInteger c = modularSub(n, val.toBigInt
        eger());
    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
 */
C.1.4 Implementation of $\mathbb{F}_{p_{256}}$

```java
import java.math.BigInteger;

public final class P256Impl implements IFieldElement {

    private BigInteger n;

    private static final BigInteger p256 =
        new BigInteger("1157920892136639158845489189441407835823756587050694295063279181484986042008\n        72803218278134276413359253700753333792313620882823773081532681788927708669833447\n        372783172001594334251442435704432852740786038269268767102185068723631453813038\n        392256105240720084805683463570588882269065700301718303608290971058243676254500\n        772633476244448222102372417919234110938084521125262206188046945193082356401814\n        50980827\n    );

    public P256Impl(BigInteger a) {
        n = a;
    }

    public P256Impl add(IFieldElement val) {
        BigInteger c = modularAdd(n.val, i.val);
        return new P256Impl(c);
    }

    public P256Impl compareTo(IFieldElement val) {
        BigInteger c = modularSub(n.val, i.val);
        return n.compareTo(i);
    }

    public P256Impl equals(Object obj) {
        return n.equals(obj);
    }

    public P256Impl inv() {
        return new P256Impl(n.modInverse(p256));
    }

    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.negative().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 sqrt()
{
    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;
}
```

C.1.5 Implementation of $\mathbb{F}_{p^{384}}$

```java
import java.math.BigInteger;

public final class P384Impl implements IFieldElement {
    private BigInteger n;
    private static final BigInteger p384 =
        new BigInteger("39402061963944792122790
        4010014361380507973927046544666794829340
        4245721771496870329047266088258938001861
        606973112319");

    // Constructor
    * @param a
    * @param b
    * @param val

    public P384Impl(BigInteger a) {
        n = a;
    }

    public P384Impl(BigInteger a, BigInteger b) {
        BigInteger c = a.subtract(b);
        if(c.signedValue() == 1) {
            c = c.add(p384);
        }
        return c;
    }

    public P384Impl add(IFieldElement val) {
        BigInteger c =
            modularAdd(n, val.toBigInteger());
        return new P384Impl(c);
    }

    public int compareTo(IFieldElement val) {
        return n.compareTo(val.toBigInteger());
    }

    public boolean equals(java.lang.Object pObj) {
        return n.equals(pObj);
    }

    public P384Impl inv() {
        return new P384Impl(n.modInverse(p384));
    }

    public P384Impl mul(int m) {
        return new P384Impl(n.multiply(BigInteger.valueOf(m)).mod(p384));
    }
}
```
```java
    return new P384Impl(n.multiply(val.toBigInteger()).mod(p384));
}

/**
 * Negation
 */
public P384Impl negate()
{
    return new P384Impl(n.negate().mod(p384));
}

/**
 * Exponentiation
 */
public P384Impl pow(int exp)
{
    return new P384Impl(n.pow(exp).mod(p384));
}

/**
 * Multiplication by power of two
 */
public P384Impl shl(int value)
{
    return new P384Impl(n.shiftLeft(value).mod(p384));
}

/**
 * Division by power of two
 */
public P384Impl shr(int value)
{
    return new P384Impl(n.shiftRight(value).mod(p384));
}

/**
 * Squaring
 */
public P384Impl sgr()
{
    return new P384Impl(n.pow(2).mod(p384));
}

/**
 * Subtraction
 */
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
 */
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
 */
private BigInteger modularSub(BigInteger a, BigInteger b)
```
C.1.6 Implementation of $\mathbb{F}_{p^{21}}$

```java
import java.math.BigInteger;

public final class P521Impl implements IFieldElement{
    private BigInteger n;
    private static final BigInteger p521 =
        new BigInteger("686479766013060971498190",
            0L799001383271789435300143305403594463459,
            18584383397656052122589560614545549772,
            1683113914808660371219879971664381254702,
            8291115057151");

    /** Constructor */
    * @param a
    **/
    public P521Impl(BigInteger a){
        n = a;
    }

    /** Addition */
    * @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 */
    **/
```
```java
* /
112  public P521Impl negate() {
113      return new P521Impl(n.negative().mod(p521));
114  }
115
116  /**
117   * Exponentiation
118   * @param exp
119   */
120  public P521Impl pow(int exp) {
121      return new P521Impl(n.pow(exp).mod(p521));
122  }
123
124  /**
125   * Multiplication by power of two
126   * @param value
127   */
128  public P521Impl shl(int value) {
129      return new P521Impl(n.shiftLeft(value).mod(p521));
130  }
131
132  /**
133   * Division by power of two
134   * @param value
135   */
136  public P521Impl shr(int value) {
137      return new P521Impl(n.shiftRight(value).mod(p521));
138  }
139
140  /**
141   * Squaring
142   */
143  public P521Impl sq2() {
144      return new P521Impl(n.pow(2).mod(p521));
145  }
146
147  /**
148   * Subtraction
149   * @param val
150   */
151  public P521Impl sub(IFieldElement val) {
152      BigInteger c =
153          modularSub(n.val.toBigInteger());
154      return new P521Impl(c);
155  }
156
157  /**
158   * Conversion to BigInteger
159   */
160  public BigInteger toBigInteger() {
161      return n;
162  }
163
164  /**
165   * Conversion to String
166   */
167  public String toString() {
168      return n.toString();
169  }
170
171  /**
172   * Addition modulo p
173   * @param a
174   * @param b
175   */
176  private BigInteger modularAdd(BigInteger a, BigInteger b) {
177      BigInteger c = a.add(b);
178      if(c.bitLength() > 521)
179          c = c.subtract(p521);
180      if(c.compareTo(p521) == 1)
181          c = c.subtract(p521);
182      return c;
183  }
184
185  /**
186   * Subtraction modulo p
187   * @param a
188   * @param b
189   */
190  private BigInteger modularSub(BigInteger a, BigInteger b) {
191      BigInteger c = a.subtract(b);
192      if(c.signum() == -1)
193          return new P521Impl(c);
194  }
```
C.2 Addition and Doubling

```java
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 z1
    * @param y1
    * @param z1_2
    * @param z1_3
    * @param z2
    * @param z2_2
    * @param z2_3
    * @param pq
    * @throws IllegalArgumentException
    */

    static void addPointsJC (IFieldElement x1, IFieldElement y1, IFieldElement z1, IFieldElement zl_2, IFieldElement zl_3, IFieldElement x2, IFieldElement y2, IFieldElement xz, IFieldElement z_2, IFieldElement z_3, IFieldElement [] pq)
    throws IllegalArgumentException {

        if (z1.equals(BigInteger.ZERO)) { // P == O
            pq[0] = x2;
            pq[1] = y2;
            pq[2] = x2;
            pq[3] = x2_2;
            pq[4] = x2_3;
            return;
        }

        if (x2.equals(BigInteger.ZERO)) { // Q==O

            // Temporary variables
            IFieldElement t1, t2, t3, t4, t5, t6, t7;

            t1 = x1.mul(x2_2); /A
            t3 = y1.mul(x2_3); /C
            t2 = x2.mul(x1_2); /B
            t4 = y2.mul(x1_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
    t7 = t2.sqr();    // F^2
    t1 = t6.negative().sub(t4.shl(1)).add(t7);    // z3
    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 z1
     * @param y1
     * @param z2
     * @param y2
     * @param pq
     * @throws IllegalArgumentException
     */

    static void addPointsJCtoJ(IFieldElement x1, IFieldElement y1, IFieldElement z1, IFieldElement z2, IFieldElement y2, IFieldElement pq)
    {
        if (z1.equals(BigInteger.ZERO))    // P = O?
            return;
        if (z2.equals(BigInteger.ZERO))    // Q = O?
            return;

        // Temporary variables
        ifieldElement t1, t2, t3, t4, t5, t6, t7;
        t5 = z2.sqr();
        t1 = x1.mul(t5);    // A
        t5 = t5.mul(z2);    // B
        t4 = y2.mul(z1_3);    // D
        t5 = t2.sub(t1);    // E
        t2 = t4.sub(t3);    // F
        t5 = t1_2;
        t1 = x1;
        t2 = x2;
        t3 = y1;
        t4 = x1;
        t5 = t2;

        if (t5.equals(BigInteger.ZERO) &&
            t2.equals(BigInteger.ZERO))    // P = Q?
            return;

    }  

    t4 = t5.sqr();    // E^2
    t6 = t4.mul(t5);   // E^3

