UNIVERSITY OF CALGARY

# Fast Algorithms for Arithmetic on Elliptic Curves over Prime Fields

by

Nicholas T. Sullivan

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE

CROSS-DISCIPLINARY DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF MATHEMATICS AND STATISTICS

and

DEPARTMENT OF COMPUTER SCIENCE

CALGARY, ALBERTA

January, 2007

**THE UNIVERSITY OF CALGARY**

**FACULTY OF GRADUATE STUDIES**

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies for acceptance, a thesis entitled "Fast Algorithms for Arithmetic on Elliptic Curves over Prime Fields" submitted by Nicholas T. Sullivan in partial fulfillment of the requirements for the degree of MASTER OF SCIENCE.

Supervisor, Dr. R. Scheidler
Department of Mathematics and Statistics

Dr. M. Bauer
Department of Mathematics and Statistics

Co-supervisor, Dr. M. J. Jacobson
Department of Computer Science

Dr. J. Aycock
Department of Computer Science

Dr. A. O. Fapojuwo
Department of Electrical and Computer Engineering

Date

# Abstract

We present here a thorough discussion of the problem of fast arithmetic on elliptic curves over prime order finite fields. Since elliptic curves were independently proposed as a setting for cryptography by Koblitz [53] and Miller [67], the group of points on an elliptic curve has been widely used for discrete logarithm based cryptosystems. In this thesis, we survey, analyse and compare the fastest known serial and parallel algorithms for elliptic curve scalar multiplication, the primary operation in discrete logarithm based cryptosystems. We also introduce some new algorithms for the basic group operation and several new parallel scalar multiplication algorithms. We present a mathematical basis for comparing the various algorithms and make recommendations for the fastest algorithms to use in different circumstances.

# Acknowledgements

I would like to thank my supervisors, Drs. Renate Scheidler and Michael J. Jacobson, who have done more for me than I could have asked for. I would also like to acknowledge the glorious invention of cheese and all the mammals that help make dairy possible. Oh, and Dave Lowry, the music of Pirates are Pussies, the city of Toronto, the city of Calgary, Tubby Dog, the Ship and Anchor, The Bakery/Old School, Francine and $, the Cynar club, the discrete math seminar, toques, the town of Laramie, the purple Intrepid, the other purple Intrepid, Gull Lake Summer Village, Michael Scott, Arby's customers, Ralph Bucks, the laziness-enabling number 3 bus, Spilker'n'co., Edward Luistro and his neverending quest, CPU's pizza, midnight chinese buffets, Richard Guy, Pieter and his beard, a ghost, Redmonton and its artery the Albertabahn, Mike Vernon, everybody whose birthday or name I've ever forgotten, various belt buckles, Google the noun, Google the verb, pint day, the chemical properties of water, Alan the LaTeX guru, Carolyn, Tom, Grace, Gramps, moonlighting, beauty and other abstract ideas, authors everywhere and most especially of all, you.

# Table of Contents

# List of Tables

# List of Figures

# List of Algorithms

# Chapter 1

# Introduction

Cryptography was revolutionized by Diffie and Hellman in 1976 [27] with two concepts, public key cryptography (PKC), and their eponymous key exchange protocol. PKC and the Diffie-Hellman key exchange protocol have since become pillars of modern commercial cryptography. PKC provides a means for encryption and authentication and the Diffie-Hellman key exchange protocol allows the use of symmetric cryptography without the need for costly and slow secure communication channels to exchange keys. The security of Diffie and Hellman's method for exchanging keys relies on a mathematical *one-way function*, a function that is easy to compute but difficult to invert. An example of a one-way function is exponentiation of integers modulo a prime. As far as we know, it is easy to compute $g^x \bmod p$, but difficult to extract $x$ from $g^x$ in this setting. This extraction problem is called the *discrete logarithm problem* (DLP). Diffie and Hellman's one-way function relates closely to the DLP and can be inverted by solving it. If we assume that solving the Diffie-Hellman problem requires a lot of computation, the one way function is secure.

The concept of computational difficulty can be defined formally, and the amount of work required to solve a problem can be calculated explicitly. Research has found that the DLP on the group of integers modulo a prime can be solved in sub-exponential time using the general number field sieve [61]. According to the current state of knowledge, the problem is difficult, but not as difficult as a problem that can only be solved in exponential time.

In 1985, Koblitz [53] and Miller [67] independently proposed *elliptic curve cryptography* (ECC), an approach to public key cryptography based on the group of points on an elliptic curve over a finite field. The group of points on an elliptic curve also exhibits a discrete logarithm problem. After twenty years of research, the fastest algorithm for solving the DLP in this group still has exponential running time. This means that the same level of security can be achieved with smaller keys compared to the DH key exchange protocol, making ECC preferable for devices with constrained resources such as PDAs, pagers, smart cards and cell phones. A smaller key also means faster and more efficient computation. Furthermore, if a fast solution for the DLP in the integers modulo a prime is found, ECC will possibly still be secure.

Efficient elliptic curve arithmetic is critical for ECC. We mention here some ECC protocols and outline the requisite elliptic curve operations that are needed for them. We also describe the operations needed for the cryptographic primitives based on operations in the group of points on an elliptic curve over a finite field.

Notationally, the group under consideration is a generic additive group. Group elements are represented by capital letters and integer scalar multipliers by lower case letters. The group operation applied to two points $P$ and $Q$ is denoted by $P + Q$. The scalar multiplication operation, or $\underbrace{P + P + \cdots + P}_{n \text{ times}}$ is denoted by $nP$.

The operations we are mainly concerned with are unknown point scalar multiplication (computing $nP$ where both $n$ and $P$ are not known in advance), known point scalar multiplication (computing $nP$ where $n$ is unknown and $P$ is known) and known multiplier scalar multiplication (computing $nP$ where $n$ is known and $P$ is unknown). We study algorithms specific to each of these cases in Chapter 3.

A public key encryption scheme allows two parties to communicate securely. One popular encryption scheme is ECIES [9]. In this scheme, encryption requires the computation of one unknown point scalar multiplication and one known point scalar multiplication. Decryption requires one known multiplier scalar multiplication. PSEC [33] is another encryption scheme. In this protocol, encryption requires the computation of one unknown point scalar multiplication and one known point scalar multiplication. Decryption requires a known multiplier scalar multiplication and a known point scalar multiplication.

Key establishment algorithms allow two parties to establish a common secret for use in symmetric cryptography. ECDH [96] is the elliptic curve variant of the Diffie-Hellman key agreement protocol. Each party requires the computation of one known point scalar multiplication and one unknown point scalar multiplication. Another popular key establishment algorithm is ECMQV [59]. For this protocol, each party requires the ability to compute both known point scalar multiplication and unknown point scalar multiplication.

A signature scheme allows one party to digitally sign a piece of data, providing authentication and non-repudiation. ECDSA [46] is the standard digital signature scheme using elliptic curves. Signature generation requires a known point scalar multiplication and verification requires the computation of $aP + bQ$ where $P$ is known, $Q$ is unknown and $a, b$ are unknown.

The elliptic curve scalar multiplication operations are the most time consuming parts of each of the above protocols. In order to implement these protocols efficiently, it is important that unknown point, known point and known multiplier scalar multiplication are all computed as efficiently as possible.

## 1.1 Contributions of the Thesis

The focus of this thesis is efficient arithmetic on the group of points on an elliptic curve over a prime field with applications to ECC. Elliptic curve arithmetic is a comparatively new and very active area of research with a large body of literature. The purpose of this thesis is twofold. First, it provides a comprehensive survey of known techniques for elliptic curve arithmetic, including a description and mathematical analysis. Second, it introduces several new algorithms for elliptic curve arithmetic. The intention is to determine the most efficient algorithms for calculations on elliptic curves in different situations with a thorough analysis and comparison of the latest techniques and the newly introduced ideas.

The specific focus of this thesis is elliptic scalar multiplication in software. We cover addition, doubling and tripling for elliptic curves over prime fields only and use these operations to determine the cost of various serial and parallel scalar multiplication algorithms. This analysis could be translated for non-prime field of interest such as binary fields or optimal extension fields. In this thesis, only prime fields are examined because they are considered the fastest to implement in software. The parallel algorithms examined in Chapter 4 are parallel at the level of point addition and we do not examine algorithms that are parallel at the finite field operation level. We consider many sequential algorithms in the unknown point, known point and known multiplier cases, and parallel algorithms in the unknown point case. Since some fast algorithms are suceptible to side-channel attack, a brief overview side-channel resistant methods and simultaneous scalar multiplication are presented in Chapter 5.

The main contribution of this thesis is an analysis and comparison of the fastest algorithms for elliptic curve scalar multiplication over prime fields. The results provide a thorough comparison of such algorithms under reasonable and standard assumptions. This analysis is novel and constitutes a significant tool for determining the most efficient scalar multiplication algorithm for any given setting in elliptic curve cryptography. This thesis provides the most extensive and careful comparison of the fastest known serial algorithms. The work on parallel algorithms is the only such analysis and provides the first estimates of the speedup that can be obtained by this type of parallelization.

In addition, a number of novel algorithms are presented, including several parallel algorithms for elliptic curve scalar multiplication. A new modification of the point tripling algorithm is presented that allows the double-base chain algorithm (Section 3.2.6) to be the fastest algorithm not requiring storage space. In Chapter 4, we introduce and analyze a number of new parallel algorithms. We present a new parallel algorithm used to precompute points and we introduce several left-to-right and right-to-left algorithms for scalar multiplication as well as a double-base method for scalar multiplication. The algorithms provide a modest speedup over the single processor scalar multiplication algorithms, and could be useful in certain settings where time is at a premium such as large-scale encryption. The new algorithms could be useful in non-constrained implementations and are shown to be faster than the existing algorithms in certain situations.

### 1.1.1 Methodology

A methodology for determining the efficiency of an algorithm is a necessary first step in order to make valid comparisons between different algorithms that perform the same operation. Algorithms for elliptic curves can be classified as either high-level or low-level. Specifically, high-level algorithms operate with the elliptic curve group operations and low-level algorithms deal with finite field arithmetic. The focus of this thesis is high-level algorithms for elliptic curve operations. Low level algorithms are discussed in Appendix A.

For the operations in the field, the analysis will pertain to software implementations. In order to maintain flexibility, we will assume as little as possible about the hardware. We will use constants to represent the time it takes to compute field operations such as addition, multiplication and inversion. The relative times for these constants are based on past implementations, e.g. Brown *et al.* [16].

Elliptic curve operations such as point doubling, addition and tripling are computed using formulas involving finite field elements. The analysis of elliptic curve operations is based on the number of finite field operations required to evaluate the formulas. Elliptic curve scalar multiplication is a compound operation, computed using a sequence of elliptic curve operations. To analyze an elliptic curve scalar multiplication algorithm, the expected (or average) number of elliptic curve operations needed for the algorithm is computed. Once this is determined, we choose a specific representation for the group elements. The number of field operations to compute these operations on points in the chosen representation is then identified. The scalar multiplication algorithm cost formulas are combined with the group operation choices, resulting in a cost formula in terms of finite field multiplications, squarings and inversions. These formulas are determined for elliptic curves over a set of primes called the NIST primes, which are used in practice.

The costs of different scalar multiplication algorithms are made comparable by substituting the relative speeds of squaring and inversion into the cost formulas to obtain a cost in terms of only finite field multiplications. In our context, squaring is slightly faster than multiplication and inversion is much slower than multiplication. By expressing their costs in terms of the number of multiplications, we can directly compare the various scalar multiplication algorithms.

Analyses based on operation counts provide a relatively solid mathematical picture of the efficiency of the various elliptic curve algorithms. This analysis will provide a good measure for the computational requirements of existing implementations and also good predictions for future implementations. They can also be used to provide estimates for the time it would take to perform the algorithms on any system with any parameters. A new implementation of these algorithms is not included as it is beyond the scope of the thesis.

### 1.1.2 Thesis Outline

In Chapter 2, we introduce the theory of elliptic curves. This chapter provides the background needed to define the group of rational points of an elliptic curve over a finite field. The relevant properties of this group are described and formulas for the group law are presented. For the rest of the thesis, we consider elliptic curves over prime fields only. We present explicit algorithms for computing the group operation for elliptic curves over prime fields using different coordinate systems. We also present some new explicit algorithms for point tripling.

In Chapter 3, we focus on the predominant operation in elliptic curve cryptography: scalar multiplication. The standard double-and-add method borrowed from modular exponentiation is presented as well as more efficient techniques. The algorithms studied here fall into the categories of unknown point algorithms, known point algorithms and known multiplier algorithms. The unknown point algorithms include binary NAF, window NAF, sliding window, fractional window, and double-base number systems. The known point algorithms are the windowing and comb methods. The known multiplier algorithms are based on addition chains. A novel analysis is done to determine the average computational cost of these algorithms and the storage space required. The algorithms from this section can all be found in the literature.

In Chapter 4, we present parallel algorithms for scalar multiplication. A parallel computing framework is introduced and several parallel algorithms are analyzed and compared with respect to effective computation cost and the communication time between processors. Many of the algorithms presented in this section are new. The algorithms introduced include a new parallel precomputation algorithm, a new modification of the $p^{th}$ order binary algorithm, several new algorithms for right-to-left and left-to-right parallel scalar multiplication, a parallelization of windowing from the literature, and a new algorithm based on double-base $n$-chains and Montgomery's ladder. We find that a small speed improvement can be obtained by parallelization, depending on the cost of communication between processors.

In Chapter 5, we summarize the results of the thesis. Further topics such as side channel attacks, multiple point multiplication and elliptic curves over binary fields are briefly discussed and directions for further work are proposed.

Finally, two appendices are included that deal with extra material. In Appendix A, we introduce finite fields as well as algorithms for arithmetic in prime fields. In Appendix B, we present all the tables from Chapters 3 and 4 that were not immediately needed.

# Chapter 2

# Elliptic Curve Basics

This chapter provides an introduction to elliptic curves as mathematical objects. We develop the theory of elliptic curves, define the group of points on an elliptic curve and provide explicit algorithms for the operation on this group. We focus on the computational aspects of elliptic curve arithmetic that will be useful for cryptographic applications.

An elliptic curve over a given field is defined by an equation called a *Weierstraß equation*. A point on an elliptic curve is a solution to the Weierstraß equation over the given field. The set of points on an elliptic curve can be made into an Abelian group. The group operation is usually described algebraically, with formulas derived from the geometry of the curve, but it can also be described graphically in some circumstances.

We can identify different curves that generate isomorphic groups of points using the notion of curve isomorphism. Two curves that generate the same group can be thought of as interchangeable. This interchangeability allows for specialized equations for elliptic curves over certain fields. In particular, curve isomorphism allows curves over fields with characteristic two (*binary curves*) to be represented by a *simplified Weierstraß equation* that is used to simplify the rules for the group operation. The same can be done for curves over fields of odd prime characteristic (*prime curves*) and curves of characteristic zero. The two types of curves that are most useful in cryptography are binary and prime curves. Elliptic curves over prime fields are the focus of this thesis.

Many cryptographic protocols (Diffie-Hellman [27], ElGamal [32]) are generic in the sense that they can be adapted to use any Abelian group. Elliptic curve cryptosystems are specific cases of these generic cryptosystems that use the group of points on an elliptic curve over a finite field. In order for a group to be useful in this setting, it must somehow be "difficult" in a computational sense to compute the inverse operation of exponentiation in the group. The group must be finite but large enough so that it is intractable to compute discrete logarithms with generic algorithms (see Pollard [84]) and the group needs a large cyclic subgroup in order to prevent Pohlig-Hellman attacks (see Pohlig and Hellman [83]). It will be shown that both prime curves and binary curves have these desirable properties. Moreover, for correctly chosen prime curves and binary curves there is no known sub-exponential algorithm for solving the discrete logarithm problem, making both types of curves excellent candidates for cryptographic applications.

The formulas for performing the group operation on points in affine or projective coordinates over binary or prime fields are derived from the geometric definition of the field operation. Performing this group operation on a computer requires the formulas to be translated into a sequence of operations in the base field. Efficient algorithms are developed for elliptic curve point addition and doubling in affine and projective coordinate systems. Additionally, other coordinate systems based on projective coordinates (Jacobian [18], Chudnovsky-Jacobian [18], Lopez-Dahab [66]) allow for alternative implementations of elliptic curve arithmetic. In certain applications it can be more efficient to use different coordinate systems simultaneously. Algorithms for mixed-coordinate operations were developed in order to accommodate this need and are presented in this chapter.

The algorithms put forth in this chapter for the group of points on an elliptic curve over a prime field are the basic building blocks of the scalar multiplication algorithms in Chapter 3.

## 2.1 Elliptic Curves

Elliptic curves are mathematical objects that arise naturally in branches of mathematics including algebraic geometry and number theory. An elliptic curve is a specific case of a non-singular algebraic curve. For a broad introduction to the study of algebraic curves see Fulton [34].

An elliptic curve is defined in terms of a Weierstraß equation and a base field. The points that satisfy the equation are called rational points on the curve. As we will see, these points form a group with a little help, and are often referred to as the group of points on an elliptic curve.

In this section, we present the generalized Weierstraß equation and describe the limitations to be applied in order to define a genuine elliptic curve. An equivalence relation can be defined on the set of elliptic curves, thus providing a mechanism to derive simplified Weierstraß equations for elliptic curves over certain fields. We also briefly discuss alternative forms of an elliptic curve.

### 2.1.1 Weierstraß Equation

We begin with the definition of an elliptic curve.

**Definition 2.1.1.** *Let $K$ be a field, $\overline{K}$ its algebraic closure. An* elliptic curve *$E$ over*

*$K$ is denoted by $E/K$ and is given by the* Weierstraß equation

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

*where $a_1, a_2, a_3, a_4, a_6 \in K$, and for each point $(x, y)$ with coordinates in $\overline{K}$ satisfying Equation 2.1 and the partial derivatives $2y + a_1x + a_3$, and $3x^2 + 2a_2x + a_4 - a_1y$ do not vanish simultaneously at that point.*

The criteria that both partial derivatives do not vanish simultaneously means that the curve is *non-singular*. A method for testing a curve for non-singularity involves a quantity called the *discriminant* of the curve. Let

$$b_2 = a_1^2 + 4a_2, \quad b_4 = a_1a_3 + 2a_4, \quad b_6 = a_3^2 + 4a_6, \text{ and}$$
$$b_8 = a_1^2a_6 - a_1a_3a_4 + 4a_2a_6 + a_2a_3^2 - a_4^2.$$

The *discriminant* of the curve $E/K$, denoted by $\Delta$, is defined to be

$$\Delta = -b_2^2b_8 - 8b_4 - 27b_6^2 + 9b_2b_4b_6. \tag{2.2}$$

The discriminant is useful because $\Delta = 0$ if and only if $E/K$ is non-singular.

The object of interest for this chapter is the set of $L$-rational points associated with an elliptic curve $E/K$ with $K \subseteq L \subseteq \overline{K}$. This set corresponds closely to the set of solutions to the Weierstraß equation defining $E/K$. Included in the set are the points with coordinates in $L$ that satisfy the Weierstraß equation and an additional point denoted by $\infty$. This point can be thought of as being infinitely far up the $y$-axis, so that it intersects any line parallel with the $y$-axis. This point corresponds to a point in projective space that is not found in affine space and is necessary for the group law. This is clarified in the next section when coordinate systems are discussed; for now, $\infty$ will be treated as just another point.

**Definition 2.1.2.** *If $L$ is an extension of $K$ or $K$ itself, then the set of* $L$-rational points *over $E/K$ is defined to be the set of all points $(x, y) \in L \times L$ that satisfy Equation 2.1 along with the point $\infty$. This set is denoted $E(L)$.*

The set of $K$-rational points is also called the set of *rational points* on a curve. The sets of $L$-rational points are very important because they have a group structure that is the setting for elliptic curve cryptography.

### 2.1.2 Simplified Weierstraß Equations

It is useful to define an equivalence relation on the set of elliptic curves over a given field. This equivalence relation will also be shown to be a bijection between the sets of points on the curves. In Section 2.3.5, we show that two equivalent elliptic curves have isomorphic groups of points.

**Definition 2.1.3.** *Two curves $E/K$ (with variables $x, y$) and $E'/K$ (with variables $x', y'$) are* isomorphic over $K$ *if and only if there exist constants $r, s, t \in K$ and $u \in K^*$ such that the change of variables*

$$(x, y) \leftarrow (u^2x' + r, u^3y' + su^2x' + t) \tag{2.3}$$

*transforms the equation of $E/K$ into the equation of $E'/K$. Such a transformation is called an* admissible change of variables *or an* isomorphism *from $E$ to $E'$.*

An isomorphism of curves can also be used to define a bijection of sets between the sets of $L$-rational points of two curves in the following way. Suppose that the change of variables $(x, y) \leftarrow (u^2x' + r, u^3y' + su^2x' + t)$ transforms the equation of the elliptic curve $E/K$ into the equation for the elliptic curve $E'/K$ for $r, s, t \in K$

and $u \in K^*$ where $L$ is a field with $K \subseteq L \subseteq \overline{K}$. Then the map

$$\phi : E(L) \to E'(L)$$

defined by

$$\phi(a, b) = (u^2 a' + r, u^3 b' + s u^2 a' + t)$$

is a bijection with

$$\phi^{-1}(a', b') = \left( \frac{a - r}{u^2}, \quad \frac{b - t - s u^2 a'}{u^3} \right),$$

sending $\infty$ to $\infty$.

With this equivalence relation on elliptic curves, the next step is to find canonical forms of the Weierstraß equation with fewer parameters that can cover all equivalence classes of curves. For characteristic other than 2 or 3, the substitution

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

is an admissible change of variables with $u = 1$, $r = (a_1^2 + 4a_2)/12$, $s = a_1/2$ and $t = a_3/2$. It transforms Equation (2.1) from a long form Weierstraß equation to one of the form

$$E' : (y')^2 = (x')^3 + ax' + b, \tag{2.4}$$

where $a, b \in K$ with discriminant $\Delta = -16(4a^3 + 27b^2) \neq 0$.

For curves of characteristic 2 and 3, there are different admissible changes of variable depending on the *supersingularity* of the curve. Supersingular elliptic curves are curves for which the cardinality of the set of rational points over any extension field is not divisible by the characteristic of the field. Supersingular curves are less secure than non-supersingular curves in some cryptographic applications (see

Galbraith [35]) and desirable in others, and we will not discuss them in detail in this thesis.

If the characteristic of $K$ is 2, and $a_1 = 0$, then the curve is supersingular and the admissible change of variables

$$(x, y) = (x' + a_2, y')$$

leads to

$$E' : (y')^2 + cy' = (x')^3 + ax' + b, \tag{2.5}$$

where $a, b, c \in K$ with discriminant $\Delta = c^4 \neq 0$. If the characteristic of $K$ is 2 and $a_1 \neq 0$, then the curve is non-supersingular and the admissible change of variables

$$(x, y) = \left( x'a_1^2 + \frac{a_3}{a_1}, \quad a_1^3 y' - \frac{a_1^2 a_4 + a_3^2}{a_1^3} \right)$$

leads to

$$E' : (y')^2 + x'y' = (x')^3 + a(x')^2 + b, \tag{2.6}$$

where $a, b \in K$ with discriminant $\Delta = b \neq 0$.