```
static void addPointsJCtoJM ( IFIELD_ELEMENT x1, y1, z1, IFIELD_ELEMENT x2, y2, IFIELD_ELEMENT z2, IFIELD_ELEMENT x3, y3 ) throws ILLEGAL_ARGUMENT_EXCEPTION {  
  if ( t1.equals ( BigInteger.ZERO ) ) {  // P == O?  
    pq[0] = x2;  
    pq[1] = y2;  
    pq[2] = z2;  
  }  
  t5 = t1.sqr();  
  t1 = t1.sub ( t6 ).sub ( t4.sshl ( t1 ) );  // z3  
  t2 = t2.mul ( t4.sub ( t1 ) );  
  t2 = t2.mul ( t3.mul ( t6 ) );  // y3  
  t5 = t5.mul ( z1 );  
  t5 = t5.mul ( z2 );  // z3  

  // Return values  
  pq[0] = t1;  
  pq[1] = t2;  
  pq[2] = t5;  
}  
it ( x2.equals ( BigInteger.ZERO ) ) {  // Q == O?  
  pq[0] = x1;  
  pq[1] = y1;  
  pq[2] = z1;  
  t5 = pq[2].sqr();  
  t5 = pq[3].sqr();  
  t5 = pq[3].mul ( a );  
  return;  
}  
```
\begin{verbatim}
250     t1 = t7.sub(t6).sub(t4.shl(1));  // z3
251     t2 = t2.mul(t4.sub(t1));
252     t2 = t2.sub(t3.mul(t6));  // y3
253     t3 = t5.mul(z1);
254     t3 = t3.mul(z2);  // z3
255
256     t4 = t3.sqr();  // z3^2
257     t4 = t4.sqr();  // z3^4
258     t4 = t4.mul(a);  // a z3^4
259
260     // Return values
261     pq[0] = t1;
262     pq[1] = t2;
263     pq[2] = t3;
264     pq[3] = t4;
265 
266     /**
267     * Add two points in Affine / Jacobian coordinates
268     * and express the result in Jacobian coordinates.
269     * @param z1
270     * @param y1
271     * @param z2
272     * @param y2
273     * @param pq
274     * @throws IllegalArgumentException
275     */
276     static void addPointsAtoJ (IFieldElement x1,
277                               IFieldElement y1,
278                               IFieldElement x2,
279                               IFieldElement y2,
280                               IFieldElement[] pq,
281                               IFieldElement one)
282              throws IllegalArgumentException {
283     if (x1 == null) { // P == O?
284         pq[0] = x2;
285         pq[1] = y2;
286         pq[2] = one;
287         return;
288     }
289
290     IFieldElement t1, t2, t3, t4, t5, t6, t7;
291
292     t5 = x2.sub(x1);  // E
293     t2 = y2.sub(y1);  // F
294     if (t5.equals(BigInteger.ZERO) &&
295         t2.equals(BigInteger.ZERO)) { //P=Q?
296         // Should use double instead
297         throw new IllegalArgumentException();
298     }
299
300     t4 = t5.sqr();  // E^2
301     t6 = t4.mul(t5);  // E^3
302     t4 = x1.mul(t4);  // AE^2
303     t7 = t2.sqr();  // F^2
304     t1 = t7.sub(t6).sub(t4.shl(1));  // z3
305     t2 = t2.mul(t4.sub(t1));
306     t2 = t2.sub(t1.mul(t6));  // y3
307     t3 = t5;  // z3
308
309     // Return values
310     pq[0] = t1;
311     pq[1] = t2;
312     pq[2] = t3;
313 }
\end{verbatim}
static void addPointsAJtoJ (IFeldElement 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 (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 = x2.sqr();  
    t1 = x1.mul(t5); // A  
    t5 = t5.mul(x2);  
    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)); // z3  
    t2 = t2.mul(t4.shl(1));  
    t2 = t2.sub(t3.mul(t6)); // y3  
    t3 = t5.mul(x2); // 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 z1  
* @param y1  
* @param z2  
* @param y2  
* @param a  
* @param b  
* @param one  
* @throws IllegalArgumentException  
*/  
static void addPointsAJtoJM (IFeldElement x1, IFieldElement y1, IFieldElement x2, IFieldElement y2, IFieldElement[] pq, IFieldElement[] one) throws IllegalArgumentException {  
    if (x1 == null) { // P == O?  
        pq[0] = x2;  
        pq[1] = y2;  
}
```java
pq[2] = x2;
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)); // z3
if t2.equals(BigInteger.ZERO) // Q==0?
    if (x1==null) { // P = O?
pq[0] = x2;
pq[1] = y2;
pq[2] = one;
pq[3] = a;
return;
}
if(x2 == null){ // Q==0?
```
```java
508     pq[0] = x1;
509     pq[1] = y1;
510     pq[2] = one;
511     pq[3] = a;
512     return;
513 }
514
515     IFieldElement t1, t2, t3, t4, t5, t6, t7;
516     t5 = x2.sub(x1); // E
517     t2 = y2.sub(y1); // F
518     if(t5.equals(BigInteger.ZERO) &&
519         t2.equals(BigInteger.ZERO)) // P=Q?
520         // Should use double instead
521         throw new IllegalArgumentException();
522     t4 = t5.sqr(); // E^2
523     t6 = t4.mul(t5); // E^3
524     t4 = x1.mul(t4); // AE^2
525     t7 = t2.sqr(); // F^2
526     t1 = t7.sub(t6).sub(t4.shl(1)); // z3
527     t2 = t2.mul(t4.sub(t1));
528     t2 = t2.sub(y1.mul(t6)); // y3
529     t3 = t5; // z3
530     t4 = t3.sqr(); // z3^2
531     t4 = t4.sqr(); // z3^4
532     t4 = t4.mul(a); // az3^4
533     // Return values
534     pq[0] = t1;
535     pq[1] = t2;
536     pq[2] = t3;
537     pq[3] = t4;
538 }
539
540 /* Add two affine points without doing inversion. */
541 /* @param x1 */
542 /* @param y1 */
543 /* @param x2 */
544 /* @param y2 */
545
546     static void addPointsA (IFieldElement x1,
547         IFieldElement y1,
548         IFieldElement x2,
549         IFieldElement y2,
550         IFieldElement[] pq)
551         throws IllegalArgumentException {
552         if (x1 == null) { // P == O?
553             pq[0] = x2;
554             pq[1] = y2;
555             return;
556         }
557         if (x2 == null) { // Q == O?
558             pq[0] = x1;
559             pq[1] = y1;
560             return;
561         }
562         if (x1.equals(x2) &&
563             (y1.equals(y2) || y1.equals(y2.negate())))
564             // P = P^{-1}
565             throw new IllegalArgumentException();
566     IFieldElement d =
567         (y2.sub(y1).mul((x2.sub(x1)).inv()));
568     pq[0] = d.sqr().sub(x1).sub(x2);
569     pq[1] = d.mul(x1.sub(pq[0])).sub(y1);
570     /*
571     */
572 */
```
```java
static void addPointsA_NoInversions (IFieldElement x1, IFieldElement y1, IFieldElement x2, IFieldElement y2, IFieldElement e)

throws IllegalArgumentExceptionException {  
if (x1 == null) { // P == 0?
    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 IllegalArgumentExceptionException();

//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);
}

//Temporary values
IFieldElement t1, t2, t3, t4, t5, t6;
t1 = b.shl(2); //4b
t1 = t1.negate(); //−4b
t2 = x1.mul(z2); //z2z
  t1 = t1.mul(t2); //−2z1z2
  t3 = x1.mul(z2); //z2z
  t4 = x2.mul(z1); //z2z1
  t5 = t3.add(t4); //z1z2 + z2z1
  t1 = t1.mul(t5); //−2z1z2(z1z2 + z2z1)
t5 = x1.mul(x2); //z1z2
  t1 = t2.negate();
t2 = t2;
716 \begin{align*}
t 1 &= y.2.\text{mul}(x.1); \quad // y.2z1 \\
t 2 &= y.1.\text{mul}(x.2); \quad // y.1z2 \\
t 1 &= z.1.\text{sub}(t.2); \quad // A \\
t 3 &= x.2.\text{mul}(x.1); \quad // x.2z1 \\
t 4 &= x.1.\text{mul}(x.2); \quad // x.1z2 \\
t 3 &= z.3.\text{sub}(t.4); \quad // B \\
\text{if}(t.1.\text{equals}(\text{BigInteger}.\text{ZERO}) &&
\text{t.3.\text{equals}(\text{BigInteger}.\text{ZERO}))} \quad // P = Q^2
\text{throw new IllegalArgumentException();}
\end{align*}

725 \begin{align*}
t 5 &= x.1.\text{mul}(x.2); \quad // x.1z2 \\
t 6 &= t.1.\text{sqr}(); \quad // A^*2 \\
t 6 &= 16.\text{mul}(t.5); \quad // A^*2z1z2 \\
t 7 &= t.3.\text{sqr}(); \quad // B^*2 \\
t 8 &= t.7.\text{mul}(t.3); \quad // B^*3 \\
t 6 &= 16.\text{sub}(t.8); \quad // A^*2z1z2-B^*3 \\
t 7 &= t.7.\text{sub}(t.6); \quad // B^*2z1z2-C \\
t 7 &= t.11.\text{mul}(t.7); \quad // A(B^*2z1z2-C) \\
t 4 &= t.7.\text{sub}(t.18.\text{mul}(t.2)); \quad // Y3 \\
t 9 &= 18.\text{mul}(t.5); \quad // Z3 \\
\end{align*}

740 \begin{align*}
\text{pq}[0] &= t.3; \\
\text{pq}[1] &= t.4; \\
\text{pq}[2] &= t.5; \\
\end{align*}

743 \begin{align*}
\end{align*}

744 */
745 */
746 static void addPointsP (IFieldElement x1, 747 IFieldElement y1, 748 IFieldElement z1, 749 IFieldElement x2, 750 IFieldElement y2, 751 IFieldElement z2, 752 IFieldElement[] pq)
750 throws IllegalArgumentException {
751 // Temporary variables
752 IFieldElement t1, t2, t3, t4, t5, t6, t7, t8;
753
static void doublePointJMtoJ (IFieldElement x1, IFieldElement y1, IFieldElement z1, IFieldElement az1_4, IFieldElement[] pp) {
    // Temporary variables
    IFieldElement t1, t2, t3, t4, t5, t6, t7;
    t1 = y1.sqrt();
    t2 = x1.shl(2).mul(t1); //A
    t3 = x1.sqrt();
    t3 = t3.add(t3).add(t3);
    t5 = t3.mul(t2.shl(1)).sub(t1); //y3
    t6 = y1.shl(1).mul(z1); //z3
    t4 = t3.sqrt(); //B/2
    t4 = t4.sub(t2.shl(1)); //z3
    t7 = t1.sqr().shl(3); //C
    t4 = t4.sqr(); //B^2
    t4 = t4.sub(t2.shl(1)); //z3
    t5 = t3.mul(t2.sub(t4)).sub(t1); //y3
    t6 = y1.shl(1).mul(z1); //z3
    t7 = t1.sqr().shl(3); //az1_4
    // Return values
    pp[0] = t4;
    pp[1] = t5;
    pp[2] = t6;
    pp[3] = t7;
}

static void doublePointJtoA (IFieldElement x1, IFieldElement y1, IFieldElement az1_4, IFieldElement[] pp) {
    // Temporary variables
    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)); //z3
    t5 = t3.mul(t2.sub(t4)).sub(t1); //y3
    t6 = y1.shl(1).mul(z1); //z3
    t7 = t1.sqr().mul(az1_4); //az1_4
    // Return values
    pp[0] = t4;
    pp[1] = t5;
    pp[2] = t6;
    pp[3] = t7;
}