If $K$ has characteristic 3, then there are also two cases to consider. If $a_1^2 = -a_2$, then the curve is supersingular and the admissible change of variables

$$(x, y) = (x', y' + a_1 x' + a_3)$$

leads to

$$E' : (y')^2 = (x')^3 + ax' + b, \tag{2.7}$$

where $a, b \in K$ with discriminant $\Delta = -a^3 \neq 0$. If $a_1^2 \neq -a_2$, then the curve is non-supersingular and the admissible change of variables

$$(x, y) = \left( x' + \frac{a_4 - a_1 a_3}{a_1^2 + a_2}, \quad y' + a_1 x' + a_1 \left( \frac{a_4 - a_1 a_3}{a_1^2 + a_2} \right) + a_3 \right)$$

leads to

$$E' : (y')^2 = (x')^3 + a(x')^2 + b, \tag{2.8}$$

where $a, b \in K$ with discriminant $\Delta = -a^3 b \neq 0$.

The short forms (2.4) to (2.8) are referred to as *short* or *simplified Weierstraß equations.* Any curve over $K$ is isomorphic to a curve defined by a short Weierstraß equation. We will assume that any curve over a prime field of characteristic $\neq 2, 3$ is defined by a short Weierstraß equation of the form of (2.4) and that any non-supersingular curve over a binary field is given in the form of (2.6).

### 2.1.3 Alternative Models

Chudnovsky and Chudnovsky [18] studied four basic forms of elliptic curves including the Weierstraß model, the Jacobi model, the Jacobi form and the Hessian model. Another alternative model is the Montgomery form [74], which is resistant to side-channel attacks. These other models can be useful for other purposes. For example, Liardet and Smart [62] use the Jacobi form to prevent power analysis attacks, while Joye and Quisquater [47] use the Hessian form for its interesting side-channel properties and parallelizability. The Montgomery form can be used to derive faster point operations [79]. The drawbacks are that Hessian, Jacobi and Montgomery forms only apply to elliptic curves over fields with certain properties. For the purpose of this thesis, we focus on the Weierstraß model. The Weierstraß model is completely general in the sense that any elliptic curve can be transformed into Weierstraß form, whereas the same can not be said about the Montgomery form, the Jacobi form or the Hessian form.

## 2.2 Coordinate Systems

In the previous section, the set of $L$-rational points on an elliptic curve over a field $K$ was defined to be the set of solutions over $L$ of a Weierstraß equation in two variables. A point in this form is represented by a pair $(x, y) \in L \times L$; this form of representation is called the affine coordinate system. An alternative coordinate system that can also be used is the projective coordinate system. In such a system, rather than representing a point by a pair of elements, a point is represented by an equivalence class of triples.

Projective coordinate systems are useful because they allow points to be represented using a redundant representation and give a realization of the point at infinity. Computations on the curve are done with different formulas for different representations. Specifically, the formulas for the yet-to-be-defined group operation with standard affine coordinates all require either a finite field inversion or division step. According to Hankerson *et al.* [43], these operations can be very costly relative to field multiplication. Projective coordinates allow for group operations to be calculated without any inversions at the cost of additional field multiplications and a larger representation that may no longer be unique. In this section, several variants of projective systems are examined.

### 2.2.1 Projective Coordinates

The only model for a curve we have examined so far in this chapter has been the affine model. In the affine model, curves are defined by equations in $n$ variables with coefficients in $K \subseteq L$ and $L$-rational points on the curve are elements of the space

$L^n = \underbrace{L \times L \times \cdots \times L}_{\text{n times}}$. For elliptic curves, the points lie in the two-dimensional affine space $L \times L$.

An alternative model for a curve is the projective model. Projective space is similar to affine space in that points in $n$-dimensional projective space are $(n+1)$-tuples of elements of $L$. The difference is that the point $(0, 0, \ldots, 0)$ is not included and that points are equivalent up to scalar multiplication. There are $n$ copies of $n$-dimensional affine space in $n$-dimensional projective space, found by setting one coordinate to 1 and allowing the other coordinates to range over the field.

**Definition 2.2.1.** *For a field $K$, define an equivalence relation $\cong$ on $K^3 \backslash \{(0,0,0)\}$ as follows: $(X_1, Y_1, Z_1) \cong (X_2, Y_2, Z_2)$ if $X_1 = \lambda X_2, Y_1 = \lambda Y_2, Z_1 = \lambda Z_2$ for some $\lambda \in K^*$. We call the equivalence class containing a triple $(X, Y, Z)$ a* projective point *and denote it by $(X : Y : Z)$. The set of all such points*

$$\{(X : Y : Z) \mid (X, Y, Z) \neq (0, 0, 0)\}/\langle\cong\rangle$$

*is called the* projective plane*.*

A polynomial $f \in K[x_1, \ldots, x_n]$ for $n \in \mathbb{N}$ is called *homogeneous* when all terms have the same degree. Curves in $n$-dimensional projective space are defined by homogeneous equations in $n+1$ variables. An elliptic curve in projective space is defined by an equation which is the "homogenization" of the equation for the curve in affine space. We can identify points in projective space with those in affine space with the following maps:

$$\mu : K \times K \to \{(X : Y : Z) \mid Z \neq 0\}/\langle\cong\rangle$$
$$(x, y) \mapsto (x : y : 1).$$

The partial inverse to this map is the following:

$$\begin{aligned} \nu : \{(X : Y : Z) \mid Z \neq 0\}/\langle\cong\rangle &\to K \times K \\ (X : Y : Z) &\mapsto (X/Z, Y/Z). \end{aligned}$$

In order to get an equation of an elliptic curve in projective space, the substitutions $x \leftarrow X/Z$ and $y \leftarrow Y/Z$ are made in the equation for the curve in affine space. Both sides of the resulting equation are multiplied by $Z^3$, resulting in an equation of homogeneous polynomials of degree three. This homogeneous equation is the "homogenization" of the equation for the curve, or the *projective Weierstraß equation.*

For a non-supersingular curve over a field of characteristic other than 2 or 3, the simplified Weierstraß equation is $E : y^2 = x^3 + ax + b$, and the projective Weierstraß equation is

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

For a non-supersingular curve over a field of characteristic 2, the simplified Weierstraß equation is $E : y^2 + xy = x^3 + ax^2 + b$, giving

$$E : Y^2Z + XYZ = X^3 + aX^2Z + bZ^3,$$

for the projective Weierstraß equation. These new equations are non-singular because if there is a point $[X : Y : Z]$ on the curve that will satisfy all three partial derivatives, then $(X/Z, Y/Z)$ will satisfy the partial derivatives of the affine curve, which is a contradiction.

Notice that every solution $(X : Y : Z) = (X/Z : Y/Z : 1)$ in projective space (with $Z \neq 0$) to the projective equation for the curve corresponds to the affine solution $(X/Z, Y/Z)$ of the affine equation of the curve and vice versa. Therefore $\mu \circ \nu(X : Y : Z) = (X : Y : Z)$ for all $(X : Y : Z) \in \{(X : Y : Z) \mid Z \neq 0\}/\langle\cong\rangle$. By composing the the functions $\nu$ and $\mu$, $\nu \circ \mu(x, y) = (x, y)$ for all $(x, y) \in K \times K$. This shows that there is a one-to-one correspondence between affine points satisfying the affine Weierstraß equation and projective points with $Z \neq 0$ satisfying the projective Weierstraß equation.

Solving for $Z = 0$ in the projective Weierstraß equation results in only one point, $(0 : 1 : 0)$. To complete the bijection between affine points on an elliptic curve and projective points on an elliptic curve, the point $(0 : 1 : 0)$ is associated with the point $\infty$ in the affine group.

Projective space has a property that is not found in affine space dealing with the intersection of curves. Two curves in projective space intersect at a point when the equations for both curves are simultaneously satisfied by the coordinates of the point. The multiplicity of such an intersection is a measure of how similar the curves behave at the point. For the intersection point $P$ of an elliptic curve and a line, the multiplicity is one when the line is not tangent to the curve at $P$, two when the line is tangent to the curve at $P$ and is not an inflection point, and three when the curve is tangent to $P$ and is an inflection point. Bèzout's Theorem [89, p. 242] states that two curves of degree $n$ and $m$ in projective space intersect at $k$ points with multiplicities $m_k$ so that $\sum_{i=1}^{k} m_k = n \times m$. Therefore any line intersects an elliptic curve in three places, which is the essential fact for defining the group operation. In the affine case, this does not hold. A vertical line will intersect an elliptic curve with multiplicity at most 2 while every non-vertical line will intersect an elliptic curve with multiplicity 3.

Note that in projective space, any of the three points of intersection could have 0 as a $Z$ coordinate and therefore lie outside of standard affine space. Luckily, an elliptic curve will only intersect this portion of projective space (called the *line at infinity*) at one point. In the case where the characteristic of $K$ is greater than 3, this point is $(0 : 1 : 0)$. We call this point the point at infinity, or $\infty$. Including $\infty$ as an $L$-rational point on an elliptic curve allows the three-points-to-a-line rule to hold in the affine model as well. Any line that would have intersected $(0 : 1 : 0)$ in the projective case is said to intersect the point $\infty$ in the affine case, preserving the fact that any line will intersect an elliptic curve in three places for the affine model. This is the justification for the inclusion of the point $\infty$ in the set of $L$-rational points of a curve in affine space.

### 2.2.2 Generalized Projective Coordinates

The standard projective model of an elliptic curve is useful for improving the efficiency of computation on elliptic curves, but was surpassed by another projective model introduced by Chudnovsky and Chudnovsky [18]. These projective coordinate systems lead to more efficient arithmetic and rely on a generalized definition of projective space.

**Definition 2.2.2.** *For a field $K$, define an equivalence relation $\cong_{c,d}$ on $K^3 \backslash \{(0,0,0)\}$ as follows:*

$(X_1, Y_1, Z_1) \cong (X_2, Y_2, Z_2)$ *if* $X_1 = \lambda^c X_2, Y_1 = \lambda^d Y_2, Z_1 = \lambda Z_2$ *for some* $\lambda \in K^*$, $c, d \in \mathbb{N}$.

*We call the equivalence class containing a triple* $(X, Y, Z)$ *a* $(c, d)$-projective point *and denote it by* $(X : Y : Z)$. *The set of all such points*

$$\{(X : Y : Z) \mid (X, Y, Z) \neq (0, 0, 0)\}/\langle \cong_{(c,d)} \rangle$$

*is called* $(c, d)$-projective 2-space.

If $Z \neq 0$, then a unique representative for $(X : Y : Z)$ is $(X/Z^c, Y/Z^d, 1)$. This gives a one-to-one correspondence between affine points and projective points with $Z \neq 0$.

A similar process to homogenization is used to find the formula for a curve in $(c, d)$-projective space. When $c = d = 1$, the projective version of the equation is homogeneous under the standard notion of degree. With different values of $c$ and $d$, the resulting equation is homogeneous with the following measure of degree:

$$\deg(X) = c, \quad \deg(Y) = d, \quad \text{and} \quad \deg(Z) = 1.$$

To obtain the $(c, d)$-projective version of an affine curve $E : y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6$, substitute $x \to X/Z^c$ and $y \to Y/Z^d$, then multiply through by the highest power of $Z$ in the denominator. If $3c > 2d$, the resulting $(c, d)$-projective curve is

$$E : Y^2Z^{3c-2d} + a_1XYZ^{2c-d} + a_3YZ^{3c-d} = X^3 + a_2X^2Z^c + a_4XZ^{2c} + a_6Z^c.$$

If $3c < 2d$, the resulting $(c, d)$-projective curve is

$$E : Y^2 + a_1XYZ^{d-c} + a_3YZ^d = X^3Z^{2d-3c} + a_2X^2Z^{2d-2c} + a_4XZ^{2d-c} + a_6Z^{2d}.$$

If $3c = 2d$, the resulting $(c, d)$-projective curve is

$$E : Y^2 + a_1XYZ^{d-c} + a_3YZ^d = X^3 + a_2X^2Z^c + a_4XZ^{2c} + a_6Z^{3c}.$$

Solving for $Z = 0$ shows that there is only one $(c, d)$-projective point with $Z = 0$. This corresponds to the point at infinity in the affine coordinate system. To convert back to affine representation, substitute 1 for $Z$. The affine coordinate system is denoted by $\mathcal{A}$.

The main types of projective coordinates used for curves over prime fields are:

- Standard projective coordinates: $c = 1, d = 1$. This coordinate system will be denoted by $\mathcal{P}$. In this system, $\infty$ is represented by $(0 : 1 : 0)$.

- Jacobian projective coordinates: $c = 2, d = 3$. Jacobian and Chudnovsky Jacobian projective coordinates were proposed by Chudnovsky and Chudnovsky [18]. In Jacobian coordinates, point doubling can be performed in fewer finite field operations than in projective coordinates. Jacobian-based coordinate systems are mainly used for elliptic curves over prime fields. This coordinate system will be denoted by $\mathcal{J}$. In this system, $\infty$ is represented by $(1 : 1 : 0)$.

- Chudnovsky Jacobian Coordinates: $((X : Y : Z), Z^2, Z^3)$. The first three coordinates are the same as in Jacobian projective coordinates and two redundant elements are added. These redundant values are used to speed up computations in certain operations (such as point addition). The representation also takes more space and requires extra computation for other operations (such as point doubling). This coordinate system will be denoted by $\mathcal{J}^c$. In this system, $\infty$ is represented by $((1 : 1 : 0), 0, 0)$.

- Modified Jacobian Coordinates: $((X : Y : Z), aZ^4)$. The modified Jacobian coordinate system was introduced by Cohen *et al.* [20] for prime curves. The first three coordinates are the same as in the Jacobian projective case, and one redundant element is added. In this context $a$ is the coefficient of $x$ in the simplified Weierstraß equation (2.4) for the curve. The redundant value is used to speed up computation for point doubling, but also requires extra storage space and more field squarings. This coordinate system has the advantage of having the most efficient point doubling operation, but its advantages versus Jacobian coordinates are eliminated when the coefficient $a$ of the Weierstraß equation (2.4) of the elliptic curve is chosen to be $-3$. This coordinate system will be denoted by $\mathcal{J}^m$. In this system, $\infty$ is represented by $((1:1:0), 0)$.

- Lopez-Dahab (LD) projective coordinates: $c = 1, d = 2$. This coordinate system is useful for elliptic curves over binary fields. In this system, $\infty$ is represented by $(1:0:0)$.

## 2.3 The Group of Points

The set of $L$-rational points on an elliptic curve $E/K$ is a widely studied object in mathematics. Part of the reason that this set is so interesting and relevant to cryptography is that it has an associated group structure. Specifically, there is a natural group operation on the set of points that can be used to create an Abelian group with $\infty$ acting as the identity.

The Abelian group in question arises naturally from the fact that any line intersects an elliptic curve in at most three points. The formula for addition in this group will be derived from the intersection of the equation of a line and the general Weierstraß equation.

Groups of points on elliptic curves over finite fields are the product of two cyclic groups (Section 2.3.5). The parameters of the curve can be chosen so that the group of points has a large prime divisor and therefore a large cyclic subgroup. The fact that the group has a large cyclic subgroup is important for secure cryptography.

### 2.3.1 The Group Law

The group law on the set of points of an elliptic curve over any field $K$ relies on the fact that any line intersects an elliptic curve in at most three points. As is the convention for Abelian groups, the operations will be written additively.

We know that a curve of degree 3 (an elliptic curve in this case) and a curve of degree 1 (a line) intersect in $k \leq 3$ points $P_1, \ldots, P_k$ with $\sum_{i=1}^{k} m_i = 3$, where $m_i$ is the multiplicity of the intersection at $P_i$. We define the following relation on the set of co-linear points:

$$\sum_{i=1}^{k} m_i P_i = \infty. \tag{2.9}$$

If $\infty$ is defined to be the identity element, the relation can be informally described as "co-linear points sum to zero". Since the number of points in the intersection is at most three, this relation can define a group operation. Given two $L$-rational points in affine form $P = (x_1, y_1)$ and $Q = (x_2, y_2)$ on an elliptic curve $E/K$, the group operation is described by the following steps:

- To find $-R$ for any point $R \in E(L)$, $R \neq \infty$, do the following:

  1. Take the line through $R$ and $\infty$. In affine coordinates, a line through $\infty$ is a line of the form $x = a$ for some constant $a \in L$.

2. If the intersection of this line and $E(L)$ contains only $R$ and $\infty$, then $R + R + \infty = \infty$, i.e. $R = -R$.
3. If the intersection of this line contains another point $S$, then $R + S + \infty = \infty$ and hence $-R = S$.

- To compute $P + Q$, where $P$ and $Q \in E(L)$ with $P \neq Q$, $x_1 \neq x_2$, do the following:

  1. Take the unique line through $P$ and $Q$ and find its intersection with $E(L)$
  2. If this intersection contains only $P$ and $Q$, then the line is tangent to the curve at either $P$ or $Q$. This means that either $2P + Q = \infty$ (when the line tangent to $P$ contains $Q$) or $P + 2Q = \infty$ (when the line tangent to $Q$ contains $P$. In the first case, the sum $P + Q$ is $-Q$, and it is $-P$ otherwise.
  3. If the intersection contains another point $R$, then $P + Q + R = \infty$, and hence $P + Q = -R$.

  If $x_1 = x_2$, then the line through $P$ and $Q$ is parallel to the $y$-axis and so $\infty$ is in the intersection, therefore $P + Q = \infty$ and $P = -Q$.

- To compute $P + P$, or $2P$, do the following:

  1. Take the unique line through $P$ tangent to $E$ (this line is unique because the curve is non-singular) and the intersection of this line with $E(L)$.
  2. If this intersection contains only $P$, then $P$ is an inflection point, so $3P = \infty$, and hence $2P = -P$.

3. If there is another point $R$ in the intersection, then $2P + R = \infty$ and hence $P + P = -R$.

When $K = \mathbb{R}$, this process can be shown graphically. Figure 2.1(a) describes the addition of the points $P$ and $Q$ on an elliptic curve. First, a line is drawn through $P$ and $Q$, this line intersects the curve at three points, $P$, $Q$ and $-(P + Q)$. In order to find $P + Q$, the negative of $-(P + Q)$ is taken. A vertical line through $-(P + Q)$ intersects the curve at two points, $-(P + Q)$ and our desired point $P + Q$.

Figure 2.1(b) describes the doubling of the point $P$ on a curve over $\mathbb{R}$. First, a line is drawn through $P$ and tangent to the curve. This line intersects the curve at two points, $P$, and $-(2P)$. In order to find $2P$, the negative of $-(2P)$ is taken. A vertical line through $-(2P)$ intersects the curve at two points, $-(2P)$ and our desired point $2P$.

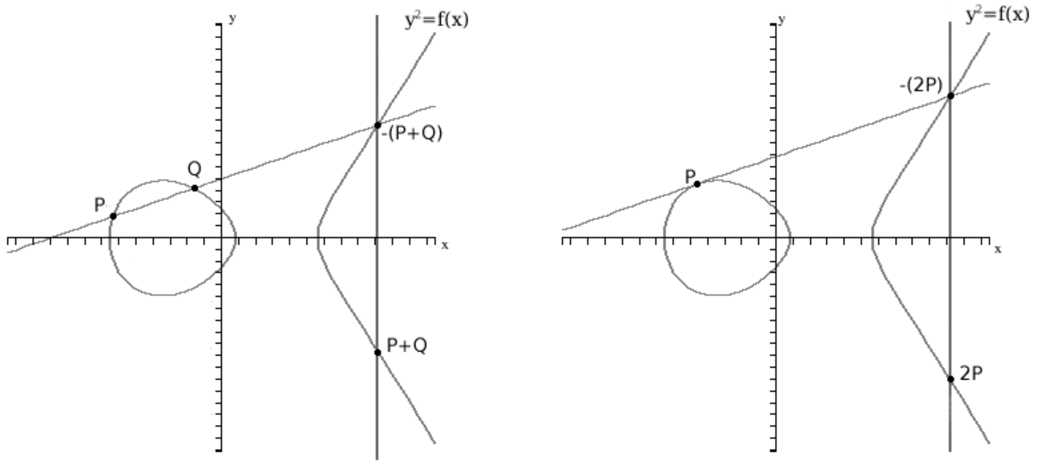
We now derive the addition formulas for an arbitrary field $K$. Let $E/K$ be an elliptic curve given by the Weierstraß equation $E : y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6$ with $a_1, a_2, a_3, a_4, a_6 \in K$ and $K \subseteq L \subseteq \overline{K}$.

Let $P$, $Q \in E(L)$ with $P \neq Q$ be points in affine coordinates, $P = (x_1, y_1)$, $Q = (x_2, y_2)$, where $x_1 \neq x_2$. We compute the coordinates of $P + Q = R = (x_3, y_3)$. The line through $P$ and $Q$ has slope

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

and passes through $P$. Denoting the constant term by $\mu$, the equation becomes $y = \lambda x + \mu$. Since $P$ satisfies this equation, we get $\mu = y_1 - \lambda x_1$. The equation for the line is then

$$y = \lambda x + \frac{x_1y_2 - x_2y_1}{x_1 - x_2}.$$



(a) Point Addition over $\mathbb{R}$ (b) Point Doubling over $\mathbb{R}$

Figure 2.1: Point Arithmetic over $\mathbb{R}$.

The intersection of the line with the curve is obtained by equating the equation for the line and the Weierstraß equation for $E/K$:

$$(\lambda x + \mu)^2 + (a_1 x + a_3)(\lambda x + \mu) = x^3 + a_2 x^2 + a_4 x + a_6.$$

Solving this equation is equivalent to finding the roots of the polynomial $r(x)$, defined by

$$r(x) = x^3 + (a_2 - \lambda^2 - a_1\lambda)x^2 + (a_4 - 2\lambda\mu - a_3\lambda - a_1\mu)x + a_6 - \mu^2 - a_3\mu.$$

Two of the roots of this polynomial are already known, namely the $x$-coordinates of $P$ and $Q$. We know that the negative of the sum of the roots of a monic polynomial is equal to the coefficient of the second highest term. If $x_3$ is the third root, we have $\lambda^2 + a_1\lambda - a_2 = x_1 + x_2 + x_3$. This implies the $x$-coordinate of the third co-linear point, $-R$, is $x_3 = \lambda^2 + a_1\lambda - a_2 - x_1 - x_2$. Furthermore, the point $-R$ is on the same line and therefore has $y$-coordinate $\widetilde{y_3} = \lambda x_3 + \mu$.

By the choice of the neutral element, any point and its inverse share the same $x$-coordinate. If a point $-R = (x_3, y_3')$ lies on the curve, so does $R = (x_3, y_3)$. By simultaneously solving the Weierstraß equation evaluated at $(x, y)$ and $(x, y')$, we find that $y' = -y - a_1x - a_3$. For $R = (x_3, y_3)$, we get that $y_3 = -\lambda x_3 - \mu - a_1x_3 - a_3$.

There are two cases for which $x_1 = x_2$, namely $y_1 = y_2$ and $y_1 \neq y_2$. If $y_1 = y_2$, the operation for doubling $P$ is analogous to point addition except that $\lambda$ is the slope of $E/K$ at $P$ obtained through implicit differentiation. If $y_1 \neq y_2$ then the line through the two points is vertical and the two points sum to $\infty$. In summary,

$$\begin{aligned} -P &= (x_1, -y_1 - a_1x_1 - a_3), \qquad (2.10)\\ P + Q &= \begin{cases} \infty & \text{if } Q = -P,\\ (\lambda^2 + a_1\lambda - a_2 - x_1 - x_2, \lambda(x_1 - x_3) - y_1 - a_1x_3 - a_3) & \text{otherwise,} \end{cases} \end{aligned}$$

where

$$\lambda = \begin{cases} \dfrac{y_1 - y_2}{x_1 - x_2} & \text{if } P \neq \pm Q,\\ \dfrac{3x_1^2 + 2a_2x_1 + a_4 - a_1y_1}{2y_1 + a_1x_1 + a_3} & \text{if } P = Q. \end{cases}$$

Proposition 2.3.1 from Washington [94, page 3] shows that the binary operation $+$ is in fact a group operation.