}
// 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
 t4 = t3.sqr(); // B^2
 t4 = t4.sub(t2.shl(1)); // z^3
 t5 = t3.mul(t2.sub(t4)).sub(t1); // y^3
 t6 = y1.shl(1); // z^3
 t7 = t1.shl(1).mul(a); // az^3^4

// Return values
pp[0] = t4;
pp[1] = t5;
pp[2] = t6;
pp[3] = t7;
}

return;

// Temporary variables
IFieldElement t1, t2, t3, t4, t5, t6,

t1 = y1.sqr(); // y^1^2
t2 = x1mul(t1); // A
t3 = y1.shl(1); // z^3
t4 = t1.shl(2); // z^3^2 = 4y^1^3
 t5 = t4.shl(1);
t5 = t5.mul(y1); // z^3^3 = 8y^1^3
 t6 = x1.sqr();
t6 = t6.add(t6.add(t6)).sub(three); // B
t1 = t6.sqr(); // B^2
t1 = t1.add(t2.shl(1)); // z^3
 t2 = t6.mul(t2.sub(t1));
t2 = t2.shl(1).mul(y1); // y^3

// 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 Chudnovsky Jacobian coordinates.
* @param x1
* @param y1
* @param pp
*/

static void doublePointAtJ (IFieldElement x1,
  IFieldElement y1, IFieldElement zero,
  IFieldElement one,
  IFieldElement three,
  IFieldElement [] pp) throws

if (x1 == null) { // P == O?
  pp[0] = one;
  pp[1] = one;
  pp[2] = zero;
  pp[3] = zero;
  pp[4] = zero;
}


/**
* Double a point in affine coordinates. Express the
* result in Jacobian coordinates.
* @param x1
* @param y1
* @param pp
*/

static void doublePointAtJC (IFieldElement x1,
  IFieldElement y1, IFieldElement zero,
  IFieldElement one,
  IFieldElement three,
  IFieldElement [] pp) throws

if (x1 == null) { // P == O?
  pp[0] = one;
  pp[1] = one;
  pp[2] = zero;
  pp[3] = zero;
  pp[4] = zero;
}
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;

// Temporary variables
IFieldElement t1, t2, t3, t4, t5;

// 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){

/**
 * Double a point in affine coordinates.
 * @param x
 * @param y
 * @param a
 * @param pp
 * @throws IllegalArgumentException
 */
static void doublePointA (IFieldElement x1,
    IFieldElement y1,
    IFieldElement a,
```java
    IFriendElement[] pp = {
        if (x1 == null) { // P == O
            pp[0] = pp[1] = null;
            return;
        }
        if (y1.equals(y1.negate()))
            throw new IllegalArgumentException();

        // Temporary variables
        IFriendElement 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);
        pp[0] = t4;
        pp[1] = t6;
    }

    static void doublePointA_NoInversions(IFriendElement x1, IFriendElement y1, IFriendElement a, IFriendElement d, IFriendElement[] pp) {
        if (x1 == null) { // P == O
            pp[0] = pp[1] = null;
            return;
        }
        if (y1.equals(y1.negate()))
            throw new IllegalArgumentException();

        IFriendElement 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);
        pp[0] = t4;
        pp[1] = t6;
    }

    /**
     * Double a point in affine coordinates
     * with no inversions.
     * @param x1
     * @param y1
     * @param a
     * @param d
     * @param pp
     * @throws IllegalArgumentException
     */
    ```
* Montgomery trick (in general form by Breit and Joyce).

```java
static void doublePointMontgomeryP (IFieldElement x1,
  IFieldElement x2, IFieldElement a, IFieldElement b,
  IFieldElement p)
```

throws IllegalArgumentExceptionException

```java
// Temporary values
IFieldElement t1, t2, t3, t4, t5;
```

```java
t1 = x1.sqr(); // z1^2
```

```java
t2 = t1.sqr(); // z1^2
```

```java
t3 = x1.mul(t2); // z1^2
```

```java
t3 = b.mul(t3); // b2z1^2
```

```java
// t2 = t2.mul(a); // a*z1^2
```

```java
t2 = t2.negate();
```

```java
t4 = t2; // t2 = t2
```

```java
t2 = t2.shl(1);
```

```java
t2 = t2.add(t4); // a*z1^2
```

```java
t4 = t1.sub(t2); // z1^2 - a*z1^2
```

```java
t4 = t4.sqr(); // (z1^2 - a*z1^2)^2
```

```java
t5 = x1.mul(t3); // z1b1^2
```

```java
t5 = t5.shl(3); // 8z1b1^2
```

```java
pp[0] = t4.sub(t5); // z3
```

```java
pp[1] = t4.shl(2); // z3
```

```java
}
```

```java
/**
 * Double a point in affine coordinates using Montgomery trick and give the result in projective coordinates (in general form by Breit and Joyce).
 * @param x1
 * @param x2
 * @param a
 * @param b
 * @param pp
 */
```

```java
static void doublePointMontgomeryAtoP (IFieldElement x1,
  IFieldElement a,
  IFieldElement b,
  IFieldElement p)
```

```java
// Temporary values
IFieldElement t1, t2, t3, t4, t5;
```

```java
t1 = x1.sqr(); // z1^2
```

```java
t2 = t1.sub(a).sqr(); // (z1^2 - a)^2
```

```java
t3 = b.mul(t1); // b^2 z1^2
```

```java
t3 = t3.shl(3); // 8b^2 z1^2
```

```java
pp[0] = t2.sub(t3); // (z1^2 - a)^2 - 8b^2 z1^2
```

```java
t1 = t1.mul(t1); // z1^3
```

```java
t2 = t1.negate();
```

```java
t3 = t2;
```

```java
t2 = t2.shl(1);
```

```java
t2 = t2.add(t3); // -3z1
```

```java
t1 = t1.add(t4).add(b);
```

```java
pp[1] = t1.shl(2);
```

```java
}
```
1175 * Double a point in affine coordinates to
1176  projective coordinates.
1177 * 
1178 * @param x1
1179 * @param y1
1180 * @param z1
1181 */
1182 
1183 static void doublePointAtOP (IFieldElement x1,
1184  IFieldElement y1,
1185  IFieldElement z1,
1186  IFieldElement three,
1187  IFieldElement [] pp){
1188    // Temporary variables
1189    IFieldElement t1, t2, t3, t4;
1190    t1 = x1.sqrt(); /*x1^2*/
1191    t1 = t1.shl(1).add(t1); /*3x1^2*/
1192    t1 = t1.sub(three); /*A*/
1193    t2 = y1.sqrt(); /*y1^2*/
1194    t3 = t3.shl(2); /*A^2*/
1195    t4 = t4.sub(t3.shl(1)); /*D*/
1196    t3 = t3.shl(t4); /*4C-D*/
1197    t3 = t3.mul(t3); /*A(4C-D)*/
1198    t3 = t3.shl(t2.sqr().shl(3)); /*YS*/
1199    t1 = t2.shl(3).mul(y1); /*Z3*/
1200    t2 = y1.shl(1).mul(t4); /*X3*/;
1201    pp[0] = t2;
1202    pp[1] = t3;
1203    pp[2] = t1;
1204    }
1205 */
1206 */
1207  
1208  /**********************************************************************************/
1209  * Unified addition
1210  **********************************************************************************/
1211  */
1212  */
1213  */
1214  * Adds two points in projective coordinates using
1215  * the unified addition formula (in general form
1216  * by Briet and Joyce).
1217  */
1218 */
1219 * @param x1
1220 * @param y1
1221 * @param z1
1222 * @param x2
1223 * @param y2
1224 * @param z2
1225 * @param pq
1226 */
1227 
1228 static void addPointsUnifP (IFieldElement x1,
1229  IFieldElement y1,
1230  IFieldElement z1,
1231  IFieldElement x2,
1232  IFieldElement y2,
1233  IFieldElement z2,
1234  IFieldElement [] pp) {
1235    throws IllegalArgumentException
1236    if (y1.equals(y2.negate())) //P=Q
1237      throw new IllegalArgumentException();
1238    // Temporary values
1239    IFieldElement t1, t2, t3, t4, t5, t6;
1240    t1 = x1.mul(x2); /*A*/
1241    t2 = x1.mul(x2); /*B*/
1242    t3 = x2.mul(x1); /*C*/
1243    t4 = y1.mul(x2); /*D*/
1244    t5 = y2.mul(x1); /*E*/
1245    t6 = t2.add(t3); /*F*/
1246    t4 = t4.add(t5); /*G*/
1247    t5 = t6.sqr(); /*F^2*/
1248    t5 = t5.sub(t2.mul(t3)); /*F^2-BC*/
1249  
1250  */
1251  */
1252  }
C.3 Scalar Multiplication without SPA/DPA Countermeasures

C.3.1 Original IBM Test Implementation

```java
import java.math.BigInteger;

public final class ECCHM {
    private static final BigInteger THREE =
        BigInteger.valueOf(3);

    /**
     * Scalar multiplication using addition-subtraction;
     * see IEEE P1363-2004: A.10.3.
     * @param p_z
     * @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],
                    P[1].negate(),m,R);
            kp[0] = R[0];
            kp[1] = R[1];
        }
    }
```
```java
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(p_y))]
    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;
 * @param p
 * @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.negative()))
    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 ECCM implements IECCMultiply{
    /**
     * Scalar multiplication using addition–subtraction;
     * see IEEE P1363-2004: A.10.3.
     * @param p
     * @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[] P = { x1, y1 });
```
C.3.3 Efficient Implementation

```java
public fieldElement[] 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.pointsA(Q[0],Q[1],P[0],
        P[1],negate(),Q);
    } else {
        Q[0] = Q[0];
        Q[1] = Q[1];
    }
}

public void multiplyPoint (fieldElement x1, 
fieldElement y1, 
fieldElement zero, 
fieldElement one, 
fieldElement three, 
int[] naf, int w, 
BigInteger k, 
fieldElement[] kp) 
throws IllegalArgumentExceptionException{
    int limit = ((int)Math.pow(2,w-1)-1);
    HashMap<Integer,fieldElement[]> precomputed = 
    new HashMap<Integer,fieldElement[]>();
    //Use simultaneous inversions in
    //the precomputations
    Auxiliary.precompAffine(x1,y1,w,three.negate(),
    precomputed);
    //Use the modification to reduce the number of
    //initial doublings.
    fieldElement[] 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) {
        c--;
    }
    //If k_l < limit, something can be saved.
    if (k_l < limit){
        q = new fieldElement[3];
        int l = BigInteger.valueOf(k_l).bitLength();
        int t = (l-1) * Math.pow(2,l-1)*
        ((int)Math.pow(2,w-1)-1);
        fieldElement[] p1 = precomputed.get(limit);
        fieldElement[] p2 = precomputed.get(t);
    }
```

Appendix C.

Source Code
C.4 Scalar Multiplication with SPA Countermeasures

C.4.1 Double-and-add Always

```java
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 {

    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};
```
```java
36     for (int i = k.bitLength() - 2; i >= 0; i --)
37         Addition.doublePointJ(q_0[0], q_0[1], q_0[2],
38             zero, one, q_0);
39         if (k.testBit(i)) {
40             Addition.addPointsAJtoJ(x1, y1, q_0[0],
41                 q_0[1], q_0[2],
42                 q_0, one);
43         } else {
44             Addition.addPointsAJtoJ(x1, y1, q_0[0],
45                 q_0[1], q_0[2],
46                 q_1, one);
47         }
48         }
49         Auxiliary.jacobianToAffine(q_0, kp);
50     }

C.4.2 W-double-and-add Always

import java.math.BigInteger;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.Map;

public final class ECCWtA implements IECCMultiply {
    /*
     * Scalar multiplication using wNAF method
     * (w-double-and-one-add-always) and mixed coordinates.
     * @param xi
     * @param yi
     * @param a
     * @param m
     * @param k
     * @param kp
     */

    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[]>();
            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[1] != 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

public final class ECCMontgomery implements ECCMultiply {
    
    /**
     * Scalar multiplication using Montgomery's ladder;
     * @param x1
     * @param y1
     * @param a
     * @param b
     * @param one
     * @param k
     * @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, b2);