**Proposition 2.3.1.** *Let $K$ be a field, $L$ an extension of $K$ and $E/K$ an elliptic curve over $K$. Let $+$ denote the group operation on points. Then $(E(L), +)$ is an Abelian group.*

This proposition shows that the set of $L$-rational points on an elliptic curve is an Abelian group under point addition. Previously, we defined an isomorphism on curves in order to simplify the Weierstraß equation for curves over certain fields. Theorem 2.3.2, taken from Silverman [88, Sec. 2.2], justifies this definition.

**Theorem 2.3.2.** *Let $E/K$ and $E'/K$ be elliptic curves. If $E/K$ and $E'/K$ are isomorphic as elliptic curves, then $E(K)$ and $E'(K)$ are isomorphic as groups.*

This result provides sufficient justification for our initial definition of isomorphism between curves. Algorithms for computing the group operations efficiently in the various coordinate systems will be explored later in this chapter.

### 2.3.2 Scalar Multiplication

Scalar multiplication is an operation that can be performed on any additively written group. We will denote scalar multiplication by $n \in \mathbb{N}$ on the set $E(L)$ of $L$-rational points on an elliptic curve $E/K$ for $K$ a field and $K \subseteq L \subseteq \overline{K}$ with the following operator:

$$\begin{array}{rccc} n: & E(L) & \rightarrow & E(L) \\ & P & \mapsto & \underbrace{P + P + \cdots + P}_{n \text{ times}}. \end{array}$$

The scalar product of a point $P \in E(L)$ by $n \in \mathbb{N}$ is therefore denoted $nP$. This operator will be very important in Chapter 3.

### 2.3.3 Custom Group Law Formulas

In order to perform arithmetic on an elliptic curve group, the fundamental operations needed are point addition and point doubling. The general formulas presented in the previous section can be used to perform these operations. These formulas are based on the general Weierstraß equation (2.1) and can be improved by considering a simplified Weierstraß such as equations (2.4) or (2.6) instead.

Suppose that $E/\mathbb{F}_q$ is a curve n prime field of odd characteristic given by a simplified Weierstraß equation (2.4). It follows from (2.11) and (2.10) that the following rules hold:

1. Identity: $\infty + P = P + \infty = P$ for all $P \in E(\mathbb{F}_q)$.

2. Negation: If $P = (x, y) \in E(\mathbb{F}_q)$, then $-P = (x, -y)$. If $P = \infty$, then $-P = P$.

3. Addition: If $P = (x_1, y_1) \in E(\mathbb{F}_q)$, $Q = (x_2, y_2) \in E(\mathbb{F}_q)$ and $P \neq \pm Q$, then $P + Q = (x_3, y_3)$ with

$$x_3 = \lambda^2 - x_1 - x_2 \text{ and } y_3 = \lambda(x_1 - x_3) - y_1, \tag{2.11}$$

where $\lambda = (y_2 - y_1)/(x_2 - x_1)$.

4. Doubling: If $P = (x_1, y_1)$ and $P \neq -P$ then $2P = (x_3, y_3)$ with

$$x_3 = \lambda^2 - 2x_1 \text{ and } y_3 = \lambda(x_1 - x_3) - y_1, \tag{2.12}$$

where $\lambda = 3x_1^2 + a/2y_1$.

In fact, these rules hold for any field of characteristic not equal to 2 or 3.

If $E/\mathbb{F}_q$ is a non-supersingular curve over a binary field given by a simplified Weierstraß equation (2.6) then it follows from Equations (2.10) and (2.11) that the following rules hold on points in affine form:

1. Identity: $\infty + P = P + \infty = P$ for all $P \in E(\mathbb{F}_q)$.

2. Negation: If $P = (x, y) \in E(\mathbb{F}_q)$, then $-P = (x, x+y)$; if $P = \infty$, then $-P = P$.

3. Addition: If $P = (x_1, y_1) \in E(\mathbb{F}_q)$, $Q = (x_2, y_2) \in E(\mathbb{F}_q)$ and $P \neq \pm Q$, then $P + Q = (x_3, y_3)$ with

$$x_3 = \lambda^2 + \lambda + a = x_1^2 + x_2 + a \text{ and } y_3 = \lambda(x_1 + x_3) + x_3 + y_1, \tag{2.13}$$

where $\lambda = (y_1 + y_2)/(x_1 + x_2)$.

4. Doubling: If $P = (x_1, y_1)$ and $P \neq -P$ then $2P = (x_3, y_3)$ with

$$x_3 = \lambda^2 + \lambda + a \text{ and } y_3 = \lambda(x_1 + x_3) + x_3 + y_1, \tag{2.14}$$

where $\lambda = x_1 + y_1/x_1$.

These formulas allow for the computation of addition and doubling of elliptic curve points when they are given in affine coordinates. By making the substitutions $x \leftarrow X/Z^c, y \leftarrow Y/Z^d$, these formulas can be used to compute points in $(c, d)$-projective coordinates. In the next section, we will describe the algorithms that are used to compute group operations using the previous formulas in various coordinate systems with the minimum number of field operations.

### 2.3.4 Group of Points Over $\mathbb{F}_q$

A special case to consider is that of elliptic curves over a finite field $\mathbb{F}_q$. In order to use the group $E(\mathbb{F}_q)$ for cryptographic purposes, it is necessary to have an understanding of the size of the group and its algebraic structure.

There are a few basic requirements for a group to be useful for discrete logarithm based cryptography. At the very least, the group must be of sufficiently large size and it must have a large cyclic subgroup. In this section, it is shown that for elliptic curves over finite fields, the size of an elliptic curve group is approximately the same as that of the field over which it is defined, that the group is a direct product of two cyclic subgroups, and other useful properties.

### 2.3.5 Basic Group Properties

The size of the group $E(\mathbb{F}_q)$ is denoted by $\#E(\mathbb{F}_q)$. Since $E(\mathbb{F}_q)\backslash\{\infty\} \subseteq \mathbb{F}_q \times \mathbb{F}_q$, this set is finite. This is strengthened with Theorem 2.3.3, the Hasse-Weil bound. The proof of the Hasse-Weil bound and more details concerning the possible values of $\#E(\mathbb{F}_q)$ can be found in Waterhouse [95, pages 536–538].

**Theorem 2.3.3 (Hasse-Weil bound).** *Let $E/\mathbb{F}_q$ be an elliptic curve. Then*

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

Since $\sqrt{q}$ is small relative to $q$, we have $\#E(\mathbb{F}_q) \approx q$. Theorem 2.3.4, from Koblitz [54, Sec 6.1], gives the exact order of the group $E(\mathbb{F}_{q^n})$ as long as $\#E(\mathbb{F}_q)$ is known. This is useful for the case when $q$ is a small prime such as 2 or 3.

**Theorem 2.3.4.** *Let $E/\mathbb{F}_q$ be an elliptic curve and define $t$ via $t = q + 1 - \#E(\mathbb{F}_q)$. Then*

$$\#E(\mathbb{F}_{q^n}) = q^n + 1 - V_n,$$

*where $\{V_n\}$ is the sequence defined by $V_0 = 2, V_1 = t, V_n = V_1 V_{n-1} - qV_{n-2}$ for $n \geq 2$.*

Theorem 2.3.5, from Washington [94, Sec. 4.3], describes the group structure of an elliptic curve.

**Theorem 2.3.5.** *Let $E$ be an elliptic curve defined over $\mathbb{F}_q$. Then $E(\mathbb{F}_q)$ is isomorphic to $\mathbb{Z}/n_1\mathbb{Z} \oplus \mathbb{Z}/n_2\mathbb{Z}$ where $n_1$ and $n_2$ are uniquely determined positive integers such that $n_2$ divides both $n_1$ and $q-1$.*

The most useful scenario for cryptographic applications is the case when one of the two subgroups is a very small group or trivial. Galbraith and McKee [36] have made conjectures about the probability of a random curve having such a group structure, including the following:

**Conjecture 2.3.6.** *Let $P_1$ be the probability that a number within $2\sqrt{p}$ of $p+1$ is prime. Then the probability that an elliptic curve over $\mathbb{F}_{p^k}$ has a prime number of points is asymptotic to $c_pP_1$ as $p \to \infty$, where*

$$c_p = \frac{2}{3}\prod_{l>2}\left(1-\frac{1}{(l-1)^2}\right)\prod_{\substack{l|p-1\\ l>2}}\left(1+\frac{1}{(l+1)(l-2)}\right).$$

*Here the products are over all primes $l$ satisfying the stated conditions.*

Note that $\prod_{l>2}\left(1-\frac{1}{(l-1)^2}\right) \approx 0.6601618$ is the Hardy-Littlewood twin primes constant. Galbraith and McKee also give a similar conjecture about the probability that a random elliptic curve over $\mathbb{F}_p$ has $k \cdot q$ points for a prime $q$. These conjectures are backed up by numerical evidence and suggest that these curves happen with a large enough probability that a curve with $k \cdot p$ points for small $k$ can be found in a reasonable amount of time by randomly choosing curves.

## 2.4 Optimizing Formulas for Prime Curve Arithmetic

In this section, we examine the explicit algorithms for computing in the group of points on an elliptic curve defined over a finite field. The goal of this section is to describe formulas that compute point doubling and point addition in the minimal number of field operations. We also present the best known algorithms for performing these operations in affine, projective and mixed coordinate systems.

The notation $C + D \rightarrow E$ will denote the operation that adds two points in coordinate systems $C$ and $D$, and outputs a point in coordinate system $E$. Similarly, $2C \rightarrow E$ will denote the operation that doubles a point in coordinate system $C$ and outputs a point in coordinate system $E$. The *field cost* of an algorithm is defined to be the number of multiplications ($M$), squarings ($S$) and inversions ($I$) in $\mathbb{F}_q$ required for the algorithm. The operations that are omitted (addition, subtraction) take an insignificant amount of time relative to the other operations as indicated in Appendix A.

The problem of minimizing the storage space needed for the implementation of an algorithm is important because elliptic curve cryptography is often used on constrained devices. However, this is not examined in detail in this exposition. If there are two algorithms with the same field cost but differing storage space requirements, we choose the most space-efficient algorithm. If the number of temporary variables is of the utmost importance, there are a number of options including modifying the algorithms so that the input variables are overwritten and used as temporary variables. Input variables are lost but temporary space is saved.

We present the algorithms for the curve $E/\mathbb{F}_p$ with equation $E : y^2 = x^3 + ax + b$ for a prime $p$ greater than 3. A restriction that we will make is that we will always assume $a = -3$ for the curve. This restriction does not reduce the security of the curve or put a restriction on the possible group structures that can be obtained (Theorem 3.15 of Hankerson *et al.* [43]). This restriction is common practice and allows the value $3X^2 + aZ^4$, which requires 3 squarings to be computed naïvely, to be calculated as $3(X - Z^2)(X + Z^2)$ with only one squaring and one multiplication. This allows a saving of 2 squarings in the doubling formula for points in Jacobian form.

Although arithmetic on binary curves is also cryptographically important, the focus of this thesis is elliptic curves over prime fields. Algorithms for binary arithmetic are presented by Hankerson *et al.* [43, Sec. 3.2.3] and by Cohen *et al.* [6, Sec 13.3].

### 2.4.1 Affine Operations

The algorithms for calculating point addition and point doubling in affine coordinates follow directly from the formulas in Section 2.3.3 and are similar to presentations in Hankerson *et al.* [43, Sec 3.1], and Hitchcock *et al.* [44].

```
**Algorithm 1:** Point Doubling ($2\mathcal{A} \to \mathcal{A}$)
**Input:** Affine Point $P = (x_1, y_1)$,
**Output:** Affine Point $2P = (x_2, y_2)$
(1)   **if** $P = \infty$ **then return** $\infty$
(2)   $r_1 \leftarrow x_1^2$
(3)   $r_1 \leftarrow r_1 - 1$
(4)   $r_2 \leftarrow 2r_1$
(5)   $r_1 \leftarrow r_2 + r_1$
(6)   **if** $r_1 = 0$ **then return** $\infty$
(7)   $r_2 \leftarrow 2y_1$
(8)   $r_2 \leftarrow r_2^{-1}$
(9)   $r_1 \leftarrow r_1 \cdot r_2$
(10)  $r_3 \leftarrow (-2)x_1$
(11)  $r_2 \leftarrow r_1^2$
(12)  $x_2 \leftarrow r_1 + r_2$
(13)  $r_3 \leftarrow x_1 - x_2$
(14)  $r_2 \leftarrow r_1 \cdot r_3$
(15)  $y_2 \leftarrow r_2 - y_1$
```

```
(16)  $Q := (x_2, y_2)$
(17)  **return** $Q$
```

The field cost for Algorithm 1 is $I + 2M + 2S$, and 3 temporary variables are required.

```
**Algorithm 2:** Point Addition ($\mathcal{A} + \mathcal{A} \rightarrow \mathcal{A}$)
**Input:** Affine Points $P_1 = (x_1, y_1) \neq P_2 = (x_2, y_2)$
**Output:** Affine Point $P_1 + P_2 = (x_3, y_3)$
(1)   **if** $x_1 = x_2$ **then return** $\infty$
(2)   $r_1 \leftarrow x_2 - x_1$
(3)   $r_2 \leftarrow y_2 - y_1$
(4)   $r_2 \leftarrow r_2^{-1}$
(5)   $r_1 \leftarrow r_1 \cdot r_2$
(6)   $r_3 \leftarrow r_1^2$
(7)   $r_3 \leftarrow r_3 - x_1$
(8)   $x_3 \leftarrow r_3 - x_2$
(9)   $r_2 \leftarrow x_1 - x_3$
(10)  $r_3 \leftarrow r_1 \cdot r_2$
(11)  $y_3 \leftarrow r_3 - y_1$
(12)  $Q := (x_3, y_3)$
(13)  **return** $Q$
```

The field cost for Algorithm 2 is $I{+}2M{+}S$ and 3 temporary variables are required. These formulas are rarely used in practice because they involve the expensive finite field inversion operation. In Appendix A, we explain that one inversion takes the same time as approximately 30 to 80 multiplications in $\mathbb{F}_p$ for large primes depending on the implementation. It will be demonstrated in the next subsection that the addition and doubling operations can be performed in significantly less time using projective coordinates.

### 2.4.2 Projective Operations

The formulas for point arithmetic in $(c, d)$-projective coordinates can be obtained from the formulas for affine coordinates with the substitutions $x_i = X_i/Z_i^c$ and $y_i = Y_i/Z_i^d$ for input points $(x_i, y_i)$. This substitution leaves several terms with negative exponents in the formulas. Since the projective point is really an equivalence class of points, then for any value $\alpha$, the point $(X\alpha^c, Y\alpha^d, Z\alpha)$ can also be returned. The formulas can therefore be modified to remove denominators by multiplying through by constants using a specifically chosen $\alpha$. The resulting formulas can be used to compute addition or doubling for points given in $(c, d)$-projective coordinates.

The Chudnovsky Jacobian ($(2, 3)$-projective) form of the Weierstraß equation $E : y^2 = x^3 + ax + b$ is

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

the point at infinity is $(1 : 1 : 0)$ and the negative of $(X : Y : Z)$ is $(X : -Y : Z)$. We will continue to assume that $a = -3$.

By replacing $x_1$ by $X_1/Z_1^2$ and $y_1$ by $Y_1/Z_1^3$ in Equation (2.11) and cancelling denominators, the formula for adding $(X_1 : Y_1 : Z_1)$ to $(X_2 : Y_2 : Z_2)$ becomes

$$U_1 = X_1Z_2^2, \quad U_2 = X_2Z_1^2,$$

$$S_1 = Y_1Z_2^3, \quad S_2 = Y_2Z_1^3,$$

$$X = U_2 - U_1, \quad Y = S_2 - S_1,$$

$$X_3 = -X^3 - 2U_1X^2 + Y^2,$$

$$Y_3 = -S_1X^3 + Y(U_1X^2 - X_3),$$

$$Z_3 = Z_1Z_2X.$$

One complication is that it is not immediately clear if two points given in projective form are equal. It is possible that two points are used as the input of an addition formula and that both points are the same. The algorithms we present determine whether $P = \pm Q$ and call either the doubling algorithm or return $\infty$.

Similarly, by replacing $x_1$ by $X_1/Z_1^2$ and $y_1$ by $Y_1/Z_1^3$ in Equation (2.12) and cancelling denominators, the formula for doubling $(X_1 : Y_1 : Z_1)$ becomes

$$M = 3X_1^2 + aZ_1^4,$$

$$S = 4X_1Y_1^2, \quad T = -2S + M^2,$$

$$X_3 = T,$$

$$Y_3 = -8Y_1^4 + M(S - T),$$

$$Z_3 = 2Y_1Z_1.$$

Algorithms 3 to 6 calculate point addition and doubling in Jacobian and Chudnovsky Jacobian coordinates on a curve $E/\mathbb{F}_p$. The algorithms are adapted from presentations in Hankerson *et al.* [43, Sec 3.2], and Hitchcock *et al.* [44].

```
**Algorithm 3:** Point Doubling ($2\mathcal{J} \to \mathcal{J}$)
**Input:** Jacobian Point $P = (X_1 : Y_1 : Z_1)$
**Output:** Jacobian Point $2P = Q = (X_3 : Y_3 : Z_3)$
(1)   **if** $Z_1 = 0$ **then return** $P_1$
(2)   $r_1 \leftarrow Z_1^2$
(3)   $Z_3 \leftarrow X_1 - r_1$
(4)   $r_1 \leftarrow X_1 + r_1$
(5)   $Z_3 \leftarrow r_1 \cdot Z_3$
(6)   $r_1 \leftarrow 2Z_3$
(7)   $r_1 \leftarrow r_1 + Z_3$
(8)   **if** $r_1 = 0$ **then return** $(1 : 1 : 0)$
(9)   $Y_3 \leftarrow 2Y_1$
(10)  $Y_3 \leftarrow Y_3^2$
(11)  $Z_3 \leftarrow Y_3^2$
```

```
(12)  $Y_3 \leftarrow Y_3 \cdot X_1$
(13)  $Z_3 \leftarrow Z_3/2$
(14)  $X_3 \leftarrow r_1^2$
(15)  $X_3 \leftarrow X_3 - Y_3$
(16)  $X_3 \leftarrow X_3 - Y_3$
(17)  $Y_3 \leftarrow Y_3 - X_3$
(18)  $Y_3 \leftarrow Y_3 \cdot r_1$
(19)  $Y_3 \leftarrow Y_3 - Z_3$
(20)  $Z_3 \leftarrow Y_1 \cdot Z_1$
(21)  $Z_3 \leftarrow 2Z_3$
(22)  $Q := (X_3 : Y_3 : Z_3)$
(23)  **return** $Q$
```

The field cost Algorithm 3 is $4M + 4S$ (or $M + S$ if $2P = \infty$) and only one temporary variable is required.

```
**Algorithm 4:** Point Addition ($\mathcal{J} + \mathcal{J} \rightarrow \mathcal{J}$)
**Input:** Jacobian Points $P_1 = (X_1 : Y_1 : Z_1)$, $P_2 = (X_2 : Y_2 : Z_2)$
**Output:** Jacobian Point $P_1 + P_2 = (X_3 : Y_3 : Z_3)$
(1)   **if** $Z_1 = 0$ **then return** $P_2$
(2)   **if** $Z_2 = 0$ **then return** $P_1$
(3)   $r_1 \leftarrow Z_2^2$
(4)   $X_3 \leftarrow X_1 \cdot r_1$
(5)   $r_1 \leftarrow Z_2 \cdot r_1$
(6)   $Y_3 \leftarrow Y_1 \cdot r_1$
(7)   $r_1 \leftarrow Z_1^2$
(8)   $r_2 \leftarrow X_2 \cdot r_1$
(9)   $r_1 \leftarrow Z_1 \cdot r_1$
(10)  $r_1 \leftarrow Y_2 \cdot r_1$
(11)  $r_1 \leftarrow r_1 - Y_3$
(12)  $r_2 \leftarrow r_2 - X_3$
(13)  **if** $r_2 = 0$
(14)      **if** $r_1 = 0$
(15)          $Q := 2P_1$ using $2\mathcal{J} \rightarrow \mathcal{J}$
(16)          **return** $Q$
(17)      **else**
(18)          $Q := (1 : 1 : 0)$
(19)          **return** $Q$
(20)  $Z_3 \leftarrow Z_1 \cdot Z_2$
```

```
(21)  $Z_3 \leftarrow Z_3 \cdot r_2$
(22)  $r_3 \leftarrow r_2^2$
(23)  $r_2 \leftarrow r_2 \cdot r_3$
(24)  $r_3 \leftarrow r_3 \cdot X_3$
(25)  $X_3 \leftarrow r_1^2$
(26)  $Y_3 \leftarrow r_2 \cdot Y_3$
(27)  $r_2 \leftarrow X_3 - r_2$
(28)  $X_3 \leftarrow 2r_3$
(29)  $X_3 \leftarrow r_2 - X_3$
(30)  $r_2 \leftarrow r_3 - X_3$
(31)  $r_2 \leftarrow r_1 \cdot r_2$
(32)  $Y_3 \leftarrow r_2 - Y_3$
(33)  $Q := (X_3 : Y_3 : Z_3)$
(34)  return $Q$
```

The field cost for Algorithm 4 is $12M + 4S$, unless either $P_1 \pm P_2$ or either point is $\infty$. In the case where $P_1 = P_2$, the field cost is $6M + 2S$ plus the cost of $2\mathcal{J} \to \mathcal{J}$, $10M + 6S$ in total. When $P_1 = -P_2$, the field cost is $6M + 2S$. The operation does not require any additions or multiplications when either point is $\infty$.

The algorithms for $\mathcal{J}^c$ are similar to those for $\mathcal{J}$ but take advantage of the additional terms provided by the Chudnovsky representation. In Chudnovsky Jacobian representation, the terms $Z_1^2, Z_1^3, Z_2^2$, and $Z_2^3$ are already computed, which saves $2M + 2S$, while $Z_3^2, Z_3^3$ have to be computed at the end, costing $M + S$.

```
Algorithm 5: Point Doubling ($2\mathcal{J}^c \to \mathcal{J}^c$)
Input: Chudnovsky Jacobian Point $P_1 = ((X_1 : Y_1 : Z_1), Z_1^2, Z_1^3)$
Output: Chudnovsky Jacobian Point $2P_1 = ((X_3 : Y_3 : Z_3), Z_3^2, Z_3^3)$
(1)   if $Z_1 = 0$ then return $P_1$
(2)   $Z_3 \leftarrow Y_1 \cdot Z_1$
(3)   $Z_3 \leftarrow 2Z_3$
(4)   $(Z_3)^2 \leftarrow X_1 - (Z_1^2)$
(5)   $(Z_3)^3 \leftarrow X_1 + (Z_1^2)$
(6)   $(Z_3)^2 \leftarrow (Z_3)^3 \cdot (Z_3)^2$
(7)   $(Z_3)^3 \leftarrow 2(Z_3)^2$
(8)   $(Z_3)^3 \leftarrow (Z_3)^3 + (Z_3)^2$
```

```
(9)     if $(Z_3)^3 = 0$ then return $(1 : 1 : 0)$
(10)    $Y_3 \leftarrow 2Y_1$
(11)    $Y_3 \leftarrow Y_3^2$
(12)    $(Z_3)^2 \leftarrow Y_3^2$
(13)    $Y_3 \leftarrow Y_3 \cdot X_1$
(14)    $(Z_3)^2 \leftarrow (Z_3)^2/2$
(15)    $X_3 \leftarrow ((Z_3)^3)^2$
(16)    $X_3 \leftarrow X_3 - Y_3$
(17)    $X_3 \leftarrow X_3 - Y_3$
(18)    $Y_3 \leftarrow Y_3 - X_3$
(19)    $Y_3 \leftarrow Y_3 \cdot (Z_3)^3$
(20)    $Y_3 \leftarrow Y_3 - (Z_3)^2$
(21)    $Z_3^2 \leftarrow (Z_3)^2$
(22)    $Z_3^3 \leftarrow (Z_3^2) \cdot (Z_3)$
(23)    $Q := ((X_3 : Y_3 : Z_3), Z_3^2, Z_3^3)$
(24)    return $Q$
```

The field cost for Algorithm 5 is $5M+4S$ and no temporary variables are required.

**Algorithm 6:** Point Addition $(\mathcal{J}^c + \mathcal{J}^c \rightarrow \mathcal{J}^c)$
**Input:** Chudnovsky Jacobian Points $P_1 = ((X_1 : Y_1 : Z_1), Z_1^2, Z_1^3)$, $P_2 = ((X_2 : Y_2 : Z_2), Z_2^2, Z_2^3)$
**Output:** Chudnovsky Jacobian Point $P_1+P_2 = ((X_3 : Y_3 : Z_3), Z_3^2, Z_3^3)$

```
(1)     if $Z_1 = 0$ then return $P_2$
(2)     if $Z_2 = 0$ then return $P_1$
(3)     $X_3 \leftarrow X_1 \cdot (Z_2^2)$
(4)     $Y_3 \leftarrow Y_1 \cdot (Z_2^3)$
(5)     $(Z_3^3) \leftarrow X_2 \cdot (Z_1^2)$
(6)     $(Z_3^2) \leftarrow Y_2 \cdot (Z_1^3)$
(7)     $(Z_3^2) \leftarrow (Z_3^2) - Y_3$
(8)     $(Z_3^3) \leftarrow (Z_3^3) - X_3$
(9)     if $(Z_3^3) = 0$
(10)        if $(Z_3^2) = 0$
(11)            $Q := 2P_1$ using $2\mathcal{J}^c \rightarrow \mathcal{J}^c$
(12)            return $Q$
(13)        else
(14)            $Q := ((1 : 1 : 0), 0, 0)$
(15)            return $Q$
(16)    $Z_3 \leftarrow Z_1 \cdot Z_2$
(17)    $Z_3 \leftarrow Z_3 \cdot (Z_3^3)$
```

```
(18)  $r_1 \leftarrow (Z_3^3)^2$
(19)  $(Z_3^3) \leftarrow (Z_3^3) \cdot r_1$
(20)  $r_1 \leftarrow r_1 \cdot X_3$
(21)  $X_3 \leftarrow (Z_3^2)^2$
(22)  $Y_3 \leftarrow (Z_3^3) \cdot Y_3$
(23)  $(Z_3^3) \leftarrow X_3 - (Z_3^3)$
(24)  $X_3 \leftarrow 2r_1$
(25)  $X_3 \leftarrow (Z_3^3) - X_3$
(26)  $(Z_3^3) \leftarrow r_1 - X_3$
(27)  $(Z_3^3) \leftarrow (Z_3^2) \cdot (Z_3^3)$
(28)  $Z_3^2 \leftarrow (Z_3)^2$
(29)  $Z_3^3 \leftarrow Z \cdot (Z_3^2)$
(30)  $Q := ((X_3 : Y_3 : Z_3), Z_3^2, Z_3^3)$
(31)  **return** $Q$
```

The field cost for Algorithm 6 is $11M + 3S$, unless either $P_1 = \pm P_2$ or either point is $\infty$, and the algorithm only requires one temporary variable. In the case where $P_1 = P_2$, the field cost is $4M$ plus the cost of $2\mathcal{J}^c \to \mathcal{J}^c$, or $9M + 4S$ in total. When $P_1 = -P_2$, the field cost is $4M$. The operation does not require any additions or multiplications when either point is $\infty$.

The addition formula for Chudnovsky Jacobian coordinates takes less computation than Algorithm 4 because $Z_1^2, Z_1^3, Z_2^2$, and $Z_2^3$ are all precomputed.

### 2.4.3 Mixed Coordinate Operations

Mixed coordinate operations were introduced by Cohen *et al.* [20]. The authors noted that in previous research, the arithmetic on the curve was performed using only one coordinate system. They then developed formulas for addition and doubling that would correctly add two points in different coordinate systems or take a point in one coordinate system and return its double in another.

The algorithms that are most useful in later chapters are described in complete detail with the number of finite field operations required by each of them. Algorithms 7 to 9 are mixed coordinate algorithms derived from Hankerson *et al.* [43, Sec. 3.2] and Hitchcock *et al.* [44]. The first mixed algorithm takes an affine point and returns its double in Jacobian form.

```
Algorithm 7: Point Doubling ($2\mathcal{A} \to \mathcal{J}$)
Input: Affine Point $P_1 = (x_1, y_1)$
Output: Jacobian Point $2P_1 = Q = (X_3 : Y_3 : Z_3)$
(1)    if $P = \infty$ or $y_1 = 0$ then return $(1 : 1 : 0)$
(2)    $r_1 \leftarrow x_1^2$
(3)    $Z_3 \leftarrow 2r_1$
(4)    $Z_3 \leftarrow Z_3 + r_1$
(5)    $Y_3 \leftarrow 2y_1$
(6)    $Z_3 \leftarrow Y_3 \cdot x_1$
(7)    $Y_3 \leftarrow Y_3^2$
(8)    $Y_3 \leftarrow Y_3^2$
(9)    $Y_3 \leftarrow Y_3/2$
(10)   $X_3 \leftarrow Z_3^2$
(11)   $r_1 \leftarrow 2Z_3$
(12)   $X_3 \leftarrow X_3 - r_1$
(13)   $r_1 \leftarrow r_3 - X_3$
(14)   $r_1 \leftarrow r_1 \cdot Z_3$
(15)   $Y_3 \leftarrow r_1 - Y_3$
(16)   $Z_3 \leftarrow y_1$
(17)   $Q := (X_3 : Y_3 : Z_3)$
(18)   return $Q$
```

The field cost for Algorithm 7 is $2M + 4S$, a savings of two multiplications compared to the algorithm for $2\mathcal{J} \to \mathcal{J}$. Only one temporary variable is required.

Algorithm 8 takes two affine points and returns their sum in Jacobian form. It is assumed that these two points are distinct.

```
Algorithm 8: Point Addition ($\mathcal{A} + \mathcal{A} \to \mathcal{J}$)
Input: Two Affine Points $P_1 = (x_1, y_1), P_2 = (x_2, y_2)$
Output: Jacobian Point $P_1 + P_2 = Q = (X_3 : Y_3 : Z_3)$
(1)    if $P_1 = \infty$ then return $P_2$
```

```
(2)   if $P_2 = \infty$ then return $P_2$
(3)   if $x_1 = x_2$ then return $\infty$
(4)   $Z_3 \leftarrow x_2 - x_1$
(5)   $r_1 \leftarrow Z_3^2$
(6)   $r_2 \leftarrow Z_3 \cdot r_1$
(7)   $r_1 \leftarrow r_1 \cdot x_1$
(8)   $r_1 \leftarrow 2r_1$
(9)   $Y_3 \leftarrow y_1 - y_2$
(10)  $X_3 \leftarrow Y_3^2$
(11)  $X_3 \leftarrow X_3 - r_1$
(12)  $X_3 \leftarrow X_3 - r_2$
(13)  $r_2 \leftarrow r_2 \cdot Y_1$
(14)  $r_1 \leftarrow r_1 - X_3$
(15)  $Y_3 \leftarrow Y_3 \cdot r_1$
(16)  $Y_3 \leftarrow Y_3 - r_2$
(17)  $Q := (X_3 : Y_3 : Z_3)$
(18)  return $Q$
```

The field cost for Algorithm 8 is $4M + 2S$, a savings of 8 multiplications and 2 squarings compared to the algorithm for $\mathcal{J} + \mathcal{J} \rightarrow \mathcal{J}$. Only two temporary variables are required.

Algorithm 9 takes two affine points and a Jacobian point and returns their sum in Jacobian form.

```
Algorithm 9: Point Addition ($\mathcal{J} + \mathcal{A} \rightarrow \mathcal{J}$)
Input: Affine Points $P_1 = (X_1 : Y_1 : Z_1)$, $P_2 = (x_2, y_2)$
Output: Jacobian Point $P_1 + P_2 = (X_3 : Y_3 : Z_3)$
(1)   if $Z_1 = 0$ then return $P_2$
(2)   if $Z_2 = 0$ then return $P_1$
(3)   $r_1 \leftarrow Z_1^2$
(4)   $r_2 \leftarrow x_2 \cdot r_1$
(5)   $r_1 \leftarrow Z_1 \cdot r_1$
(6)   $r_1 \leftarrow y_2 \cdot r_1$
(7)   $r_1 \leftarrow r_1 - Y_1$
(8)   $r_2 \leftarrow r_2 - X_1$
(9)   if $r_2 = 0$
(10)      if $r_1 = 0$
```

```
(11)            $Q := 2P_1$ using $2\mathcal{A} \to \mathcal{J}$
(12)            **return** $Q$
(13)        **else**
(14)            $Q := (1 : 1 : 0)$
(15)            **return** $Q$
(16)    $Z_3 \leftarrow Z_1 \cdot r_2$
(17)    $r_3 \leftarrow r_2^2$
(18)    $r_2 \leftarrow r_2 \cdot r_3$
(19)    $r_3 \leftarrow r_3 \cdot X_3$
(20)    $X_3 \leftarrow r_1^2$
(21)    $Y_3 \leftarrow r_2 \cdot Y_3$
(22)    $r_2 \leftarrow X_3 - r_2$
(23)    $X_3 \leftarrow 2r_3$
(24)    $X_3 \leftarrow r_2 - X_3$
(25)    $r_2 \leftarrow r_3 - X_3$
(26)    $r_2 \leftarrow r_1 \cdot r_2$
(27)    $Y_3 \leftarrow r_2 - Y_3$
(28)    $Q := (X_3 : Y_3 : Z_3)$
(29)    **return** $Q$
```

The field cost for Algorithm 9 is $8M + 3S$, unless $P_1 = \pm P_2$. If $P_1 = P_2$, the field cost is $5M + 5S$ and if $P_1 = -P_2$, the field cost is $3M + S$. The algorithm requires three temporary variables.

There are numerous other combinations of coordinate systems, most of which are listed by Cohen *et al.* [20]. The costs marked with $*$ were derived by the author. The modified Jacobian coordinate system $\mathcal{J}^m$ was omitted because it does not provide any advantage over $\mathcal{J}$ when $a = -3$ in the Weierstraß Equation (2.4).

The results of Table 2.1 illustrate the advantages and disadvantages of each representation. Chudnovsky Jacobian coordinates provide the most efficient addition unless one of the points to be added is in affine coordinates. Jacobian coordinates provides the fastest doubling. For a large compound operation that requires significantly more doublings than additions, the Jacobian representation is likely to be

Table 2.1: Field Cost of Mixed Addition and Doubling

| Doubling | | Addition | |
|---|---|---|---|
| $2\mathcal{A} \to \mathcal{A}$ | $I + 2M + 2S$ | $\mathcal{A} + \mathcal{A} \to \mathcal{A}$ | $I + 2M + S$ |
| $2\mathcal{A} \to \mathcal{J}$ | $2M + 4S$ | $\mathcal{A} + \mathcal{A} \to \mathcal{J}$ | $4M + 2S$* |
| $2\mathcal{A} \to \mathcal{J}^c$ | $3M + 5S$ | $\mathcal{A} + \mathcal{J} \to \mathcal{J}$ | $8M + 3S$ |
| $2\mathcal{J} \to \mathcal{J}$ | $4M + 4S$ | $\mathcal{A} + \mathcal{J}^c \to \mathcal{J}$ | $7M + 2S$* |
| $2\mathcal{J} \to \mathcal{J}^c$ | $5M + 5S$ | $\mathcal{J} + \mathcal{J} \to \mathcal{J}$ | $12M + 4S$ |
| $2\mathcal{J}^c \to \mathcal{J}$ | $4M + 3S$ | $\mathcal{J} + \mathcal{J}^c \to \mathcal{J}$ | $11M + 3S$ |
| $2\mathcal{J}^c \to \mathcal{J}^c$ | $5M + 4S$ | $\mathcal{J}^c + \mathcal{J}^c \to \mathcal{J}$ | $10M + 2S$ |
| | | $\mathcal{A} + \mathcal{A} \to \mathcal{J}^c$ | $5M + 3S$ |
| | | $\mathcal{A} + \mathcal{J} \to \mathcal{J}^c$ | $9M + 4S$* |
| | | $\mathcal{A} + \mathcal{J}^c \to \mathcal{J}^c$ | $8M + 3S$ |
| | | $\mathcal{J} + \mathcal{J} \to \mathcal{J}^c$ | $13M + 5S$* |
| | | $\mathcal{J} + \mathcal{J}^c \to \mathcal{J}^c$ | $12M + 4S$* |
| | | $\mathcal{J}^c + \mathcal{J}^c \to \mathcal{J}^c$ | $11M + 3S$ |

more efficient while Chudnovsky Jacobian or affine-Jacobian coordinates are likely to be more efficient for a compound operation requiring more additions than doublings.

The conversion from $(X : Y : Z)$ in $\mathcal{J}$ to $(x, y)$ in $\mathcal{A}$ with associated costs in parentheses is as follows:

1. $Z^{-1}$ is computed $(I)$.

2. $(Z^{-1})^2$, $(Z^{-1})^3$ are computed $(M + S)$.

3. $x = X \cdot (Z^{-2})$ and $y = X \cdot (Z^{-3})$ are computed $(2M)$.

The total is $I+3M+S$. Some of the most useful operations that will be utilized in subsequent sections, with their respective field costs in parentheses, are the following:

- $2\mathcal{A} \to \mathcal{J}$, $(2M + 4S)$: This is the fastest doubling operation; initial points are often given in affine coordinates.

- $2\mathcal{J} \rightarrow \mathcal{J}$, $(4M + 4S)$: In most of the algorithms for scalar multiplication, doubling will be featured more heavily than addition, therefore fast doubling is essential. This is the most efficient algorithm for doubling intermediate points when $a = -3$.

- $\mathcal{A} + \mathcal{A} \rightarrow \mathcal{J}$, $(2M + 4S)$: This is the fastest addition operation; initial points are often given in affine coordinates.

- $\mathcal{A} + \mathcal{J} \rightarrow \mathcal{J}$, $(8M + 3S)$: The addition of points given in affine form with intermediate points in Jacobian form will be needed in certain scalar multiplication algorithms.

- $\mathcal{J} + \mathcal{J} \rightarrow \mathcal{J}$, $(12M + 4S)$: This addition algorithm is used for adding intermediate points that are in Jacobian form.

- $\mathcal{J}^c + \mathcal{J}^c \rightarrow \mathcal{J}^c$, $(11M + 3S)$: This algorithm is more efficient for the addition of intermediate points, but requires a conversion from Jacobian to Chudnovsky Jacobian coordinates. This addition is preferable when a long string of additions needs to be performed.

- $\mathcal{J}^c + \mathcal{J}^c \rightarrow \mathcal{J}$, $(10M + 2S)$: This operation can be used to terminate the long string of additions mentioned above and return the value in Jacobian coordinates.

- $\mathcal{J} \rightarrow \mathcal{J}^c$, $(M + S)$: This operation is used to convert Jacobian coordinates to Chudnovsky Jacobian coordinates when a number of repeated additions is needed such as in certain precomputations.

- $\mathcal{J} \to \mathcal{A}$, $(I + 3M + S)$: This operation is needed to convert the intermediate points from Jacobian coordinates to affine coordinates.

### 2.4.4 Specialized Formulas

In this section, we focus on formulas for operations that involve two or more basic operations performed in sequence. Using custom formulas, certain operations can be performed faster when performed in sequence. Examples of this include a double followed by an add or an addition of a point to itself twice (tripling). If we treat these compound operations as a single operation, custom formulas can be found that reduce their complexity. Just like for doubling and addition, we will denote the sequenc double then add by $2A + B \to C$ and triple by $3A \to C$ for points given in coordinate systems $A, B$ and $C$.

The formulas for $3\mathcal{A} \to \mathcal{A}$ were introduced by Eisentrager *et al.* [31] and improved by Ciet *et al.* [19]. These formulas were used by Dimitrov *et al.* [28] to derive the following formula for tripling in Jacobian coordinates. Given $P = (X_1 : Y_1 : Z_1)$, we compute $3P = (X_3 : Y_3 : Z_3)$ as

$$M = 3X_1^2 + aZ_1^4,$$

$$E = 12X_1Y_1^2 - M^2, T = 2Y_1^2 \cdot 4Y_1^2,$$

$$X_3 = 8Y_1^2(T - ME) + X_1E^2,$$

$$Y_3 = Y_1(4(ME - T)(2T - ME) - E^3), \text{ and}$$

$$Z_3 = Z_1E.$$

This formula computes the triple of a point for the case where $a$ is an arbitrary value using $10M + 6S$ operations. A repeated version that computes $3^k\mathcal{J} \to \mathcal{J}$ was described, with field cost $(11k-1)M+(4k+2)S$. The authors did not examine the case where $a=-3$. In this case, it is possible to reduce the cost of tripling an affine point to $9M+5S$. We used this fact to derive two new algorithms for point tripling, Algorithms 10 and 11, that are not found in the literature. Algorithm 10 takes a point in affine form and returns its triple in Jacobian form. We will make the plausible assumption that $P$ does not have order 3.

```
**Algorithm 10:** Point Tripling ($3\mathcal{A} \rightarrow \mathcal{J}$)
**Input:** Affine Point $P_1 = (x_1, y_1)$
**Output:** Jacobian Point $3P_1 = Q = (X_3 : Y_3 : Z_3)$
(1)    **if** $Z_1 = 0$ or $y_1 = 0$ **then return** $P_1$
(2)    $r_1 \leftarrow X_1^2$
(3)    $r_1 \leftarrow r_1 + (-1)$
(4)    $r_2 \leftarrow 2r_1$
(5)    $r_2 \leftarrow r_2 + r_1$
(6)    $r_1 \leftarrow Y_1^2$
(7)    $r_1 \leftarrow 2r_1$
(8)    $r_3 \leftarrow 2r_1$
(9)    $r_3 \leftarrow X_1 \cdot r_3$
(10)   $Y_3 \leftarrow 2r_3$
(11)   $r_3 \leftarrow r_3 + Y_3$
(12)   $Y_3 \leftarrow r_2^2$
(13)   $Y_3 \leftarrow -Y_3$
(14)   $r_3 \leftarrow r_3 + Y_3$
(15)   $r_4 \leftarrow r_1^2$
(16)   $r_4 \leftarrow 2r_4$
(17)   $Y_3 \leftarrow r_3^2$
(18)   $Z_3 \leftarrow X_1 \cdot Y_3$
(19)   $r_2 \leftarrow r_2 \cdot r_3$
(20)   $r_2 \leftarrow -r_2$
(21)   $r_2 \leftarrow r_2 + r_5$
(22)   $X_3 \leftarrow r_1 \cdot r_2$
(23)   $X_3 \leftarrow 2X_3$
(24)   $X_3 \leftarrow 2X_3$
(25)   $X_3 \leftarrow X_3 + Z_3$
(26)   $r_1 \leftarrow r_1 + r_2$
```

```
(27)  $r_2 \leftarrow -r_2$
(28)  $Z_3 \leftarrow Z_3 \cdot r_3$
(29)  $r_1 \leftarrow r_1 \cdot r_2$
(30)  $r_1 \leftarrow 2r_1$
(31)  $r_1 \leftarrow 2r_1$
(32)  $Y_3 \leftarrow r_1 - Z_3$
(33)  $Y_3 \leftarrow Y_3 \cdot Y_1$
(34)  $Z_3 \leftarrow Z_1$
(35)  $Q := (X_3 : Y_3 : Z_3)$
(36)  **return** $Q$
```

Algorithm 10 takes $7M + 5S$ and 4 temporary variables or $S$ when $2P = \infty$.

**Algorithm 11:** Point Tripling ($3\mathcal{J} \rightarrow \mathcal{J}$)
**Input:** Jacobian Points $P_1 = (X_1 : Y_1 : Z_1)$
**Output:** Jacobian Point $3P_1 = Q = (X_3 : Y_3 : Z_3)$

```
(1)   **if** $Z_1 = 0$ **then return** $P_1$
(2)   $r_1 \leftarrow Z_1^2$
(3)   $r_2 \leftarrow X_1 - r_1$
(4)   $r_1 \leftarrow X_1 + r_1$
(5)   $r_1 \leftarrow r_1 \cdot r_2$
(6)   $r_2 \leftarrow 2r_1$
(7)   $r_2 \leftarrow r_2 + r_1$
(8)   **if** $r_2 = 0$ **then return** $(1 : 1 : 0)$
(9)   $r_1 \leftarrow Y_1^2$
(10)  $r_1 \leftarrow 2r_1$
(11)  $r_3 \leftarrow 2r_1$
(12)  $r_3 \leftarrow X_1 \cdot r_3$
(13)  $Y_3 \leftarrow 2r_3$
(14)  $r_3 \leftarrow r_3 + Y_3$
(15)  $Y_3 \leftarrow r_2^2$
(16)  $Y_3 \leftarrow -Y_3$
(17)  $r_3 \leftarrow r_3 - Y_3$
(18)  $r_4 \leftarrow r_1^2$
(19)  $r_4 \leftarrow 2r_4$
(20)  $Y_3 \leftarrow r_3^2$
(21)  $Z_3 \leftarrow X_1 \cdot Y_3$
(22)  $r_2 \leftarrow r_2 \cdot r_3$
(23)  $r_2 \leftarrow -r_2$
(24)  $r_2 \leftarrow r_2 + r_5$
```

```
(25)  $X_3 \leftarrow r_1 \cdot r_2$
(26)  $X_3 \leftarrow 2X_3$
(27)  $X_3 \leftarrow 2X_3$
(28)  $X_3 \leftarrow X_3 + Z_3$
(29)  $r_1 \leftarrow r_1 + r_2$
(30)  $r_2 \leftarrow -r_2$
(31)  $Z_3 \leftarrow Z_3 \cdot r_3$
(32)  $r_1 \leftarrow r_1 \cdot r_2$
(33)  $r_1 \leftarrow 2r_1$
(34)  $r_1 \leftarrow 2r_1$
(35)  $Y_3 \leftarrow r_1 - Z_3$
(36)  $Y_3 \leftarrow Y_3 \cdot Y_1$
(37)  $Z_3 \leftarrow Z_1 \cdot r_3$
(38)  $Q := (X_3 : Y_3 : Z_3)$
(39)  **return** $Q$
```

Algorithm 11 has a field cost of $9M+5S$ and uses 4 temporary variables, a savings of $M+S$ and 3 temporary variables when compared with the formula by Dimitrov *et al.* [28].

Formulas for $2\mathcal{A}+\mathcal{A} \to \mathcal{A}$ were introduced by Eisentrager *et al.* [31] and improved by Ciet *et al.* [19]. This algorithm was used to develop algorithms for $2\mathcal{A}+\mathcal{A} \to \mathcal{J}$ and $2\mathcal{A}+\mathcal{J} \to \mathcal{J}$ that take fewer field operations than any combination of doubling and addition algorithms. These are not included because they are not used in later chapters of the thesis and are generally not useful for elliptic curve scalar multiplication.

Table 2.2 illustrates the field cost of various special operations. The costs marked with $*$ were derived by the author. There are no known custom algorithms for $2\mathcal{J}+\mathcal{A} \to \mathcal{J}$ and $2\mathcal{J}+\mathcal{J} \to \mathcal{J}$ that take fewer operations than the combination of a double and an addition. The cost of these operations in Table 2.2 are given as the sum of the costs of the corresponding double and addition.