        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, p2);
            } 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
C.4.4 Unified Addition

```java
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 xi
     * @param y1
     * @param zero
     * @param one
     * @param three
     * @param naf
     * @param w
     * @param k
     * @param kp
     * @throws IllegalArgumentException
     *
     * @throws IllegalArgumentException
     *
     * @return
     */

    public void multiplyPoint (IFieldElement x1,
                               IFieldElement y1,
                               IFieldElement zero ,
                               IFieldElement one ,
                               IFieldElement three ,
                               int[] naf, int w,
                               BigInteger k,
                               IFieldElement[] kp)
    {
        // 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 /|p\m/ 3P,...,|p \m(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[] {x1++,y1++,one });

            // Calculate /-i|P
            precomputed.put(-i,new IFieldElement[] {x1++,y1++,one });
        }

        IFieldElement[] start =
            precomputed.get(naf[naf.length -1]);

        IFieldElement[] q =
            new IFieldElement[] {start[0],
                                 start[1],
                                 start[2]};
    }
}
```
C.4.5 Side channel Atomicity

```java
import java.math.BigInteger;
import java.lang.Math;
import java.util.ArrayList;

public final class ECCAtomicity implements IECCMultiply {
    private static final int[][] A_low =
        AtomicityMatrix.getMatrix();

    public static int[][] getMatrix()
    {
        return new int[][] {
            // Define double
            new int[][]{{4,3,3,5,1,4,4,4,1,4},
                        {4,4,5,5,4,4,6,6,6,4,5},
                        {6,2,3,5,4,5,5,6,6,6,6},
                        {4,2,2,4,4,4,6,6,4,5},
                        {6,4,1,6,6,6,6,7,6,6,6},
                        {8,5,5,1,7,8,7,7,7,2,3} },
            new int[] {4,4,4,4,4,4,4,4,1,6,6} },
            new int[] {6,5,6,2,4,6,8,8,4,5,5},
            new int[] {5,10,4,6,10,4,5,6,5,5,5},
            new int[] {6,11,3,7,1,5,7,8,5,7,7},
            new int[] {6,6,4,8,6,4,8,8,4,8,8},
            new int[] {3,7,3,8,7,3,8,8,7,8,7},
            new int[] {8,7,7,9,10,11,9,9,8,7,7},
            new int[] {5,5,8,9,5,5,6,4,5,8,8},
            new int[] {7,7,8,9,7,9,4,4,2,6,6},
            new int[] {1,4,4,1,1,9,4,5,1,5,5},
            new int[] {5,4,5,8,5,5,7,8,5,7,7},
            new int[] {2,6,7,2,2,5,7,7,7,2,5}};
    }
}
```
FieldElement y1,
FieldElement zero,
FieldElement one,
FieldElement three,
int[] saf, int w,
BigInteger k,
FieldElement[] kp)
throws IllegalArgumentException
{
// Initialise temporary variables
FieldElement[] R = new FieldElement[12];
for (int i = 0; i < 12; i++)
    R[i] = one;

// Precomputations
int limit = ((int)Math.pow(2, w-1)) - 1;
HashMap<Integer, FieldElement[]> precomputed =
    new HashMap<Integer, FieldElement[]>((3 * limit);

Auxiliary.precompAffine(x1, y1, three, negate(),
    precomputed);
precomputed.put(0, new FieldElement[]{x1, y1});

FieldElement[] start =
    R[1] = start[0];
    R[2] = start[1];
    int s = 1;
    int m = 0;
for (int i = naf.length - 2; i >= 0; i--)
    if (k[i] = naf[i];
        FieldElement[] p = precomputed.get(naf[i]);
        R[10] = p[0];
        m = (s = i)? 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[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][7].negate();
R[A_low[m][8] =
    R[A_low[m][9].add(R[A_low[m][10]]);
}
Auxiliary.jacobianToAffine(new FieldElement[]{
    R[1][R[2][R[3]],
    kp);
public void multiplyPoint (IFieldElement x1,  
  IFieldElement y1,  
  IFieldElement zero,  
  IFieldElement one,  
  IFieldElement three,  
  int[] na, int w,  
  BigInteger k,  
  IFieldElement[] kp)  
  throws IllegalArgumentException {
// Get the random pair (Q,Q^-1). The method  
  // getRandon 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 3P, \ldots, |\pm \frac{2^w-1-1}{2}P|P 
  int limit = ((int)Math.pow (2,w-1))-1;  
  HashMap<BigInteger,IFieldElement> precomputed =  
    new HashMap<Integer,IFieldElement> ();  
  Auxiliaty.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 = na[naf.length - 1];  
  // Get the value of kappa
  int kappa = 1;  
  int c = naf.length - 2;
  while (naf[c] == 0){  
    kappa++;  
  }  
  c--;  

  // if k <= limit, something can be saved.
  if (k < limit){
    q = new IFieldElement [3];  
    // Number of bits in k
    int l = BigInteger.valueOf(k.bitLength());  
    int m = (k-1) * Math.pow (2, l-1)  
      * Math.pow (2, w-1);  
    IFieldElement[] pl = precomputed.get (limit);  
    IFieldElement[] p2 = precomputed.get (m);  
    Addition.addPointsAtol (p1[0],p1[1],  
      p2[0],p2[1],q,one);  
    for (int i=1; i <= kappa - w1-1;i++)  
      Addition.doublePointJ (q[0],q[1],q[2],  
        zero,one,q);  
    s = c;  
  }

  // if k == limit, nothing can be saved.
  else{
    IFieldElement[] temp = precomputed.get (k);  
    q = new IFieldElement [3];  
    Addition.doublePointAtol (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.addPointsAtol (pre[0],pre[1],  
        q[0],q[1],q[2],  
        q,one);  
    }
  }
C.5.2 Point Randomization by Redundancy

```java
1 public final class ECCPointRandomization implements IECCMultiply {
2    /**
3     * Scalar multiplication using wNAF method and mixed
4     * coordinates. DPA countermeasure: Point
5     * randomization by redundancy.
6     *
7     * @param x1  
8     * @param y1  
9     * @param a   
10     * @param m   
11     * @param k   
12     * @param kp  
13     * @throws IllegalArgumentException 
14     */
15    public void multiplyPoint ([IFieldElement x1,  
16        [IFieldElement y1,  
17        [IFieldElement zero,  
18        [IFieldElement one,  
19        [IFieldElement three,  
20        int[] naf, int w,  
21        BigInteger k,  
22        IFeldElement[] kp)
23        throws IllegalArgumentException {
24            // Precomputations
25            int limit = ((Int)Math.pow(2, w-1)-1);
26            HashMap<Integer, IFieldElement[]> precomputed =
27                    new HashMap<Integer, IFieldElement[]>((3*limit ));
28            Auxiliary.precompAffine(x1, y1, w, true, negate(),  
29                    precomputed);
30            // Randomize the representation in
31            // Jacobian coordinates
32            IFieldElement rfe =
33                    RandomFieldElement.getRandom(k, bitLength());
34            IFieldElement rfe_2 = rfe.sqr();
35            IFieldElement rfe_3 = rfe_2.mul(rfe);
36            // Use the modification to reduce the number of
37            // initial doublings.
38            IFieldElement[] q;
39            int s;
40            int k_1 = naf[naf.length-1];
41            // Get the value of kappa
42            int kappa = 1;
43            int c = naf.length-2;
44            while (naf[c]==0) {
45                kappa++;
46                c--;
47            }
48            //If k_1 < limit something can be saved.
49            if (k_1 < limit) {
50                q = new IFieldElement[3];
51                // Number of bits in k_1
52                int t = Biginteger.valueOf(k_1).bitLength();
53                int l = (int)Math.pow(2, l-1)*
54                        ((int)Math.pow(2, w-1)-1);
55                IFieldElement[] p1 = precomputed.get(limit);
56                IFieldElement[] p2 = precomputed.get(t);
```
C.5.3 Curve Randomization by Isomorphisms

```java
import java.math.BigInteger;
import java.lang.Math;
import java.util.HashMap;
import java.util.Map;

public final class ECCurveRandomization implements IECCMultiply{

    private static final ECCM ncm = new ECCM();

    /** 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;
        ...
```
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 =
    new IFieldElement[2 * limit];

int s;
int k_1 = naf.length - 1;