Table 2.2: Field Cost of Special Operations

| Operations | Field Cost |
|---|---|
| $2\mathcal{A} + \mathcal{A} \rightarrow \mathcal{A}$ | $I + 9M + 2S$ |
| $2\mathcal{A} + \mathcal{A} \rightarrow \mathcal{J}^*$ | $11M + 3S$ |
| $2\mathcal{A} + \mathcal{J} \rightarrow \mathcal{J}^*$ | $15M + 4S$ |
| $2\mathcal{J} + \mathcal{A} \rightarrow \mathcal{J}$ | $12M + 7S$ |
| $2\mathcal{J} + \mathcal{J} \rightarrow \mathcal{J}$ | $16M + 8S$ |
| $3\mathcal{A} \rightarrow \mathcal{A}$ | $I + 7M + 4S$ |
| $3\mathcal{A} \rightarrow \mathcal{J}^*$ | $7M + 5S$ |
| $3\mathcal{J} \rightarrow \mathcal{J}^*$ | $9M + 5S$ |

# Chapter 3

# Elliptic Curve Scalar Multiplication

Scalar multiplication, adding a point $P$ to itself $k$ times, is an important operation on the group of points on an elliptic curve, especially for ECC. In this chapter, we describe, analyze and compare several algorithms to compute $kP$.

Scalar multiplication requires a point and an integer with which to multiply it. In certain situations, either the point or the multiplier is known in advance. The majority of the algorithms presented in this chapter are designed for the case in which neither the point nor the scalar multiplier are known in advance. This is called the *unknown point* case and any precomputation involving the point to be multiplied is included as part of the operation. We also describe algorithms for the *known point* case and the *known multiplier* case.

The unknown point methods introduced in this chapter are related to the classical binary scalar multiplication algorithm (see Knuth [52, Sec. 4.20]). The binary method computes the scalar multiple of a point by a sequence of successive doublings and additions determined by the digits of the binary representation of the multiplier. Improvements to this algorithm can be made by taking different binary representations of the multiplier such as the non-adjacent form (NAF) or the width $w$ non-adjacent form ($w$-NAF). These and similar representations lead to the binary NAF (Section 3.2.2), window NAF (3.2.3), sliding window (3.2.4), and fractional window (3.2.5) algorithms. An algorithm based on double-base number systems (3.2.6) generalizes the binary algorithm further.

In the known point or known multiplier cases, some steps can be taken to speed up scalar multiplication. We discuss techniques used in these special circumstances in Sections 3.3 and 3.4. The main techniques for the known point case are the fixed-base windowing technique and the comb method. These involve expensive precomputations and provide a significant speedup compared to the unknown point algorithms. The known multiplier algorithms rely on generating an addition chain for the multiplier.

The algorithms in this chapter will be analyzed in terms of two levels of complexity. First, we present the expected number of elliptic curve operations needed and second, the number of finite field operations needed. The number of finite field operations is determined by specifying the point representations and the elliptic curve operations used in the algorithms. The analyses in this chapter are all new. These estimates are average case estimates unless otherwise noted.

## 3.1 Conventions

In this chapter, we deal with elliptic curves over prime fields. Suppose that $E/\mathbb{F}_p$ is such an elliptic curve where $p$ is a large prime. In this case, $E(\mathbb{F}_p)$ is the group of rational points. The focus of this chapter is computing the scalar multiple $kP$ of a point $P \in E(\mathbb{F}_p)$ where $k \in \mathbb{N}$. The time it takes to perform the scalar multiplication operation is related to the size of the integer $k$. We focus on the size of various representations of $k$, where the integer $d = \lfloor \log_2 k \rfloor + 1$ is the bit-length of the binary representation of $k$.

In this chapter, we will deal with numbers and their base 2 representations.

Integers are represented using finite sums of the form $\sum_{i\geq 0} a_i 2^i$. If $n$ is an integer and can be written as a finite sum $n = \sum_{i\geq 0} a_i 2^i$, we call $n = \sum_{i\geq 0} a_i 2^i$ a *base-2 representation* of $n$. The following notation is used to denote base 2 representations:

$$(\cdots, a_2, a_1, a_0)_2 = \cdots + a_2 2^2 + a_1 2^1 + a_0.$$

Each $a_i$ is called a *digit*. In the usual binary system, each digit is equal to 0 or 1. In this chapter, we deal with alternative systems for which the digits and the base can take on different values.

The running time analysis of scalar multiplication is based on two levels of complexity, elliptic curve operations and finite field operations. For the number of elliptic curve additions and doublings, the time needed to perform an addition will be denoted $A$, the time for a doubling $D$ and the time needed for a tripling $T$. For finite field operations, as before, a finite field multiplication will be denoted $M$, a squaring $S$ and an inversion $I$. Operations such as finite field addition, negation and doubling that take a negligible amount of time in comparison to multiplication will not be counted.

As explained in Section 2.2, there are different representations of elliptic curve points. Elliptic curve operations have different costs in different representations. For this discussion, we will only consider points in Chudnovsky Jacobian form ($\mathcal{J}^c$), Jacobian form ($\mathcal{J}$), and affine form ($\mathcal{A}$). Projective coordinates ($\mathcal{P}$) are not used because addition and doubling are both slower in this form than in Chudnovsky Jacobian form.

In the detailed analysis of an algorithm, we describe each step of the algorithm in terms of operations listed in Tables 2.1 and 2.2. Every elliptic curve operation has an associated cost in terms of the number of operations required in the base field. The total cost of an algorithm is the total of the costs of the required group operations. This total is called the *field cost* and the field cost with substitutions $S = (4/5)M$ and $I = 80M$ will be called the $M$-cost. These estimates are standard and are justified in Section A.4 and based on the findings of Brown *et al.* [16]. It must be emphasized that the field costs presented in this chapter are *average case* estimates. We must also note that while the standard substitution for $I$ is $80M$, it may be as low as $30M$ in certain implementations and this consideration is noted when relevant.

In 1999, NIST [78] published a set of recommended primes for prime field elliptic curve cryptography that can be found in Section A.3.3 of the Appendix. These primes are denoted by $P192$, $P224$, $P256$, $P384$ and $P521$ and their bit-lengths are 192, 224, 256, 384 and 521, respectively. These primes are chosen because they have a specific form that permits faster modular reduction (see Appendix A for more details). In elliptic curve cryptography, the scalars used in scalar multiplication are approximately the same size as the underlying field of the curve. In each section of this chapter, we examine the field cost and $M$-cost of the scalar multiplication algorithms for multipliers of sizes $192, 224, 256, 384$ and 521 bits, in order to correspond with curves over the corresponding NIST primes. In some algorithms, the multiplier is given in a base 2 representation other than the standard binary representation. In such cases, we assume the multipliers have length $192, 224, 256, 384$ and 521 so that they have the same length as the corresponding NIST primes.

## 3.2 Unknown Point Scalar Multiplication

In this section, we present algorithms for elliptic curve scalar multiplication when neither the point nor the multiplier are known in advance. A general overview of these algorithms is presented by Gordon [39].

Most of the algorithms in this section are variations of the binary algorithm presented in Section 3.2.1. The variations improve efficiency by utilizing other representations of integers for which the digits can be taken from a set larger than $\{0, 1\}$. Including $-1$ in the set of digits lead to the NAF algorithm (Section 3.2.2); using larger digits leads to the window NAF, sliding window, and fractional window algorithms (Sections 3.2.3, 3.2.4 and 3.2.5). The binary algorithm is generalized for binary-ternary representations of integers in the double-base chain algorithm (3.2.6). All the algorithms presented in this section are found in the literature in one form or another.

### 3.2.1 Binary Method

The naïve method for computing $kP$ from $P$ is to add $P$ to itself $k$ times. This method is inefficient since it takes $k$ elliptic curve operations. Binary exponentiation is a simple method for exponentiation that requires on the order of $\log(k)$ elliptic curve operations. This algorithm is equivalent to the square-and-multiply method for modular exponentiation (see Gordon [39]) but described in additive form. It is sometimes referred to as the *double-and-add* algorithm. The algorithm relies on the fact that every integer $k$ has a binary representation $(k_{d-1}, \ldots, k_0)_2$ such that $k = k_{d-1}2^{d-1} + \cdots + k_1 2 + k_0$ with $k_{d-1} \neq 0$ and $k_i \in \{0, 1\}$. Using Horner's rule [52, Ch. 4] we can write

$$kP = 2(2 \ldots (2k_{d-1}P + k_{d-2}P) + \cdots + k_1P) + k_0P,$$

allowing $kP$ to be computed using a sequence of doublings and additions.

In the right-to-left variant of this algorithm, based on Horner's rule, the computation begins with the least significant digits. Each binary power multiple of $P$ is computed, and is added to the running total when the corresponding digit is 1. Algorithm 12 is adapted from Gordon [39].

```
Algorithm 12: Right-to-Left Binary Scalar Multiplication
Input: Affine point $P$, positive integer $k$ with binary representation
$(k_{d-1}, \ldots, k_0)_2$
Output: Affine point $kP$
(1)   $Q \leftarrow \infty$
(2)   $T \leftarrow P$
(3)   for $i = 0$ down to $d - 1$
(4)       if $k_i \neq 0$ then $Q \leftarrow Q + T$
(5)       $T \leftarrow 2T$
(6)   return $Q$
```

The left-to-right variant of this algorithm begins with the most significant digits. The point $P$ is added to the running total for each non-zero digit and the total is doubled for every digit. Algorithm 13 is adapted from Gordon [39].

```
Algorithm 13: Left-to-Right Binary Scalar Multiplication
Input: Affine point $P$, positive integer $k$ with binary representation
$(k_{d-1}, \ldots, k_0)_2$
Output: Affine point $kP$
(1)   $Q \leftarrow P$
(2)   for $i = d - 2$ to $0$
(3)       if $k_i \neq 1$
(4)           $Q \leftarrow 2Q + P$
(5)       else
(6)           $Q \leftarrow 2Q$
(7)   return $Q$
```

Algorithms 12 and 13 both take $d-1$ squarings and $H(k)-1$ additions where $d$ is the bit-length of $k$ and $H(k)$ is the Hamming weight (i.e. the number of non-zero digits) of $k$. $H(k)$ is $(d+1)/2$ on average.

A more detailed analysis can be performed by determining the type of representation used for the elliptic curve points and the specific operations used to perform the computation. The following is a description of each step of the previous algorithms using operations from Tables 2.1 and 2.2. In Algorithm 12, there are approximately twice as many doublings as additions. The coordinate system with the fastest doubling is Jacobian coordinates, so this is the coordinate system we will use. Another option is Chudnovsky Jacobian coordinates, saving one multiplication and one squaring per addition, and costing an extra multiplication per doubling in comparison to Jacobian coordinates. Since $M > S$, this coordinate system is slower. Moreover, each point is represented by 5 finite field elements instead of 3.

Step 4 of Algorithm 12 defines $Q$ the first time it is reached. The second time it is reached, it is performed with $\mathcal{A}+\mathcal{J} \to \mathcal{J}$ if $k$ is odd and $\mathcal{J}+\mathcal{J} \to \mathcal{J}$ if $k$ is even. Every other time, $\mathcal{J}+\mathcal{J} \to \mathcal{J}$ is used. Step 5 is performed with $2\mathcal{A} \to \mathcal{J}$ the first time and $2\mathcal{J} \to \mathcal{J}$ the rest of the time. The result of the algorithm is in Jacobian form, so $\mathcal{J} \to \mathcal{A}$ is needed to return the required affine point. The mixed affine-Jacobian operations are used because the initial points are given in affine coordinates.

With this implementation, Algorithm 12 takes

$$\left(\frac{d-1}{2}\right) A + (d-1)D,$$

which has field cost

$$\left(\frac{d-3}{2}\right)(12M+4S)+(d-2)(4M+4S)+(8M+3S)+(2M+4S)+(I+3M+S)$$
$$= I+(10d-13)M+(6d-6)S$$

when $k$ is odd and

$$\left(\frac{d-1}{2}\right)(12M+4S)+(d-2)(4M+4S)+(2M+4S)+(I+3M+S)$$
$$= I+(10d-9)M+(6d-5)S$$

when $k$ is even.

Step 4 of Algorithm 13 is performed with $3\mathcal{A} \to \mathcal{J}$ if $i = d-2$, and $2\mathcal{J}+\mathcal{A} \to \mathcal{J}$ otherwise. Step 6 is performed with $2\mathcal{A} \to \mathcal{J}$ if $i = d-2$, and $2\mathcal{J} \to \mathcal{J}$ otherwise. The result of the algorithm is in Jacobian form, so $\mathcal{J} \to \mathcal{A}$ is needed.

Algorithm 13 has field cost

$$\left(\frac{d-3}{2}\right)(12M+7S)+(8M+4S)+\left(\frac{d-1}{2}\right)(4M+4S)+(I+3M+S)$$
$$= I+(8d-5)M+\left(\frac{11}{2}d+\frac{11}{2}\right)S$$

when $k_{d-2}$ is 1 and

$$\left(\frac{d-1}{2}\right)(12M+7S)+(2M+4S)+\left(\frac{d-3}{2}\right)(4M+4S)+(I+3M+S)$$
$$= I+(8d-7)M+\left(\frac{11}{2}d-\frac{11}{2}\right)S$$

when $k_{d-2}$ is 0.

If we suppose that both $k_{d-2}$ and $k_0$ are 1 half the time as one would expect on average, then Table 3.1 describes the average field cost of these operations. The

Table 3.1: Binary Method Average Cost

| $d$ | R2L | | L2R | |
|---|---|---|---|---|
| | Field Cost | $M$-cost | Field Cost | $M$-cost |
| 192 | I+1909.0M+1146.5S | 2906.2M | I+1516.0M+1043.0S | 2430.4M |
| 224 | I+2229.0M+1338.5S | 3379.8M | I+1772.0M+1219.0S | 2827.2M |
| 256 | I+2549.0M+1530.5S | 3853.4M | I+2028.0M+1395.0S | 3224.0M |
| 384 | I+3829.0M+2298.5S | 5747.8M | I+3052.0M+2099.0S | 4811.2M |
| 521 | I+5199.0M+3120.5S | 7775.4M | I+4148.0M+2852.5S | 6510.0M |

left-to-right binary algorithm is more efficient in this context because the point that is repeatedly added is the initial point $P$, given in affine form, which allows the faster Jacobian-affine addition formulas to be used.

### 3.2.2 Binary NAF

The signed binary representation is a redundant system for representing an integer. The standard base 2 representation for an integer $k$ is

$$k = \sum_{i=0}^{d-1} k_i 2^i,$$

where $k_i \in \{0, 1\}$. This representation is unique. A signed binary system is a base 2 representation that allows $k_i \in \{-1, 0, 1\}$. Let $(k_{d-1}, \ldots, k_0)$ denote a sequence of digits of such an integer $k$, where $k_{d-1} \neq 0$. The notation $\overline{k}_i$ will represent the integer $-k_i$. Signed binary representations are highly redundant, for example

$$3 = (1, 1) = (1, 0, \overline{1}) = (1, \overline{1}, 0, \overline{1}),$$

in such a representation. If we make the additional restriction that no two consecutive digits can be non-zero, we find that this restricted signed binary representation not only exists, it is unique for all integers. A useful fact is that this representation requires at most one additional digit when compared to the standard binary representation. This representation is called the *non-adjacent form (NAF)* and its properties, summarized below in Theorem 3.2.1, were examined in depth by Bosma [13].

**Theorem 3.2.1.** *Let $k$ be a positive integer.*

1. *$k$ has a unique NAF denoted NAF$(k)$.*
2. *NAF$(k)$ has the fewest non-zero digits of any signed digit binary representation of $k$.*
3. *The length of NAF$(k)$ is at most one more than the length of the binary representation of $k$.*
4. *If the length of NAF$(k)$ is $l$, then $2^l/3 < k < 2^{l+1}/3$.*
5. *The average density of nonzero digits among all NAFs of length $l$ is $1/3$.*

Since the first digit $k_{d-1}$ of an NAF representation for a positive integer $k$ is 1, then $k_{d-2}$ is forced to be 0. Computing the NAF representation of an integer $k$ is a straightforward process performed by Algorithm 14. This algorithm is adapted from Hankerson *et al.* [43, Alg. 3.35].

```
Algorithm 14: Computing the NAF of a Positive Integer
Input: Positive integer $k$
Output: NAF($k$) as a signed binary representation $(k_{d-1}, \dots, k_0)$
(1)  $i \leftarrow 0$
(2)  while $k \geq 1$
(3)      if $k$ is odd
(4)          $k_i \leftarrow 2 - (k \bmod 4)$
(5)          $k \leftarrow k - k_i$
```

```
(6)         else
(7)             $k_i \leftarrow 0$
(8)             $k \leftarrow k/2, i \leftarrow i + 1$
(9)     return $(k_{d-1}, \ldots, k_0)$
```

Another signed binary encoding that was recently introduced by Okeya *et al.* [81] is the *mutual opposite form* (MOF). This representation has many of the same attributes as the NAF including length expansion and average density. The significant difference is that it is computed left-to-right rather than right-to-left. This could potentially lead to improvements as the MOF can be computed more easily on the fly. See Okeya *et al.* [81] for more details on the encoding. Since the density of this representation is identical to that of the NAF, we will not consider separate algorithms for left-to-right NAF scalar multiplication and left-to-right MOF scalar multiplication.

In the binary representation, the operation performed for each digit is either $2P+Q$ or $2P$. When $-1$ is allowed for a digit, the operations performed are $2P+Q$, $2P$ or $2P-Q$. The fact that negation is a fast operation in the group of rational points on an elliptic curve allows $2P-Q$ to be computed in essentially the same amount of time as $2P+Q$. This allows for the binary algorithm to be adapted to using a signed digit representation with no additional computational cost. Algorithm 15 computes a scalar multiple of a point given the NAF of the scalar, adapted from Gordon [39].

```
Algorithm 15: Left-to-Right NAF
Input: Affine point $P$, positive integer $k$ with $\mathrm{NAF}(k) = (k_{d-1}, \ldots, k_0)$
Output: Affine point $kP$
(1)     $Q \leftarrow 2P$
(2)     for $i = d - 3$ to $0$
(3)         if $k_i = 1$
```

```
(4)            $Q \leftarrow 2Q + P$
(5)        **else if** $k_i = -1$
(6)            $Q \leftarrow 2Q - P$
(7)        **else**
(8)            $Q \leftarrow 2Q$
(9)    **return** $Q$
```

The average proportion of non-zero digits in an NAF representation is 1/3 according to Bosma [13]. Therefore the approximate average running time of this algorithm will be

$$\left(\frac{d}{3} - 1\right) A + (d-1)D,$$

where $d$ is the length of $k$.

In Algorithm 15, there are approximately three times as many doublings as additions. The coordinate system with the fastest doubling is Jacobian coordinates, so this is the coordinate system we will use. Using the operations from Tables 2.1 and 2.2, a more detailed analysis of the previous algorithm can be performed.

Step 1 of Algorithm 13 is performed with $2\mathcal{A} \to \mathcal{J}$. Steps 4 and 6 are performed with $2\mathcal{J} + \mathcal{A} \to \mathcal{J}$, Step 8 is performed with $2\mathcal{J} \to \mathcal{J}$. The result of the algorithm is in Jacobian form, so $\mathcal{J} \to \mathcal{A}$ is needed to return the result in affine form.

This algorithm takes the following number of operations on average:

$$\begin{aligned}\left(\frac{d-3}{3}\right)(12M+7S) + (2M+4S) + \left(d - 2 - \frac{d-3}{3}\right)(4M+4S) + (I+3M+S)\\ = I + \left(\frac{20}{3}d - 11\right) M + \left(\frac{15}{3}d - 6\right) S.\end{aligned}$$

Table 3.2 describes the average field cost of NAF scalar multiplication for $d$ corresponding to the 5 NIST primes, assuming $d/3$ non-zero terms. The NAF method requires no extra storage space while remaining simple and quite fast. Because of

Table 3.2: Binary NAF Method Average Cost

| $d$ | Field Cost | $M$-cost |
|---|---|---|
| 192 | I+1269.0M+954.0S | 2112.2M |
| 224 | I+1482.3M+1114.0S | 2453.5M |
| 256 | I+1695.7M+1274.0S | 2794.9M |
| 384 | I+2549.0M+1914.0S | 4160.2M |
| 521 | I+3462.3M+2599.0S | 5621.5M |

these properties, the NAF method is considered a benchmark for scalar multiplication. In the following sections, a number of algorithms will be described that are faster, but require either a certain amount of storage space or a more complicated precomputation.

### 3.2.3 Window NAF

Precomputation can potentially provide a speed boost in comparison to the NAF method. Given $w \geq 2$, a *width-$w$ NAF* (or simply $w$-NAF) of an integer $k$ is a base 2 representation

$$k = \sum_{i=0}^{d-1} k_i 2^i,$$

where $w \geq 2$, $k_i$ is odd, $|k_i| < 2^{w-1}$ for all $i$ and at most one of any $w$ consecutive digits is non-zero. The set of possible values for $k_i$ is called the *digit set* of the representation. Notice that when $w = 2$, this corresponds to the previously introduced NAF representation.

Every $k \in \mathbb{N}$ has a unique width-$w$ NAF, denoted $NAF_w(k)$ (see Muir and Stinson [76]). This representation is a generalization of the NAF representation. Theorem 3.2.2 summarizes the properties of $w$-NAF from Muir and Stinson [76].

**Theorem 3.2.2.** *Let $k, w$ be positive integers.*

1. *$k$ has a unique width-$w$ NAF.*
2. *$NAF_2(k) = NAF(k)$.*
3. *The length of $NAF_w(k)$ is at most one more than the length of the binary representation of $k$.*
4. *The average density of nonzero digits among all width-$w$ NAFs of length $l$ is $1/(w+1)$.*

Möller [71] and Bosma [13] discuss methods to eliminate the length expansion of a $w$-NAF expansion in around half the cases.

As with the NAF, there is a left-to-right analogue of the $w$-NAF, namely the $w$-MOF described in Okeya *et al.* [81]. This is simply the sliding window technique described in Section 3.2.4 performed on the MOF of a number. We will not examine this further since the $w$-MOF has the same non-zero density as the $w$-NAF.

Algorithm 16 computes the width-$w$ NAF of an integer, from Hankerson *et al.* [43, Alg. 3.35].

```
Algorithm 16: Computing the Width-$w$ NAF of an Integer
Input: Positive integers $k$,$w$
Output: Width-$w$ NAF of $k$, $(k_{i-1}, \ldots, k_0)$
(1)   $i \leftarrow 0$
(2)   while $k \geq 1$
(3)      if $k$ is odd then $k_i \leftarrow k \bmod 2^w$ (signed), $k \leftarrow k - k_i$
(4)                else $k_i \leftarrow 0$
(5)      $k \leftarrow k/2$, $i \leftarrow i + 1$
(6)   return $(k_{i-1}, \ldots, k_0)$
```

The digit set $B$ for this representation is the odd integers between $-2^{w-1}$ and $2^{w-1}$. If the points $bP$ are precomputed for every positive $b$ in the digit set $B$, the binary algorithm can again be applied. The advantage is that in this case, the number of non-zero digits and hence the number of additions needed is lower. This leads to Algorithm 17 for computing a scalar multiple of a point, given the $w$-NAF of the multiplier, adapted from Hankerson *et al.* [43, Alg. 3.36].

**Algorithm 17:** Left-to-Right Window NAF
**Input:** Positive integer $k$ with $\mathrm{NAF}_w(k) = (k_{d-1}, \ldots, k_0)$, affine point $P$
**Output:** Affine point $kP$

```
(1)   Precomputation:
(2)   Set $P_1 \leftarrow P$
(3)   $P_2 \leftarrow 2P$
(4)   foreach $i \in \{3, 5, \ldots, 2^{w-1} - 1\}$
(5)       $P_i \leftarrow P_{i-2} + P_2$
(6)   Computation:
(7)   $Q \leftarrow P_{d-1}$
(8)   for $i = d - 2$ to 0
(9)       if $k_i > 0$
(10)          $Q \leftarrow 2Q + P_{k_i}$
(11)      else if $k_i < 0$
(12)          $Q \leftarrow 2Q - P_{k_i}$
(13)      else
(14)          $Q \leftarrow 2Q$
(15)  return $Q$
```

The density of the non-zero width-$w$ NAFs of length $d$ is approximately $d/(w+1)$. This means that the running time of Algorithm 17 is approximately

$$(D + (2^{w-2} - 1)A) + \left(\left(\frac{d}{w+1} - 1\right) A + (d-1)D\right).$$

Algorithm 17 also requires the storage of $2^{w-2} - 1$ points.

There are many possibilities for the coordinate system in this algorithm. First, let us examine the possibilities for the precomputation stage. The precomputation consists of a large number of additions by the value $2P$, and the resulting values are used in additions in the main stage. This suggests that the running total should be kept in Chudnovsky Jacobian coordinates for faster addition. There are two possibilities for the representation of $2P$; it is either computed as $2\mathcal{A} \rightarrow \mathcal{J}^c$ or $2\mathcal{A} \rightarrow \mathcal{A}$. In the former case, every addition in the precomputation will be of the form $\mathcal{J}^c + \mathcal{J}^c \rightarrow \mathcal{J}^c$, and cost $11M + 3S$. If $2P$ is stored in affine coordinates, every addition will use $\mathcal{J}^c + \mathcal{A} \rightarrow \mathcal{J}^c$, and cost $8M + 3S$. The trade-off is that $2\mathcal{A} \rightarrow \mathcal{A}$ has an expensive inversion step. In fact, even if we assume $I$ is as low as $30M$, then $w$ must be at least 6 for this trade-off to be more efficient. Therefore, we will assume that for Algorithm 17, Step 3 is performed with $2\mathcal{A} \rightarrow \mathcal{J}^c$, Step 5 is performed with $\mathcal{A} + \mathcal{J}^c \rightarrow \mathcal{J}^c$ for $i = 3$, and $\mathcal{J}^c + \mathcal{J}^c \rightarrow \mathcal{J}^c$ otherwise.

For the main computation, there are approximately $w+1$ times as many doublings as additions, therefore Jacobian coordinates are used for the temporary variable. If the precomputed values are kept in Chudnovsky Jacobian coordinates, the double-and-add (for digits other than $\pm 1$) will be performed with $2\mathcal{J} + \mathcal{J}^c \rightarrow \mathcal{J}$, and cost $15M + 7S$. Another option is to use simultaneous inversion to convert the precomputed values to affine coordinates. Montgomery [75] introduced a method for computing the inverses of $k$ field elements using only one inverse and $3(k-1)$ multiplications. The operation is described in Algorithm 54 in Section A.3.4 of the Appendix. Converting the precomputed points to affine coordinates would allow each addition to use $2\mathcal{J} + \mathcal{A} \rightarrow \mathcal{J}$, costing only $12M + 7S$. The trade-off for this is dependent on the size of the precomputed set and number of additions required. The optimal scheme to use for different parameters is identified in Tables 3.3 and 3.4.

If the precomputed points are kept in Chudnovsky Jacobian coordinates for Algorithm 17, Step 10 is performed with $2\mathcal{J} + \mathcal{A} \to \mathcal{J}$ if $k_i = 1$, and $2\mathcal{J} + \mathcal{J}^c \to \mathcal{J}$ otherwise. Similarly, Step 12 is performed with $2\mathcal{J} - \mathcal{A} \to \mathcal{J}$ if $k_i = -1$, and $2\mathcal{J} - \mathcal{J}^c \to \mathcal{J}$ otherwise. Step 14 is performed with $2\mathcal{A} \to \mathcal{J}$ the first time, and $2\mathcal{J} \to \mathcal{J}$ otherwise.

If the precomputed points are converted to affine coordinates for Algorithm 17, $(I + 3(2^{w-2} - 2)M)$ is required to compute $Z_1^{-1}, Z_3^{-1}, \ldots, Z_{2^{w-1}-1}^{-1}$ using simultaneous inversion (Algorithm 54) and $3(2^{w-2} - 1)M + (2^{w-2} - 1)S$ to compute the affine form of each $P_i = (X_i Z_i^{-2}, Y_i Z_i^{-3})$. Once all the precomputed values are in affine coordinates, Step 10 is performed with $2\mathcal{J} + \mathcal{A} \to \mathcal{J}$ and Step 12 is performed with $2\mathcal{J} - \mathcal{A} \to \mathcal{J}$. Step 14 is performed with $2\mathcal{A} \to \mathcal{J}$ the first time and $2\mathcal{J} \to \mathcal{J}$ otherwise. The final step is $\mathcal{J} \to \mathcal{A}$.

When $k_i$ is non-zero, $k_i = \pm 1$ happens with probability around $(1/2^{w-2})$. The precomputation for 17 has average field cost

$$\begin{aligned}&(3M + 5S) + (8M + 3S) + (2^{w-2} - 2)(11M + 3S)\\ &\qquad\qquad\qquad = (11 \cdot 2^{w-2} - 11)M + (3 \cdot 2^{w-2} + 2)S,\end{aligned}$$

converting the table of precomputed values to affine form has cost

$$I + (6 \cdot (2^{w-2} - 1) - 3)M + (2^{w-2} - 1)S.$$

With precomputed values in Chudnovsky Jacobian form, the rest of the algorithm has average field cost

$$\left(\frac{1}{2^{w-2}}\right)\left(\frac{d}{w+1}-1\right)(12M+7S)+ \\ \left(\frac{2^{w-2}-1}{2^{w-2}}\right)\left(\frac{d}{w+1}-1\right)(15M+7S)+(2M+4S)+ \\ \left(d-\frac{d}{w+1}-1\right)(4M+4S)+(I+3M+S)$$

$$=I+\left(\left(\frac{(12+(2^{w-2}-1)15)(d-(w+1))-4d2^{w-2}}{2^{w-2}(w+1)}\right)+4d+1\right)M+ \\ \left(\left(\frac{(7+(2^{w-2}-1)7)(d-(w+1))-4d2^{w-2}}{2^{w-2}(w+1)}\right)+4d+1\right)S.$$

With the precomputed values in affine form, the rest of the algorithm has field cost

$$\left(\frac{d}{w+1}-1\right)(12M+7S)+(2M+4S)+\left(d-\frac{d}{w+1}-1\right)(4M+4S)+(I+3M+S) \\ =I+\left(\frac{8d}{w+1}+4d-11\right)M+\left(\frac{3d}{w+1}+4d-6\right)S.$$

Algorithm 54 requires the same number of temporary field elements as elements being inverted. For space-constrained implementations, this seems to pose a problem. A possible solution is to store the temporary values in the places of the coordinates $Z^2, Z^3$ of the precomputed values, since these are not used.

Tables 3.3 and 3.4 describe the average field costs and average $M$-costs of the window NAF method with various window sizes and $d$ chosen to correspond to the length of the NIST primes $P192$ and $P521$. Both versions of the algorithm are included; using the regular Chudnovsky Jacobian table and converting the table to affine form. By increasing $w$, the running time of the main portion of the algorithm decreases at the cost of the precomputation. This is advantageous until the precomputation, which has exponential cost in $w$, outweighs the gains in the main portion.

Table 3.3: Window NAF Method Average Cost ($d = 192$)

| $w$ | Regular Table | | Affine Table | | Storage |
|---|---|---|---|---|---|
| | Field Cost | $M$-cost | Field Cost | $M$-cost | |
| 3 | **I+1152.0M+914.0S** | **1963.2M** | 2I+1155.0M+915.0S | 2047.0M | 1 |
| 4 | **I+1153.3M+891.2S** | **1946.3M** | 2I+1112.2M+894.2S | 1987.6M | 3 |
| 5 | I+1159.8M+884.0S | 1947.0M | 2I+1129.0M+891.0S | 2001.8M | 7 |
| 6 | I+1210.8M+894.3S | 2006.2M | 2I+1228.4M+909.3S | 2115.9M | 15 |
| 7 | I+1354.7M+932.0S | 2180.3M | 2I+1473.0M+963.0S | 2403.4M | 31 |

Table 3.4: Window NAF Method Average Cost ($d = 521$)

| $w$ | Regular Table | | Affine Table | | Storage |
|---|---|---|---|---|---|
| | Field Cost | $M$-cost | Field Cost | $M$-cost | |
| 3 | **I+3126.0M+2476.8S** | **5187.4M** | 2I+3129.0M+2477.8S | 5271.2M | 1 |
| 4 | I+3094.4M+2404.6S | 5098.1M | **2I+2954.6M+2407.6S** | **5040.7M** | 3 |
| 5 | I+3037.8M+2364.5S | 5009.4M | **2I+2883.7M+2371.5S** | **4940.9M** | 7 |
| 6 | I+3026.2M+2351.3S | 4987.2M | 2I+2920.4M+2366.3S | 4973.5M | 15 |
| 7 | I+3115.4M+2371.4S | 5092.5M | 2I+3118.0M+2402.4S | 5199.9M | 31 |

the gains in the main The fastest version for a given amount of storage space (in terms of Chudnovsky Jacobian points) is written in bold. If none of the terms in a row are bold, then there is a faster set of parameters requiring less storage space in bold earlier in the table. For example, in row 4 of Table 3.4, the $M$-cost using an affine table is $4973.5M$, using 15 points of storage. This is not in bold because in the previous row, an $M$-cost of $4940.9M$ was achieved using only 7 points of storage. The tables for $d$ chosen to correspond with $P224$, $P256$ and $P384$ are similar and can be found in Appendix B.

Notice that the choice of whether or not to convert the table of precomputed values to affine coordinates depends on the $d$ value and the amount of storage space available. It may be noted that the storage list in these tables denotes Chudnovsky

Jacobian points. This choice can also be affected by the computation time of an inversion relative to a multiplication. If $I \approx 80M$ is assumed, the parameters in bold remain the fastest, but when $I$ is as low as $30M$, the affine table becomes slightly more advantageous.

For each NIST prime, given a reasonably small, finite amount of storage space, the window NAF method is faster than the NAF method. The next method we will examine is the sliding window method. The sliding window method has comparable running time to the window NAF.

### 3.2.4 Sliding Window

The window NAF technique uses the width-$w$ NAF of an integer to compute the scalar multiple of a point $P$. The digit set of the base 2 representation is

$$\{-2^{w-1}+1, -2^{w-1}+3, \ldots, -1, 1, \ldots, 2^{w-1}-3, 2^{w-1}-1\},$$

and therefore the points $3P, 5P, \ldots, (2^{w-1}-1)P$ have to be precomputed. The sliding window technique is similar, but uses the NAF of an integer to determine the digits of the representation.

The sliding window takes advantage of the fact that in the NAF representation of an integer, no two consecutive digits are non-zero. This means that if any $w$ consecutive digits are taken from the NAF of an integer $n$, the number it represents will be between $-2^{w+1}/3$ and $2^{w+1}/3$. This follows from the fact that any number that is greater than $2^{w+1}/3$ has a NAF that is at least $w+1$ digits according to Bosma [13].

The digits of the NAF representation of $n$ can be grouped into sets of consecutive digits of length at most $w$, with placement so that the lowest order digit in each set is odd. In the sliding window technique, the integers represented by these sets are used as the digits for a base 2 representation of $n$, and the scalar multiple $nP$ is computed as in the previous algorithms.

The choice of $w$ determines the number of points that have to be precomputed. When the minimum window width is chosen to be $w$, the number of points needed in the precomputation is around a third more than in the window NAF method with parameter $w$ (all odd points up to $2^{w+1}/3$ versus the odd point up to $2^{w+1}/4$). The advantage of this method is that the average density of non-zero digits is lower than in the $w$-NAF representation, causing the number of additions needed to compute the scalar multiple to decrease.

Algorithm 18 is a left-to-right version of the sliding window NAF algorithm adapted from Hankerson *et al.* [43, Alg. 3.38].

**Algorithm 18:** Left-to-Right Sliding Window NAF
**Input:** Affine point $P$, positive integer $k$ with $\mathrm{NAF}(k) = (k_{d-1}, \ldots, k_0)$, window width $w \in \mathbb{N}$
**Output:** Affine point $kP$

```
(1)   *Precomputation*:
(2)       $P_1 \leftarrow P$
(3)       $P_2 \leftarrow 2P$
(4)       **foreach** $i \in \{3, 5, \ldots, 2(2^w - (-1)^w)/3 - 1\}$
(5)           $P_i = P_{i-2} + P_2$
(6)       *Computation*:
(7)       $Q \leftarrow \infty$
(8)       $i \leftarrow d - 1$
(9)       **while** $i \geq 0$
(10)          find the largest $t \leq w$ such that $u \to (k_i, \ldots, k_{i-t+1})$ is
              odd.
(11)          $Q \leftarrow 2^{t-1}Q$
(12)      **if** $u > 0$
```

```
(13)            $Q \leftarrow 2Q + P_u$
(14)        else if $u < 0$
(15)            $Q \leftarrow 2Q - P_{-u}$
(16)        else
(17)            $Q \leftarrow 2Q$
(18)        $i \leftarrow i - t$
(19)    return $Q$
```

According to Hankerson *et al.* [43, Sec 3.3.1], the average length of a run of zeros between windows is

$$\nu(w) = \frac{4}{3} - \frac{(-1)^w}{3 \cdot 2^{w-2}},$$

and the running time for Algorithm 18 is approximately

$$\left(D + \left(\frac{2^w - (-1)^w}{3} - 1\right) A\right) + \left(\left(\frac{d}{w + \nu(w)} - 1\right) A + (d-1)D\right).$$

The steps of this algorithm are analogous to those of Algorithm 17 and the points are represented in the same way. The two options are to keep the precomputed values in Chudnovsky Jacobian coordinates or to use simultaneous inversion to convert them to affine coordinates.

The precomputation for Algorithm 18 has average cost

$$\begin{aligned}(3M + 5S) + (8M + 3S) + &\left(\frac{2^w - (-1)^w}{3} - 2\right)(11M + 3S) \\ &= \left(11 \cdot \frac{2^w - (-1)^w}{3} - 11\right) M + \left(3 \cdot \frac{2^w - (-1)^w}{3} + 2\right) S.\end{aligned}$$

Converting the table of precomputed values to affine form has cost

$$I + \left(6\left(\frac{2^w - (-1)^w}{3} - 1\right) - 3\right) M + \left(\frac{2^w - (-1)^w}{3} - 1\right) S.$$

With the precomputed values in Chudnovsky Jacobian form, the rest of the algorithm has a field cost of

$$\begin{aligned}&\left(\frac{6}{2^w-(-1)^w}\right)\left(\frac{d}{w+\nu(w)}-1\right)(12M+7S)+\\&\qquad\left(1-\frac{6}{2^w-(-1)^w}\right)\left(\frac{d}{w+\nu(w)}-1\right)(15M+7S)+(2M+4S)+\\&\qquad\qquad\left(d-\frac{d}{w+\nu(w)}-1\right)(4M+4S)+(I+3M+S)\end{aligned}$$

$$\begin{aligned}=I+\Bigg(\left(15-3\frac{6}{2^w-(-1)^w}\right)\left(\frac{d}{w+\nu(w)}-1\right)+4d-\frac{4d}{w+\nu(w)}+1\Bigg)S+\\7\left(\frac{d}{w+\nu(w)}-1\right)+4d-\left(\frac{4d}{w+\nu(w)}+1\right)S,\end{aligned}$$

and with the precomputed values in affine form, the rest of the algorithm has cost

$$\begin{aligned}&\left(\frac{d}{w+\nu(w)}-1\right)(12M+7S)+(2M+4S)+\\&\qquad\left(m-\frac{d}{w+\nu(w)}-1\right)(4M+4S)+(I+3M+S).\end{aligned}$$

$$=I+\left(3m+\frac{8d}{w+\nu(w)}+1\right)M+\left(4d+\frac{3d}{w+\nu(w)}+1\right)S.$$

Tables 3.5 and 3.6 describe the average field costs and average $M$-costs of the sliding window method with various window sizes and $d$ chosen to correspond to the length of the NIST primes $P192$ and $P521$, respectively. Both versions of the algorithm are included; using the regular Chudnovsky Jacobian table and converting the table to affine form. The fastest version for a given amount of storage space is written in bold. If none of the terms are bold, then there is a version requiring less storage space that is faster listed earlier in the table. The tables for $d$ chosen to correspond with $P224$, $P256$ and $P384$ are similar and can be found in Appendix B.

As with the $w$-NAF, the choice of whether or not to convert the table of precomputed values to affine coordinates depends on the $d$ value and the amount of

Table 3.5: Sliding Window Method Average Cost ($d = 192$)

| $w$ | Regular Table | | Affine Table | | Storage |
|---|---|---|---|---|---|
| | Field Cost | $M$-cost | Field Cost | $M$-cost | |
| 3 | **I+1162.0M+901.0S** | **1962.8M** | 2I+1129.3M+903.0S | 2011.7M | 2 |
| 4 | **I+1157.6M+888.7S** | **1948.6M** | 2I+1114.6M+892.7S | 1988.7M | 4 |
| 5 | I+1179.4M+887.4S | 1969.3M | 2I+1164.9M+897.4S | 2042.8M | 10 |
| 6 | I+1255.6M+905.8S | 2060.2M | 2I+1304.1M+925.8S | 2204.7M | 20 |
| 7 | I+1466.1M+962.0S | 2315.7M | 2I+1652.1M+1004.0S | 2615.3M | 42 |

Table 3.6: Sliding Window Method Average Cost ($d = 521$)

| $w$ | Regular Table | | Affine Table | | Storage |
|---|---|---|---|---|---|
| | Field Cost | $M$-cost | Field Cost | $M$-cost | |
| 3 | I+3136.0M+2436.3S | 5165.1M | **2I+3030.2M+2438.3S** | **5140.9M** | 2 |
| 4 | I+3087.7M+2392.7S | 5081.9M | **2I+2931.9M+2396.7S** | **5009.3M** | 4 |
| 5 | I+3034.9M+2358.2S | 5001.5M | **2I+2893.8M+2368.2S** | **4948.3M** | 10 |
| 6 | I+3053.7M+2356.7S | 5019.1M | 2I+2980.0M+2376.7S | 5041.4M | 20 |
| 7 | I+3210.3M+2396.3S | 5207.3M | 2I+3283.5M+2438.3S | 5394.2M | 42 |

storage space available. This choice can also be affected by the computation time of an inversion relative to a multiplication; for example, when $I$ is as low as $30M$, the affine table becomes slightly more advantageous.

The $M$-costs of the sliding window technique are comparable to the window NAF method. The next method we will examine is the fractional window method. The fractional window technique is a generalization of both the window NAF and the sliding window techniques and will tie the results of both algorithms together.

### 3.2.5 Fractional Window

For a given value $w$, the window NAF and sliding window techniques require a fixed number of precomputed points. For each successive value of $w$, the number of values that need to be precomputed doubles. The fractional window technique was introduced by Möller [72], and is a generalization of the window NAF and sliding window techniques. This representation allows more flexibility in the number of precomputed values while providing a computation time that is at least as fast.

Let $w \geq 2$ be an integer and $m$ an odd integer such that $1 \leq m \leq 2^w - 1$. The $(m, w)$-*ary fractional window representation* is a base 2 representation with digits in the set

$$B = \{\pm 1, \pm 3, \ldots, \pm(2^w + m)\}.$$

Let the map $\eta : \{0, 1, \ldots, 2^{w+2}\} \to B \cup \{0\}$ be defined as follows:

- If $x$ is even, then $\eta(x) = 0$.
- If $0 < x \leq 2^w + m$, then $\eta(x) = x$.
- If $2^w + m < x < 3 \cdot 2^w - m$, then $\eta(x) = x - 2^{w+1}$.
- If $3 \cdot 2^w - m \leq x < 2^{w+2}$, then $\eta(x) = x - 2^{w+2}$.

Algorithm 19, from Möller [71], encodes the integer $k$ into signed fractional window representation. Starting with the lowest $w+2$ digits, the algorithm identifies the appropriate digit by applying $\eta$ and subtracts this from the number. The set of digits that is examined is shifted by one and the process is repeated. This is not unlike the encoding process for window NAF (Algorithm 16) except that in this case the encoding function guarantees that the digit is in the range $\{-2^w - m, \ldots, 2^w + m\}$.

**Algorithm 19:** Encoding Signed Fractional Window
**Input:** Integer $k$, width $w \in \mathbb{N}$, parameter $m \in \{1, 3, \ldots, 2^w - 1\}$
**Output:** Signed $(m, w)$-ary fractional representation $(b_{i-1}, \ldots, b_0)$ of $k$

```
(1)   $d \leftarrow k \bmod 2^{w+2}$
(2)   $c \leftarrow \lfloor k/2^{w+2} \rfloor$
(3)   $i \leftarrow 0$
(4)   **while** $d \neq 0$ and $c \neq 0$
(5)       $b_i \leftarrow \eta(d)$
(6)       $d \leftarrow d - b_1$
(7)       $i \leftarrow i + 1$
(8)       $d \leftarrow (c \bmod 2) \cdot 2^{w+1} + d/2$
(9)       $c \leftarrow \lfloor c/2 \rfloor$
(10)  **return** $(b_{i-1}, \ldots, b_0)$
```

Just like with the $w$-NAF encoding, the length of the fractional window encoding of an integer is at most one digit longer than the length of the binary representation. Algorithm 20, from Möller [71], is the algorithm for computing the scalar multiple $kP$ of a point $P$ when $k \in \mathbb{N}$ is given in signed fractional window representation. It is nearly identical to that of left-to-right window NAF except for the different representation of $k$ and the values that have to be precomputed. The precomputed values are $\{3P, 5P, \ldots, (2^w + m)P\}$.

**Algorithm 20:** Left-to-Right Fractional Window NAF
**Input:** Affine point $P$, positive integer $k$ with signed $m, w$-ary fractional representation $(k_{d-1}, \ldots, k_0)$
**Output:** Affine point $kP$

```
(1)   *Precomputation*:
(2)   $P_1 \leftarrow P$
(3)   $P_2 \leftarrow 2P$
(4)   **foreach** $i \in \{3, 5, \ldots, 2^w + m\}$
(5)       compute $P_i \leftarrow P_{i-2} + P_2$
(6)   *Computation*:
(7)   $Q \leftarrow P$
(8)   **for** $i = d - 2$ **to** $0$
(9)       **if** $k_i > 0$
(10)          $Q \leftarrow 2Q + P_{k_i}$
(11)      **else if** $k_i < 0$
(12)          $Q \leftarrow 2Q - P_{k_i}$
(13)      **else**
```

```
(14)        $Q \leftarrow 2Q$
(15)    **return** $Q$
```

According to Möller [71], the average density of non-zero entries achieved by the signed fractional window representation with parameters $w$ and $m$ is

$$\left(\frac{1}{w+\frac{m+1}{2^w}+2}\right),$$

so the running time for Algorithm 18 is approximately

$$\left(D+\left(\frac{2^w+m+1}{2}-1\right)A\right)+\left(\left(\frac{1}{w+\frac{m+1}{2^w}}-1\right)A+(d-1)D\right).$$

The steps of this algorithm are analogous to those of Algorithms 17 and 18, and the points are represented in the same way. The two options are to keep the precomputed values in Chudnovsky Jacobian coordinates or to use simultaneous inversion to convert them to affine coordinates.