// Get the value of kappa
int kappa = 1;

int c = naf.length - 2;
while (naf[c] == 0) {
    kappa++;
    c--;
}

// If k_1 < limit something can be saved.
if (k_1 < limit) {
    q_m = new IFieldElement[4];
    // Number of bits in k_1
    int l = BigInteger.valueOf(k_1).bitLength();
    int t = (k_1 - (int) Math.pow(2, l - 1)) *
            ((int) Math.pow(2, w - 1)) + 1;
    IFieldElement[] p1 = precomputed.get(limit);
    IFieldElement[] p2 = precomputed.get(t);
    Addition.addPointsAtJacobian(p1[0], p1[1], p2[0],
                                p2[1], a_prime, q_m, one);
    for (int i = l; i <= kappa - w - 1; i++) {
        Addition.doublePointJacobian(q_m[0], q_m[1],
                                     q_m[2], q_m[3],
                                     q_m);
        s = c;
    }
}

// If k_1 = limit nothing can be saved.
else {
    IFieldElement[] temp = precomputed.get(k_1);
    q_m = new IFieldElement[4];
    Addition.addPointsAtJacobian(temp[0], temp[1],
                                  a_prime, q_m);
    s = naf.length - 3;
}

// Perform the scalar multiplication
IFieldElement[] q_m_1 = new IFieldElement[3];
for (int i = s; i >= 0; i--) {
    if (naf[i] == 0) {
        // Double to Jacobian coordinates.
        Addition.doublePointJacobian(q_m[0], q_m[1],
                                     q_m[2], q_m[3],
                                     q_m);
        }
C.6 Scalar Multiplication with SPA & DPA Countermeasures

C.6.1 Montgomery’s Ladder Algorithm & Point Randomization by Redundancy

```java
import java.math.BigInteger;

//Precompute the base point
IFieldElement[] pre =
precomputed.get(naf[i]);

//Express the result of the addition in
//modified Jacobian coordinates
//to get a more efficient doubling,
Addition.addPointsAHJM(pre[0], pre[1],
q_0, q_1, a_prime, q_m, one);
else
Addition.doublePointJM(q_m[0], q_m[1],
q_0, q_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[0] = kp[0].mul(mu_2);
kp[1] = kp[1].mul(mu_3);
kp[2] = kp[2];
kp[3] = kp[3];
}
```

```java
//Randomize the representation in
//projective coordinates
IFieldElement rfe =
RandomFieldElement.getRandom(k.bitLength());
IFieldElement rdx = x1.mul(rfe);
IFieldElement rndy = y1.mul(rfe);
FieldElement[] p1, p2;
p1 = new FieldElement[] {rndx, rfe};
p2 = new FieldElement[2];
```

```java
//First doubling is affine -> projective
Addition.doublePointMontgomeryP(x1, one, a, b, p2);
```
```java
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

```
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;
import java.util.Random;

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 m,
                               IFieldElement k,
                               IFieldElement kp)
            throws IllegalArgumentException {
        // Initialize temporary variables
        IFieldElement[] R = new IFieldElement[12];
```

```bash
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 m=0;
for(int i=naf.length-2;i>=0;i--){
    int k = naf[i];
    IFieldElement[] p = precomputed.get(naf[i]);
    R[10] = p[0];
    m = (s==1)? 0 : m+1;
    int u = Auxiliary.phi(k_i);
    s = (int)((u==0)? Math.floor(m/7) :
              Math.floor(m/18));
    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.addPointsJacOQ(r[2], r[3], R[1], R[2],
                       R[3], q, one);
Auxiliary.jacobianToAffine(q,kp);
```
```java
for (int i = 0; i < 12; i++)
    R[i] = 0n6;

// Precomputations
int limit = (((int) Math.pow(2, w - 1)) - 1);
HashMap<Integer, IFieldElement[]> precomputed =
   new HashMap<>();
Auxiliary.precomputedAffine(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[0] = start[0].mul(rfe_2);
R[1] = start[1].mul(rfe_3);
R[3] = rfe;
int s = 1;
int m = 0;
for (int i = naf.length - 2; i >= 0; i--) {
    int k = naf[i];
    IFieldElement[] p =
       precomputed.get(naf[i]);
    R[0] = p[0];
    R[1] = 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.
}
```

C.7 Auxiliary Methods

```java
import java.util.HashMap;
import java.util.Map;
import java.util.ArrayList;
import java.math.BigInteger;

public final class Auxiliary {
    static int[] getNAF(BigInteger n) {
        return getWNAF(n, 2);
    }

    /* Calculation of the non-adjacent form.
     * @param n
     * @param w
     */
```

```java
import java.util.List;
import java.math.BigDecimal;

public static int[] getWNAF(BigInteger n, int w) {
    List<Integer> NAF = new ArrayList<>();
    int m = 0;
    int[] NAFT = n.toByteArray();
    for (int i = 0; i < w; i++) {
        int t = n.bitCount() - 1;
        int phi = n.bitCount() - 1 - t;
        int s = (i == 0) ? 0 : m + 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.
        // Calculate the non-adjacent form.
        NAF.add(s);
    }
    return NAF;
}
```
static int getWNAP(BigInteger n, int w)
{
    ArrayList<BigInteger> ns =
        new ArrayList<BigInteger>();
    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;
}

public static void simInv(IFieldElement[] a, IFieldElement[] b){
    int j = a.length;
    IFieldElement[] c = new IFieldElement[];
    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>=0;i--){
        b[i] = u.mul(c[i-1]);
        u = u.mul(a[i]);
    }
    b[0] = u;
}

static void precompAffine(IFieldElement x1, IFieldElement y1, int w, IFieldElement a, HashMap<Integer, IFieldElement[]>> precomputed)

    // Temporary 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 Sp
    y[1] = temp[1]; // y-coordinate of Sp
    IFieldElement[] d;
    IFieldElement[] e;

    */
    * Simultaneous inversion in F_p.
    * @param a
    * @param b
    */

    // Precomputation in affine coordinates using
    // simultaneous inversion.
    * Returns [1/P, [3/P, ..., [2^{w-1}-1]/P.
    * @param x
    * @param y
    * @param a
    * @param precomputed
    */
```java
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
        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 \[2^{s-1}P, \ldots, 2^{s-3+2^{2+i}}P, 2^{2^{i+1}}P\]
    int k = 0;
    for (int j = pw + 1; j <= t - 1; j += 2)
        Addition.addPointsA_NonInversions(x[j - pw - 1],
                                            y[j - pw - 1],
                                            x[pw - 1],
                                            y[pw - 1],
                                            e[k],
                                            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-adding always (Okeya & Takagi).
    * @param n
    * @param w
    */
    static int[] getWNAF Dummy(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(new BigInteger[]{
                        n_i.intValue(),
                        n % n_i.intValue(),
                        n >> w	n
                    });
                    n = n % n_i.intValue();
                    n = n >> 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.

* @param x1
* @param y1
* @param a
* @param precomputed
*
* static void precompAffineWithEven(IFnFieldElement x1, 240
*                 IFnFieldElement y1, 241
*                 IFnFieldElement a, 242
*                 Int w, 243
*                 HashMap<Integer, 244
*                 IFnFieldElement[]> precomputed) { 245

  // Temporary arrays for the coordinates of the precomputed points.
  // precomputed points.
  IFnFieldElement[] x = 261
    new IFnFieldElement[((int)Math.pow(2, w-1))]; 262
  IFnFieldElement[] y = 263
    new IFnFieldElement[((int)Math.pow(2, w-1))]; 264
  IFnFieldElement[] temp = new IFnFieldElement[2]; 265

  // Addition.
  doublePointA(x1, y1, a, temp); 266

  x[0] = x1; 267
  y[0] = y1; 268
  x[1] = temp[0]; // x-coordinate of 2p 269
  y[1] = temp[1]; // y-coordinate of 2p 270

  for (int i = 1; i < w-2; i++) { 271
    int s = ((int)Math.pow(2, i-1)) + 1; 272
    int pw = ((int)Math.pow(2, i)); 273
    int h = ((int)Math.pow(2, i+1)); 274

    // Use simultaneous inversion
    IFnFieldElement d = new IFnFieldElement[pw]; 275
    IFnFieldElement e = new IFnFieldElement[pw]; 276

    for (int k = 0; k < pw-1; k++) { 277
      d[k] = x[|pw-1], sub{|k|}; 278
      d[pw-1] = y[|pw-1], shr{1}; 279
      simInv(d, e); 280

      // [2^w+i]P
      int k = 0;
      for (int j = pw+1; j <= h; j++) { 282
        if (j > pw) { 283
          Addition 284
            addPointsA_NoInversion( 285
              x[j - pw - 1], 286
              y[j - pw - 1], 287
              x[pw-1], 288
              y[pw-1], 289
              e[k], temp); 290
          x[j - 1] = temp[0]; 291
          y[j - 1] = temp[1]; 292
          k++; 293
        }

        Addition 294
          doublePointA_NoInversion( 295
            x[|pw-1], 296
            y[|pw-1], 297
            a, e[|pw-1], 298
            temp); 299

        x[h - 1] = temp[0]; 300
        y[h - 1] = temp[1]; 301
      }
    }
  }

  // Return the precomputed points in the supplied HashMap.
  int limit = ((int)Math.pow(2, w-1)); 302
  for (int i = 1; i < limit; i++) { 303
    precomputed.put(i, new IFnFieldElement[] { 304
      x[i - 1], y[i - 1] }); 305
    precomputed.put(-i, new IFnFieldElement[] { 306
      x[i - 1], 307
      y[i - 1].negate() }); 308
  }

  // Use simultaneous inversion
  IFnFieldElement d = new IFnFieldElement[pw]; 309
  IFnFieldElement e = new IFnFieldElement[pw]; 310

  for (int k = 0; k < pw-1; k++) { 311
    d[k] = x[|pw-1], sub{|k|]; 312
    d[pw-1] = y[|pw-1], shr{1}; 313
    simInv(d, e); 314

    // [2^w+i]P
    int k = 0;
    for (int j = pw+1; j <= h; j++) { 316
      if (j > pw) { 317
        Addition 318
          addPointsA_NoInversion( 319
            x[j - pw - 1], 320
            y[j - pw - 1], 321
            x[pw-1], 322
            y[pw-1], 323
            e[k], temp); 324
        x[j - 1] = temp[0]; 325
        y[j - 1] = temp[1]; 326
        k++; 327
      }

      Addition 328
        doublePointA_NoInversion( 329
          x[|pw-1], 330
          y[|pw-1], 331
          a, e[|pw-1], 332
          temp); 333

      x[h - 1] = temp[0]; 334
      y[h - 1] = temp[1]; 335
    }
  }

  // Return the precomputed points in the supplied HashMap.
  int limit = ((int)Math.pow(2, w-1)); 336
  for (int i = 1; i < limit; i++) { 337
    precomputed.put(i, new IFnFieldElement[] { 338
      x[i - 1], y[i - 1] }); 339
    precomputed.put(-i, new IFnFieldElement[] { 340
      x[i - 1], 341
      y[i - 1].negate() }); 342
  }
```

```
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;  
}

static void jacobianToAffine(IFieldElement[] j, 
    IFieldElement[] a) {
    if (j[2].equals(BigInteger.valueOf(0)))  
        a[0] = null;  
    a[1] = null;  
    return;  
}

static void getAffine(IFieldElement x1, 
    IFieldElement x2, 
    IFieldElement x, 
    IFieldElement y, 
    IFieldElement a, 
    IFieldElement b, 
    IFieldElement[] r) {
    if (x2.equals(BigInteger.ZERO)) //Q=0?  
        throw new IllegalArgumentException();

    //Temporary variables  
    IFieldElement t1, t2, t3, t4, t5, t6, t7;
    t1 = x1.mul(x1.inv());

t2 = x2.mul(z2.inv());
t3 = t1.mul(x).add(a);
t4 = t1.add(x);
t3 = t3.mul(t4); // (z3z1 - 3)(z3z1)
t4 = x.sub(t1);
t4 = t4.sqr();
t4 = t4.mul(t2); // 2b(z3z1 - 3)(z3z1) -

r[0] = t1;
r[1] = t3;
}

/**
 * Calculation of affine coordinates from
 * projective ones.
 * @param p
 * @param a
 * @param sigma
 * @param k
 * @throws IllegalArgumentException
 */
static void projectToAffine(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


Bibliography


Bibliography