Algorithm 20 has field cost

$$(3M+5S)+(8M+3S)+11\left(\frac{2^w+m+1}{2}-11\right)M+3\left(\frac{2^w+m+1}{2}+2\right)S$$

in the precomputation stage, and converting the table of precomputed values to affine form has field cost

$$I+\left(6\left(\frac{2^w+m+1}{2}-1\right)-3\right)M+\left(\frac{2^w+m+1}{2}-1\right)S.$$

With the precomputed values in Chudnovsky Jacobian form, the rest of the algorithm has field cost

$$\begin{aligned}\left(\frac{1}{2^w+m+1}\right)\left(\frac{d}{w+\frac{m+1}{2^w}+2}-1\right)(12M+7S)+\\ \left(1-\frac{1}{2^w+m+1}\right)\left(\frac{d}{w+\frac{m+1}{2^w}+2}-1\right)(15M+7S)+(2M+4S)\\ +\left(d-\frac{d}{w+\frac{m+1}{2^w}+2}-1\right)(4M+4S)+(I+3M+S)\end{aligned}$$

$$= I + \left(\left(15 - 3\frac{1}{2^w + m + 1}\right)\left(\frac{d}{w + \frac{m+1}{2^w} + 2} - 1\right) + 4d - \frac{4d}{w + \frac{m+1}{2^w} + 2} + 1\right) S +$$
$$7\left(\frac{d}{w + \frac{m+1}{2^w} + 2} - 1\right) + 4d - \left(\frac{4d}{w + \frac{m+1}{2^w} + 2} + 1\right) S$$

With the precomputed values in affine form, the rest of the algorithm has field cost

$$\left(\frac{d}{w + \frac{m+1}{2^w} + 2} - 1\right)(12M + 7S) + (2M + 4S)$$
$$+ \left(d - \frac{d}{w + \frac{m+1}{2^w} + 2} - 1\right)(4M + 4S) + (I + 3M + S)$$
$$= I + \left(3m + \frac{8m}{w + \frac{m+1}{2^w} + 2} + 1\right) M + \left(4m + \frac{3m}{w + \frac{m+1}{2^w} + 2} + 1\right) S.$$

Tables 3.7 and 3.8 describe the average field costs and average $M$-costs of the fractional window method with different values of $w$ and $m$ with $d$ chosen to correspond to the length of the NIST primes $P192$ and $P521$, respectively. Both versions of the algorithm are included; using the Chudnovsky Jacobian table and using the table converted to affine coordinates. The fastest version for a given amount of storage space is written in bold. If none of the terms are bold, then there is a faster version requiring less storage. The tables for $d$ chosen to correspond to $P224$, $P256$ and $P384$ are similar and can be found in Appendix B.

It must be noted that for a given amount of storage space, the average cost of the fractional window algorithm agrees exactly with that of the window NAF (Section 3.2.3) and sliding window (Section 3.2.4) algorithms. This demonstrates how the fractional window technique is a generalization of both previous methods.

As we will show in Section 3.2.7, if storage space is disregarded, the fractional window method is the fastest algorithm for unknown point scalar multiplication.

Table 3.7: Fractional Window Method Average Cost ($d = 192$)

| $w$ | $m$ | Regular Table | | Affine Table | | Sto. |
|---|---|---|---|---|---|---|
| | | Field Cost | $M$-cost | Field Cost | $M$-cost | |
| 1 | 1 | **I+1152.0M+914.0S** | **1963.2M** | 2I+1155.0M+915.0S | 2047.0M | 1 |
| 2 | 1 | **I+1162.0M+901.0S** | **1962.8M** | 2I+1129.3M+903.0S | 2011.7M | 2 |
| | 3 | **I+1153.3M+891.2S** | **1946.3M** | 2I+1112.2M+894.2S | 1987.6M | 3 |
| 3 | 1 | I+1157.6M+888.7S | 1948.6M | 2I+1114.6M+892.7S | 1988.7M | 4 |
| | 3 | I+1159.1M+886.7S | 1948.5M | 2I+1118.3M+891.7S | 1991.7M | 5 |
| | 5 | I+1159.5M+885.2S | 1947.7M | 2I+1123.1M+891.2S | 1996.1M | 6 |
| | 7 | I+1159.8M+884.0S | 1947.0M | 2I+1129.0M+891.0S | 2001.8M | 7 |
| 4 | 1 | I+1166.6M+885.0S | 1954.6M | 2I+1140.8M+893.0S | 2015.2M | 8 |
| | 3 | I+1173.1M+886.2S | 1962.0M | 2I+1152.8M+895.2S | 2028.9M | 9 |
| | 5 | I+1179.4M+887.4S | 1969.3M | 2I+1164.9M+897.4S | 2042.8M | 10 |
| | 7 | I+1185.7M+888.6S | 1976.5M | 2I+1177.3M+899.6S | 2057.0M | 11 |

Table 3.8: Fractional Window Method Average Cost ($d = 521$)

| $w$ | $m$ | Regular Table | | Affine Table | | Sto. |
|---|---|---|---|---|---|---|
| | | Field Cost | $M$-cost | Field Cost | $M$-cost | |
| 1 | 1 | **I+3126.0M+2476.8S** | **5187.4M** | 2I+3129.0M+2477.8S | 5271.2M | 1 |
| 2 | 1 | I+3136.0M+2436.3S | 5165.1M | **2I+3030.2M+2438.3S** | **5140.9M** | 2 |
| | 3 | I+3094.4M+2404.6S | 5098.1M | **2I+2954.6M+2407.6S** | **5040.7M** | 3 |
| 3 | 1 | I+3087.7M+2392.7S | 5081.9M | **2I+2931.9M+2396.7S** | **5009.3M** | 4 |
| | 3 | I+3073.3M+2382.2S | 5059.0M | **2I+2912.8M+2387.2S** | **4982.6M** | 5 |
| | 5 | I+3055.9M+2372.8S | 5034.1M | **2I+2896.9M+2378.8S** | **4959.9M** | 6 |
| | 7 | I+3037.8M+2364.5S | 5009.4M | **2I+2883.7M+2371.5S** | **4940.9M** | 7 |
| 4 | 1 | I+3037.6M+2362.2S | 5007.4M | 2I+2886.5M+2370.2S | 4942.6M | 8 |
| | 3 | I+3036.5M+2360.1S | 5004.6M | 2I+2889.9M+2369.1S | 4945.1M | 9 |
| | 5 | I+3034.9M+2358.2S | 5001.5M | 2I+2893.8M+2368.2S | 4948.3M | 10 |
| | 7 | I+3033.1M+2356.5S | 4998.3M | 2I+2898.2M+2367.5S | 4952.2M | 11 |

When comparing algorithms with a fixed amount of storage space, the fractional window algorithm can possibly be beaten by the algorithm described in Section 3.2.6, double-base chain scalar multiplication.

### 3.2.6 Double-Base Representation

In recent years, research has been done on a highly redundant system for representing integers, the *double-base number system (DBNS)*. Rather than representing an integer by a sum of powers of two, it is represented by a sum of mixed powers of two and three (see Dimitrov *et al.* [30]). An integer $k$ is represented as

$$k = \sum_{i=1}^{m} s_i 2^{b_i} 3^{t_i} \quad \text{with } s_i \in \{-1, 1\} \text{ and } b_i, t_i \geq 0.$$

The DBNS can result in very sparse representations of integers, as evidenced by the following theorem of Dimitrov *et al.* [29].

**Theorem 3.2.3.** *Every positive integer $k$ can be represented as the sum of at most $O\left(\frac{\log k}{\log \log k}\right)$ numbers of the form $2^b 3^t$.*

The proof is based on a result by Tijdeman [93] that states that given the ordered sequence $\{n_1, n_2, \ldots\}$ of integers that are powers of the same two primes, the maximal distance between $n_i$ and $n_{i+1}$ is sub-linear in $n_i$. Of all the representations discussed in this thesis, DBNS is the only one for which the optimal number of terms is provably sublinear in $\log(k)$.

A *canonical* representation for an integer $k$ is a representation that has a minimal number of terms. It seems that finding a canonical DBNS representation of a large integer is a hard problem. There is, however, a greedy algorithm (see Dimitrov *et* *al.* [28]) that can find a relatively sparse representation that satisfies the asymptotic bound from Theorem 3.2.3.

A more useful class of DBNS representations in our context is the class of *double-base chains.*

**Definition 3.2.4.** *Given $k \in \mathbb{N}$, a* double-base chain *for $k$ is a sequence of triples $((s_m, b_m, t_m), \ldots, (s_1, b_1, t_1))$ where $s_i \in \{1, -1\}$ for all $i$, $0 \leq b_1 \leq b_2 \leq \cdots \leq b_m$ and $0 \leq t_1 \leq t_2 \leq \cdots \leq t_m$ such that*

$$k = \sum_{i=1}^{m} s_i 2^{b_i} 3^{t_i}.$$

*The* length *of a double-base-chain is $m$. Each $s_i 2^{b_i} 3^{t_i}$ is called a* term *of the chain.*

A double-base chain can be uniquely identified by the differences $b_i - b_{i-1}$ and $t_i - t_{i-1}$ for $2 \leq i \leq m$, resulting in a compact representation of $k$. Dimitrov [28, Alg. 1] describes a variation of the greedy algorithm that returns a double-base chain for an integer with a small number of terms. This algorithm is included here as Algorithm 21.

When given the parameters $b_{max}$ and $t_{max}$, Algorithm 21 returns a double-base chain for $k$ for which the largest term is $2^b 3^t$ with $b \leq b_{max}, t \leq t_{max}$. The algorithm begins by finding the closest power of two and three $2^b 3^t$ to $k$ with $b \leq b_{max}, t \leq t_{max}$. This term is taken as the first term of the DBNS representation. The number $k$ is updated by subtracting the term from $k$. The values $b_{max}$ and $t_{max}$ are updated to $b$ and $t$ respectively. The process is repeated until $k$ is exactly a power of two and three.

**Algorithm 21:** Conversion to DBNS with Restricted Exponents

**Input:** Positive integer $k$, positive integers $b_{max}, t_{max}$, the largest allowed binary and ternary exponents
**Output:** A sequence of triples $((s_m, b_m, t_m), \ldots, (s_1, b_1, t_1))$ such that $k = \sum_{i=1}^{m} s_i 2^{b_i} 3^{t_i}$, where $b_1 \leq b_2 \leq \cdots \leq b_m$, $t_1 \leq t_2 \leq \cdots \leq t_m$ and $s_i \in \{-1, 1\}$

```
(1)   $s \leftarrow 0$
(2)   $i \leftarrow 1$
(3)   **while** $k > 0$
(4)       define $z = 2^b 3^t$, to be the best approximation of $k$ with $0 \leq$
          $b \leq b_{max}$ and $0 \leq t \leq t_{max}$
(5)       $(s_i, b_i, t_i) \leftarrow (s, b, t)$
(6)       $i \leftarrow i + 1$
(7)       $b_{max} \leftarrow b, \quad t_{max} \leftarrow t$
(8)       **if** $k < z$ **then** $s \leftarrow -s$
(9)       $k \leftarrow |k - z|$
(10)  **return** $(s_i, b_i, t_i)_{i>0}$
```

The complexity of Algorithm 21 depends greatly on the implementation of Step 4. If it is feasible to store all the mixed powers of 2 and 3, then this can be done with a simple search. Less memory-intensive algorithms have been developed by Berthé and Imbert [12]. In any case, the complexity of Algorithm 21 can be made insignificant in comparison with the time it takes for scalar multiplication. In situations where a random integer multiplier needs to be generated, it may be possible to securely generate a random double-base chain, avoiding the need for this algorithm.

Table 3.9 presents the results of the greedy algorithm on a set of random integers of lengths $d = 192, 224, 256, 384$ and 521 with various choices for $b_{max}$ and $t_{max}$. The values of $b_{max}, t_{max}$ are chosen so that $2^{b_{max}} 3^{t_{max}}$ closely approximates the corresponding NIST prime. For Table 3.9, all $2^b 3^t$ were enumerated and a few of the closest values to $2^{192}, 2^{224}, 2^{256}, 2^{384}$ and $2^{512}$ were chosen. This choice ensures that the first term of the double-base chain is most likely $(1, b_{max}, t_{max})$ and high enough so that the remainder after subtracting $(b_{max}, t_{max})$ is not too high. The

Table 3.9: Results of Greedy Algorithm on 2500 Integers

| $d$ | $b_{max}$ | $t_{max}$ | terms |
|---|---|---|---|
| 192 | 135 | 36 | 48.73 |
| | 116 | 48 | 44.07 |
| | 89 | 65 | 44.98 |
| | 51 | 89 | 59.64 |
| 224 | 140 | 53 | 52.58 |
| | 121 | 65 | 49.75 |
| | 102 | 77 | 53.04 |
| 256 | 199 | 36 | 70.05 |
| | 172 | 53 | 63.00 |
| | 134 | 77 | 56.49 |
| | 88 | 106 | 71.74 |
| 384 | 300 | 53 | 105.84 |
| | 216 | 106 | 84.60 |
| | 197 | 118 | 84.57 |
| | 132 | 159 | 107.82 |
| 521 | 437 | 53 | 151.29 |
| | 353 | 106 | 128.91 |
| | 269 | 159 | 113.43 |
| | 183 | 212 | 143.84 |

point of Table 3.9 is to find optimal values of $b_{max}, t_{max}$ for Algorithm 22, the scalar multiplication algorithm presented below. The results of Table 3.9 were computed by the author with an implementation of Algorithm 21 with a precomputed table of powers of two and three. This algorithm was applied to 2500 random integers of with bit-size $d$ for $d$ corresponding to the NIST primes. The mean number of terms for each set of parameter was computed and included as column 4 of table 3.9.

Algorithm 22, adapted from Dimitrov *et al.* [28], computes the scalar multiple of a point when given the multiplier as a double-base chain. This is computed in an analogous way to left-to-right binary multiplication. In the binary algorithm, a temporary variable stores the initial point and is doubled a number of times corresponding to the difference in the powers of the non-zero terms. In Algorithm 22, the temporary variable is doubled and tripled for the difference between successive terms. The sign of each term determines whether the base point is added or subtracted.

```
Algorithm 22: Double-Base Chain Scalar Multiplication
Input: Affine point $P$, positive integer $k$ with DBNS representation
$k = \sum_{i=1}^{m} s_i 2^{b_i} 3^{t_i}$, where $b_1 \leq b_2 \leq \cdots \leq b_m$ and $t_1 \leq t_2 \leq \cdots \leq t_m$,
$s_i \in \{\pm 1\}$
Output: Affine point $kP$
(1)   $Q \leftarrow s_1 P$
(2)   for $i = 1$ to $m - 1$
(3)      $u \leftarrow b_i - b_{i+1}, \quad v \leftarrow t_i - t_{i+1}$
(4)      $Q \leftarrow 2^u 3^v Q$
(5)      if $s_{i+1} = 1$
(6)         $Q \leftarrow Q + P$
(7)         if $s_{i+1} = -1$
(8)            $Q \leftarrow Q - P$
(9)      return $Q$
```

For an integer with $m$ terms, maximal binary exponent $b_m$ and maximal ternary exponent $t_m$, Algorithm 22 will have estimated computation time

$$(m-1)A + b_m D + t_m T.$$

Algorithm 22 requires a large number of doublings and triplings, and a smaller number of additions. The fastest coordinate system for doubling and tripling is Jacobian. We will therefore use the formulas for $\mathcal{J} + \mathcal{A} \rightarrow \mathcal{J}$, $2\mathcal{J} \rightarrow \mathcal{J}$, and $3\mathcal{J} \rightarrow \mathcal{J}$ to execute Algorithm 22. The field cost of Algorithm 22 is exactly

$$\begin{aligned}(m-1)(8M+3S)+(2M+4S)+(b_m-1)(4M+4S)+t_m(9M+5S)+(I+3M+S)\\ = I + (8M + 4b_m + 9t_m - 7)M + (3m + 4b_m + 5t_m - 2)S. \quad (3.1)\end{aligned}$$

Table 3.10: Double-Base Chain Average Cost

| $d$ | $b_m$ | $t_m$ | $m$ | Field Cost | $M$-cost |
|---|---|---|---|---|---|
| 192 | 135 | 36 | 48.73 | I+1246.8M+864.2S | 2018.2M |
| | **116** | **48** | 44.07 | I+1241.6M+834.2S | **1988.9M** |
| | 89 | 65 | 44.98 | I+1293.8M+813.9S | 2025.0M |
| | 51 | 89 | 59.64 | I+1475.1M+825.9S | 2215.9M |
| 224 | 140 | 53 | 52.58 | I+1450.6M+980.7S | 2315.2M |
| | **121** | **65** | 49.75 | I+1460.0M+956.2S | **2305.0M** |
| | 102 | 77 | 53.04 | I+1518.3M+950.1S | 2358.4M |
| 256 | 199 | 36 | 70.05 | I+1673.4M+1184.2S | 2700.7M |
| | 172 | 53 | 63.00 | I+1662.0M+1140.0S | 2654.0M |
| | **134** | **77** | 56.49 | I+1673.9M+1088.5S | **2624.7M** |
| | 88 | 106 | 71.74 | I+1872.9M+1095.2S | 2829.1M |
| 384 | 300 | 53 | 105.84 | I+2516.7M+1780.5S | 4021.1M |
| | **216** | **106** | 84.60 | I+2487.8M+1645.8S | **3884.4M** |
| | 197 | 118 | 84.57 | I+2519.6M+1629.7S | 3903.3M |
| | 132 | 159 | 107.82 | I+2814.6M+1644.5S | 4210.1M |
| 521 | 437 | 53 | 151.29 | I+3428.3M+2464.9S | 5480.2M |
| | 353 | 106 | 128.91 | I+3390.3M+2326.7S | 5331.7M |
| | **269** | **158** | 113.43 | I+3398.4M+2204.3S | **5241.9M** |
| | 183 | 212 | 143.84 | I+3783.7M+2221.5S | 5640.9M |

From Table 3.9, we obtain estimates for the values of $m$ given certain $b_m, t_m$ for the NIST primes. By evaluating Equation (3.1) with these values and comparing the $M$-costs, we can determine the average running time for the selected values of $(b_m, t_m)$. These values are presented in Table 3.10.

The optimal values for $(b_m, t_m)$ in terms of $M$-cost are listed in bold in Table 3.10. For multipliers of length $d = 192$ the optimal value of $(b_m, t_m)$ is $(116, 48)$, for $d = 224$ it is $(121, 65)$, for $d = 256$ it is $(134, 77)$, for $d = 384$ it is $(216, 106)$ and for $d = 521$ it is $(269, 159)$. Table 3.10 demonstrates that Algorithm 22 for scalar multiplication is slightly slower than the fastest methods based on the binary

Table 3.11: Scalar Multiplication ($P192$)

| Algorithm | Variables | Storage | Field Cost | $M$-cost |
|---|---|---|---|---|
| R2L Binary (12) | – | – | I+1909.0M+1146.5S | 2906.2M |
| L2R Binary (13) | – | – | I+1516.0M+1043.0S | 2430.4M |
| L2R NAF (15) | – | – | I+1269.0M+954.0S | 2112.2M |
| $w$-NAF (17) | $w = 4$, $\mathcal{J}^c$ | 3 | I+1153.3M+891.2S | 1946.3M |
| swNAF (18) | $w = 4$, $\mathcal{J}^c$ | 4 | I+1157.6M+888.7S | 1948.6M |
| fwNAF (20) | $w = 2$, $c = 3$, $\mathcal{J}^c$ | 3 | I+1153.3M+891.2S | 1946.3M |
| DBChain (22) | $b_{max} = 116, t_{max} = 48$ | – | I+1242.2M+834.5S | 1988.9M |

algorithm.

### 3.2.7 Summary

Table 3.11 provides a summary of the algorithms for scalar multiplication of unknown points in terms of their storage requirements (in points), their field costs and their $M$-costs. For each algorithm, the parameters that result in the smallest $M$-costs are chosen. The parameters are chosen to correspond to the curve over the NIST prime $P192$. The results for elliptic curves over the NIST primes $P224$, $P256$, $P384$ and $P521$ are presented in Appendix B and mirror the results of Table 3.11.

The "Variables" column lists the specific parameters that are used to give the results in the row. Storage is measured in Chudnovsky Jacobian points. For the window methods, $\mathcal{A}$ means that the precomputed table of values is converted to affine form (only appears in Appendix tables) and $\mathcal{J}^c$ means that it is kept in Chudnovsky Jacobian form.

The storage column shows the number of points stored. The only storage required for the NAF and double-base chain methods are the representations of the multiplier.

Table 3.12: Low Storage Scalar Multiplication Comparison

| $d$ | Algorithm | Storage | $M$-cost |
|---|---|---|---|
| 192 | fwNAF | 1 | 1963.2M |
| | | 2 | 1962.8M |
| | DBChain | 0 | 1988.9M |
| 224 | fwNAF | 1 | 2276.8M |
| | | 2 | 2274.3M |
| | DBChain | 0 | 2305.0M |
| 256 | fwNAF | 1 | 2590.4M |
| | | 2 | 2585.7M |
| | DBChain | 0 | 2624.7M |
| 384 | fwNAF | 1 | 3844.8M |
| | | 2 | 3831.6M |
| | DBChain | 0 | 3884.4M |
| 512 | fwNAF | 1 | 5187.4M |
| | | 2 | 5140.9M |
| | DBChain | 0 | 5241.9M |

The storage space for each of these is usually less than that of two field elements and can be ignored.

Table 3.11 demonstrates that under the assumptions made about the relative costs of field operations, the fractional window method (Algorithm 20) is the fastest algorithm for computing scalar multiples of unknown points on elliptic curves over $P192$. The fastest algorithm requiring no storage is the double-base chain method (Algorithm 22). These results are similar for the NIST primes $P224$, $P256$, $P384$ and $P521$ in Tables B.10 to B.13. The results from Tables 3.7, B.7, B.8, B.9 and 3.8 for the fractional window algorithm and Table 3.10 for the double-base chain algorithm are combined in Table 3.12 to demonstrate the $M$-cost of both algorithms when storage space is limited. Table 3.12 shows that for a given non-zero amount of storage space, the fractional window algorithm has a lower $M$-cost than the double-base chain algorithm.

## 3.3 Known Point Scalar Multiplication

In this section, we study methods for scalar multiplication that can be used when the point is known in advance. The main theoretical difference in this case compared to the unknown point case is that precomputations are not included in the running time of the algorithms. This allows for more expensive precomputation steps that can be used to speed up computation considerably.

The methods we focus on in this section are the windowing method and the comb method. Windowing works by precomputing a number of the binary power multiples of the fixed point to eliminate the need for doubling. In the comb method, expensive sums of certain multiples of $P$ are stored in order to reduce the number of operations needed.

### 3.3.1 Fixed-Base Windowing

The naïve method for computing a scalar multiple of a point using precomputation would be to compute $2^iP$ for as many $i$'s as are needed and to apply the right-to-left binary algorithm from Section 3.2.1 without any doubling. For a $d$-bit multiplier, this would cost approximately $\frac{d-1}{2}A$ and require the storage of $d-1$ extra points. Brickell *et al.* [14] proposed a version of this algorithm that uses the same idea but allows for more flexibility in the number of points stored. This idea was originally proposed by Yao [98].

Suppose $k \in \mathbb{N}$ is written as

$$k = \sum_{i=0}^{l-1} b_i 2^{wi},$$

for $b_i$ in some digit set $B$. Let $B'$ denote the set of absolute values of non-zero digits in $B$. We can write

$$k = \sum_{b \in B'} b \left( \sum_{i:b_i=b} 2^{wi} - \sum_{i:b_i=-b} 2^{wi} \right).$$

Define the points

$$P_b = \left( \sum_{i:b_i=b} 2^{wi} - \sum_{i:b_i=-b} 2^{wi} \right) P.$$

We can then calculate $kP = \sum_{b \in B'} bP_b$.

The digits $b_i$ can be computed by taking the NAF of the point $k$ and partitioning the digits into blocks of length $w$. By writing $\mathrm{NAF}(k) = K_{l-1}||K_{l-2}|| \cdots ||K_1||K_0$ and taking the integers represented by the blocks $K_i$ as the digits $b_i$ we obtain

$$b_i \in B = \{-K, -K+1, \ldots, -1, 0, 1, \ldots, K-1, K\},$$

for $0 \leq i \leq l-1$ and $K = (2^{w+1}-2)/3$ if $w$ is even and $K = (2^{w+1}-1)/3$ otherwise.

Algorithm 23 computes the scalar multiple of a point using fixed-base windowing. It proceeds in two steps, the precomputation stage and the result stage. In the precomputation stage, we compute $2^{iw}P$ for $1 \leq i \leq l-1$ by repeated doubling. In the result stage, the final sum $\sum_{b \in B} bP_b$ is computed from the precomputed values. Algorithm 23 is adapted from Hankerson *et al.* [43, Alg. 3.42] and Brickell *et al.* [14].

**Algorithm 23:** Fixed-Base NAF Windowing
**Input:** Affine point $P$, positive integer $k$ with NAF($k$) written as $l$ singed bit-strings of length $w$: $K_{l-1}||K_{l-2}|| \cdots ||K_1||K_0$

```
Output: Affine point $kP$
(1)   Offline Precomputation:
(2)      foreach $0 \leq i \leq l-1$
(3)         $P_i = 2^{wi}P$
(4)   $A \leftarrow \infty, B \leftarrow \infty$
(5)   foreach $j$ from $K$ down to 1
(6)      foreach $i$ for which $K_i = j$
(7)         $B \leftarrow B + P_j$
(8)      foreach $i$ for which $K_i = -j$
(9)         $B \leftarrow B - P_j$
(10)     $A \leftarrow A + B$
(11)  return $A$
```

The the average running time of Algorithm 23 is

$$\left(K - \frac{2K+1}{4l} + \left(l\frac{2K}{2K+1}\right)\right)A.$$

The $l\left(\frac{2K}{2K+1}\right)$ additions come from Steps 7 and 9 when $P_i$ is non-zero. One of these steps is executed once for each of the $l$ blocks $K_i$, each of which has a $1/(2K+1)$ chance of being 0 and contributing nothing.

The other $K - \frac{2K+1}{4l}$ additions come from Step 10. The total number of times that this step is applied is $K$, but if the digits $\{K, K-1, \ldots, K-T+1\}$ do not occur as some $|K_i|$, then the $B$ value is $\infty$ and no addition is performed for the first $T$ applications of this step. Since the digits $K_i$ are essentially chosen at random among $-K, \ldots, K$, the typical gap between two represented numbers should be $(2K+1)/L$. Therefore, the average highest value will be $K-(2K+1)/2L$ and the average lowest value will be $-K+(2K+1)/2L$. This leads to an average highest absolute value of $K-(2K+1)/4L$, from which we get $T = (2K+1)/4L$. In this case, the variable $B$ will be $\infty$ for Step 10 the first $T$ times. Therefore the number of additions contributed by this step is $K - \frac{2K+1}{4l}$. The algorithm also requires the storage of $l-1$ temporary points. The cases where either $P_i$ or $B$ are $\infty$ are not examined in Hankerson *et al.*

Table 3.13: Fixed-Base Window Average Cost ($d = 192$)

| $w$ | Storage | Field Cost | $M$-cost |
|---|---|---|---|
| 2 | 95 | I+639.3M+237.4S | 909.1M |
| 3 | 63 | I+523.0M+190.4S | 755.3M |
| **4** | **47** | **I+477.5M+167.8S** | **691.8M** |
| 5 | 38 | I+535.7M+177.5S | 757.7M |
| 6 | 31 | I+710.7M+219.9S | 966.6M |
| 7 | 27 | I+1143.9M+334.9S | 1491.8M |
| 8 | 23 | I+2025.4M+572.1S | 2563.1M |

[43].

Since precomputation is not an issue, we will assume that the precomputed points are stored in affine coordinates to allow faster addition. Since the only operation performed is addition, the intermediate points are stored in Chudnovsky Jacobian coordinates. The additions of $P_i$ will be $\mathcal{J}^c + \mathcal{A} \rightarrow \mathcal{J}^c$ and the other additions $\mathcal{J}^c + \mathcal{J}^c \rightarrow \mathcal{J}^c$. The average field cost of this algorithm is therefore

$$\begin{aligned}\left(l\left(\frac{2K}{2K+1}\right)\right)(8M+3S) + \left(K - \frac{2K+1}{4l}\right)(11M+3S) + (I+3M+S)\\ = I + \left((8l)\left(\frac{2K}{2K+1}\right) - 11\frac{2K+1}{4l} + 11K + 3\right)M\\ + \left((3l)\left(\frac{2K}{2K+1}\right) - 3\frac{2K+1}{4l} + 3K + 1\right)S.\end{aligned}$$

Table 3.13 lists the cost of fixed-base windowing for values of $d$ that correspond to the length of the NIST prime $P192$. Here, $l = \lceil \frac{d}{w} \rceil$ where $d$ is the size of the input and $w$ is the fixed windowing size. The row in bold represents the parameters that produce the lowest $M$-cost. The tables for $d$ chosen to correspond to $P224$, $P256$, $P384$ and $P521$ are similar and can be found in Appendix B.

These results demonstrate that with enough precomputation, scalar multiplication can be performed much more quickly than with unknown point methods. The cost of the fixed-base window method is high for very small and very large amounts of precomputation. In the next section, we examine another algorithm that is faster than fixed-base windowing in some situations.

### 3.3.2 Fixed-Base Comb

In the fixed-base windowing method, the values $2^{wi}P$ are computed and combined with addition to form $kP$. In the fixed-base comb method, rather than eliminating all the doublings, the goal is to use precomputation to utilize the doublings more efficiently. This idea was originally proposed by Lim and Lee [64].

Suppose that $k \in \mathbb{N}$ is represented by $l$ bit-strings of length $w$, i.e.

$$k = K_{l-1}||K_{l-2}|| \cdots ||K_1||K_0,$$

where $K_i = K_{i,w-1}||K_{i,w-2}|| \cdots ||K_{i,1}$ and $K_{i,j} \in \{0,1\}$. We can write $k$ as follows:

$$\begin{aligned} k &= \sum_{i=0}^{l-1} \sum_{j=0}^{w-1} K_{i,j} 2^{wi+j} \\ &= \sum_{j=0}^{w-1} \left( \sum_{i=0}^{l-1} K_{i,j} 2^{wi+j} \right) \\ &= \sum_{j=0}^{w-1} 2^j \left( \sum_{i=0}^{l-1} K_{i,j} 2^{wi} \right). \end{aligned}$$

By precomputing all the possible values for $\sum_{i=0}^{l-1} K_{i,j} 2^{wi}$, it takes $w-1$ doubles and $w-1$ additions on average to compute $kP$ .

In order to simplify the notation, we define

$$[a_{l-1}, a_{l-2}, \ldots, a_1, a_0]P := a_{l-1}2^{(l-1)w}P + a_{l-2}2^{(l-2)w}P + \ldots + a_1 2^w P + a_0 P.$$

These values are precomputed for all possible $(a_{l-1},\dots,a_0) \in \{0,1\}^l$, resulting in Algorithm 24, adapted from Hankerson *et al.* [43, Alg. 3.44].

```
Algorithm 24: Fixed-Base Comb
Input: Affine point $P$, positive integer $k$ with $k$ written as $l$ bit-strings
of length $w$: $K_{l-1}||K_{l-2}||\cdots||K_1||K_0$
Output: Affine point $kP$
(1)  Offline Precomputation:
(2)      Compute $[a_{l-1},a_{l-2},\dots,a_1,a_0]P$ for all $(a_{l-1},\dots,a_0) \in \{0,1\}^l$
(3)  $Q \leftarrow \infty$
(4)  foreach $j$ from $w-1$ down to $0$
(5)      $Q \leftarrow 2Q + [K_{l-1,j},\dots,K_{0,j}]P$
(6)  return $Q$
```

The average running time of Algorithm 24 is

$$\left(\frac{2^l-1}{2^l}(w-1)\right)A + (w-1)D.$$

The algorithm also requires the storage of $2^l - 2$ temporary points.

Precomputing the points in affine form and taking $2\mathcal{J}+\mathcal{A} \to \mathcal{J}$ as the operation when the point added is non-zero and $2\mathcal{J} \to \mathcal{J}$ otherwise, the average field cost of this algorithm is

$$\left(\frac{2^l-1}{2^l}(w-1)\right)(12M+7S)+\left((w-1)-\frac{2^l-1}{2^l}(w-1)\right)(4M+4S)+(I+3M+S).$$

Table 3.14 lists the cost of Algorithm 24 for $d$ chosen to correspond to the length of the NIST prime $P192$. The variable $w = \lceil \frac{d}{l} \rceil$ where $d$ is the size of the input and $l$ is given. The tables for $d$ chosen to correspond with $P224$, $P256$, $P384$ and $P521$ are similar and can be found in Appendix B.

In comparison to the fixed-base window, the comb method is generally faster. Fixed-base windowing is only competitive with the comb method when the amount of storage space available is in the optimal range for fixed-base windowing.

Table 3.14: Fixed-Base Comb Average Cost ($d = 192$)

| $l$ | Storage | Field Cost | $M$-cost |
|---|---|---|---|
| 2 | 2 | I+953.0M+594.8S | 1508.8M |
| 3 | 6 | I+696.0M+418.4S | 1110.7M |
| 4 | 14 | I+543.5M+321.2S | 880.5M |
| 5 | 30 | I+449.5M+263.4S | 740.2M |
| 6 | 62 | I+371.1M+216.5S | 624.4M |
| 7 | 126 | I+325.3M+189.4S | 556.8M |
| 8 | 254 | I+278.3M+161.7S | 487.7M |

## 3.4 Known Multiplier Scalar Multiplication

The case when a multiplier is known in advance occurs in certain elliptic curve protocols such as ECIES [9]. In this section, we describe some techniques for computing a scalar multiple of a point using information precomputed from the multiplier. The mathematical construct that is used to do this is an addition chain.

### 3.4.1 Addition Chains

**Definition 3.4.1.** *An* addition chain *for an integer $k$ is given by a pair of sequences $(c, d)$ such that*

$$\begin{aligned} &c = (c_0, \ldots, c_s), c_0 = 1, c_s = k \\ &c_i = c_{j_i} + c_{k_i} \quad \textit{for all} \quad 1 \le i \le s \textit{ with respect to} \\ &d = (d_0, \ldots, d_s), d_i = (j_i, k_i) \quad \textit{and} \quad 0 \le j, k \le i - 1. \end{aligned} \tag{3.2}$$

*The* length *of the addition chain is $s$.*

An addition chain for $k$ is a sequence of integers, with each element a sum of two previous elements, 1 as the first element, and $k$ as the last. The vector $c$ holds the sequence of integers in the chain and $d$ describes which previous two integers are summed to get the corresponding integer in $c$. Often, only $c$ is given since it is easy to derive a sequence $d$ from $c$.

An addition chain provides a method for computing the scalar multiple of a point, namely by computing each scalar multiple in the chain. For example, the following three sequences are addition chains for 15: $c_a = (1, 2, 3, 4, 7, 8, 15), c_b = (1, 2, 3, 6, 7, 14, 15), c_c = (1, 2, 3, 6, 12, 15)$. Notice that $c_a$ corresponds to the right-to-left binary algorithm and $c_2$ corresponds to the left-to right binary algorithm. In fact, the binary representation of an integer $k$ corresponds to a specific type of addition chain for $k$. Also notice that the length of $c_1$ and $c_2$ are 6 and that the length of $c_3$ is 5.

For a given $k$, the smallest $s$ such that there exists an addition chain of length $s$ is denoted by $l(k)$. By exhaustive search, we see that $l(15) = 5$, but the determination of $l(k)$ for larger $k$ can be quite difficult. Since the cost for addition and doubling can be different, it can be relevant to count the number of additions and doublings in the chain. Say that $w(k) = (a_k, b_k)$ where $a_k$ is the number of times that $j_i \neq k_i$ and $b_k$ is the number of times that $j_i = k_i$.

For elliptic curves, subtraction is the same speed as addition, so a useful generalization of an addition chain is an addition-subtraction chain.

**Definition 3.4.2.** *An* addition-subtraction chain *for an integer $k$ is given by a triple* *of sequences* $(c, d, e)$ *such that*

$$\begin{aligned}
&c = (c_0, \ldots, c_s),\ c_0 = 1, c_s = k \\
&c_i = c_{j_i} + e_i c_{k_i} \quad \textit{for all} \quad 1 \le i \le s \textit{ with respect to} \\
&d = (d_0, \ldots, d_s),\ d_i = (j_i, k_i) \quad \textit{and} \quad 0 \le j, k \le i-1, \\
&e = (e_0, \ldots, e_s),\ e_i = \pm 1.
\end{aligned}$$

*The* length *of the addition-subtraction chain is s.*

Addition-subtraction chains are useful for elliptic curve scalar multiplication because negation is essentially free on an elliptic curve. A minimal addition-subtraction chain can be shorter than a minimal addition chain for a given integer. For example, $c = (1, 2, 4, 8, 16, 32, 31)$ is an addition-subtraction chain for 31 with length 6 while $l(31) = 7$.

We will present a method by Bergeron *et al.* [11] for computing a short addition chain for an integer. First, we must introduce two operations on addition chains, $\oplus$ and $\otimes$. If $c = (c_0, \ldots, c_s)$ and $c' = (c'_0, \ldots, c'_t)$, then

$$c \otimes c' = (c_0, \ldots, c_s, c_s c'_0, \ldots, c_s c'_t),$$

and if $j$ is an integer in the chain $c$,

$$c \oplus j = (c_0, \ldots, c_s, c_s + j).$$

Suppose that $(1, \ldots, n \bmod k, \ldots, k)$ is an addition chain for $k$ and $(1, \ldots, \lfloor n/k \rfloor)$ is a chain for $\lfloor n/k \rfloor$ for integers $k, n$, then

$$(1, \ldots, n \bmod k, \ldots, k) \otimes (1, \ldots, \lfloor n/k \rfloor) \oplus (n \bmod k), \tag{3.3}$$

is a chain for $n$. The method proposed by Bergeron *et al.* is to choose the value of $k$ appropriately and apply Equation (3.3) recursively. When $k = \lfloor n/2 \rfloor$, then this method produces the addition chain used in the left-to-right binary algorithm. Bergeron *et al.* proposed a method based on two procedures, *minchain* and *chain*. The *minchain(n)* procedure proceeds by producing a short addition chain for $n$ . The *chain(n, k)* procedure computes $r = n \bmod k$, and if $r = 0$, then *minchain* is applied recursively to $n$ and $n/k$ and these chains are combined with $\otimes$. If $r \neq 0$, then Equation (3.3) is applied, which recursively calls *chain(k, r)* and *minchain(q)* and combines them with *chain(k, r)*$\otimes$ *minchain(q)*$\oplus r$.

```
    minchain(n)
(1)     if $n = 2^l$ then return $(1, 2, 4, \ldots, 2^l)$
(2)     if $n = 3$ then return $(1, 2, 3)$
(3)     return chain$(n, 2^{\lfloor \log n/2 \rfloor})$
    chain(n, k)
(1)     $q \leftarrow \lfloor n/k \rfloor$ and $r \leftarrow n \bmod k$
(2)     if $r = 0$ then return (minchain$(k)$ $\otimes$ minchain$(q)$)
(3)     else return chain$(k, r)\otimes$ minchain$(q)\oplus r$
```

More advanced variants on this method are described by Bergeron *et al.* [10]. Methods for determining a short chain for a given integer include Knuth's power tree method [52], Kunihiro and Yamamoto's method [57] and Yacobi's variation on the Lempel-Ziv compression algorithm [97]. Nedjah [77] examines methods based on genetic algorithms.

The shortness of an addition-subtraction chain can be measured in different ways. For known multiplier elliptic curve scalar multiplication, the natural measure would be to equate an addition-subtraction chain with the operations required to compute it. For each term of an addition chain, when $d_i = (j_i, k_i)$ has $j_i = k_i$, then the term corresponds to a doubling, and when $j_i \neq k_i$, then it corresponds to an addition. We can also measure the storage space required to compute the scalar multiplication.

In Algorithm 25 we compute the scalar multiple of a point when given an addition-subtraction chain $(c, d, e)$ for the multiplier. The algorithm works by mirroring the construction of the chain. For each $c_i$, we compute $c_i P$ by doing the following: if $e_i = 1$, we add $c_{j_i} P$ to $c_{k_i} P$; if $e_i = -1$, we subtract, and if $j_i = k_i$ we double $c_{j_i} P$. This algorithm is adapted from Cohen *et al.* [6, Alg. 9.41].

```
Algorithm 25: Scalar Multiplication Using Addition Chains
Input: Affine point $P$, integer $k$ with addition-subtraction chain
$(c, d, e)$, as defined in Def. 3.4.2
Output: Affine point $kP$
(1)    $Q_0 \leftarrow P$
(2)    for $i$ from 1 to $s$
(3)        if $j_i = k_i$ then $Q_i \leftarrow 2Q_{j_i}$
(4)                       else $Q_i \leftarrow Q_{j_i} + e_i Q_{k_i}$
(5)    return $Q_s$
```

If $a$ is the number of terms for which $j_i = k_i$ then the estimated running time of Algorithm 25 is

$$aA + (s - a)D.$$

The choice of coordinates used depends on the values of $a$ and $s$. Generally, Chudnovsky Jacobian coordinates should be used for large values of $a$ relative to $s$ and Jacobian coordinates otherwise.

For suitably chosen addition-subtraction chains, Algorithm 25 can be faster than the algorithms for unknown point scalar multiplication. Bergeron's method will compute a short addition chain for an integer but not an addition-subtraction chain. An algorithm for determining an efficient addition-subtraction chain for elliptic curve scalar multiplication needs to take into account the length of the chain as well as the number of additions versus doublings in the chain. As of yet, there is no thorough analysis of efficient algorithms to compute addition-subtraction chains that are optimal in this fashion. This is an interesting avenue of future research.

# Chapter 4

# Parallel Scalar Multiplication Methods

In this chapter, we apply parallel methods to the problem of elliptic curve scalar multiplication. Parallel computation is used to reduce the total time required to perform a calculation by using multiple processors working in tandem. Optimally, the computation is split evenly between processors. Many elliptic curve scalar multiplication algorithms cannot be optimally parallelized because of the inherent structure of the operation, but some advantages can be obtained by parallelization.

The goal of this chapter is to present, analyze and compare several techniques for improving the running time of scalar multiplication using parallelization. Parallelization can be done on two levels, the elliptic curve operation level and the finite field operation level. We will focus only on the elliptic curve operation level; that is, we will assume that the elliptic curve operations are primitive.

Our analysis is based on a variant of the shared memory parallel computation model with non-uniform memory access (NUMA) (see [85]). In our variant of this model, there are multiple processors with their own memory and some shared memory accessible to all processors. Communication between processors is performed by reading and writing to the shared memory, which has a fixed cost. The algorithms in this chapter require a small number of processors and only share elliptic curve points. This model is similar to the standard PRAM theoretical model for parallel complexity (see Karp and Ramachandran [51]) in that it is highly abstract and provides an estimate for the cost of the algorithm with minimal designated computer properties.

This model is more realistic than the PRAM model because it considers the cost of communication between processors. We use this model because it is simple to use for comparing algorithms and it is more robust than the standard model.

The first task to which we apply parallelization is precomputing tables of points, for example, to be used with methods such as width-$w$ NAF. New algorithms are presented for both computing the table of elements and for converting the table from Chudnovsky Jacobian to affine coordinates.

The next task to which parallelism is applied to is unknown point scalar multiplication. Although there is no known way to perfectly distribute the computation needed for scalar multiplication over several processors in the unknown point case, several methods exist to distribute some of the computation over several processors resulting in an overall faster computing time. These algorithms are based on extensions of both right-to-left and left-to-right binary methods, windowing techniques and double-base chain techniques. Other than the $p^{th}$ order method, the two-processor window right-to-left and the parallel Montgomery ladder technique, all the algorithms in this chapter are new.

## 4.1 Computing Model

A parallel computing model provides a framework for analyzing a parallel algorithm. In this chapter, we will use a model resembling shared memory architecture with non-uniform memory access (NUMA) (see [85]). In our parallel computation model, there is a set amount of shared memory and each processor has a certain amount of memory that it can access in a fast manner. A processor can read from or write to the shared memory, but there is a cost for doing so. We will use the variable $s_1$ to denote the cost associated with one processor reading or writing in the shared memory space. Since the cost of reading from and writing to shared memory depends on the amount of data to read or write, we will assume that $s_1$ is the cost of reading or writing one Chudnovsky Jacobian point because it is the most common representation used in this chapter and because it requires the most space of any representation.

Our model is compatible with the standard shared memory computing model, where all processors can read and write to the same memory space at no cost. This is achieved by simply setting $s_1$ to zero.

Just as in Chapter 3, we provide estimates for the number of additions ($A$), doubles ($D$) and triples ($T$) needed to execute the algorithms. Also, we determine the average field cost and $M$-cost of the algorithms for multipliers of sizes $192, 224, 256, 384$ and $521$ bits, in order to correspond with curves over the NIST primes. We will determine the computational cost of an algorithm by determining the time from the start of the first processor to the end of the last processor in terms of field cost. Assuming this processor begins immediately and has no waiting time, we call this cost the *effective cost* of the algorithm. This is the first careful comparison of these algorithms.

We are also concerned with the granularity of the algorithm; that is, the total number of communications needed between processors. In addition to the computation cost, the average number of reads and writes to the shared memory in terms of $s_1$ is estimated and included in the analysis. Since this value is indeterminate, we make certain reasonable assumptions about the maximal value of $s_1$. These are noted as they are needed.

## 4.2 Parallel Precomputation

The precomputation step in the window NAF algorithm from Section 3.2.3 can be efficiently parallelized. The values that need to be precomputed are $P_i = iP$ for $i \in \{1, 3, 5, \ldots, 2^w - 1\}$. The algorithms presented in this section for parallelizing the precomputation are new and are not found in the literature.

Algorithm 26 computes this set of points using two processors by computing $4P$ on the first processor, $3P$ on the second processor and then all multiples of $P$ that are 1 mod 4 on the first processor and 3 mod 4 on the second processor by repeatedly adding $4P$ to $P$ and $3P$ respectively.

```
Algorithm 26: Two-Processor Table Computation
Input: Affine point $P$, integer $w$
Output: Chudnovsky Jacobian points $P_i = iP$ for $i \in \{1, 3, \ldots, 2^{w-1} - 1\}$
(1)  $P_1 \leftarrow P$
(2)  processor 1:
(3)      $P_4 \leftarrow 2P_1$
(4)      $P_4 \leftarrow 2P_4$
(5)      write $P_4$ to shared memory
(6)      foreach $i \in \{5, 9, \ldots, 2^{w-1} - 3\}$
(7)          $P_i \leftarrow P_{i-4} + P_4$
(8)  processor 2:
(9)      $P_3 \leftarrow 2P_1$
(10)     $P_3 \leftarrow P_1 + P_3$
(11)     wait for $P_4$ to be written to memory, read
(12)     foreach $i \in \{7, 11, \ldots, 2^{w-1} - 1\}$
(13)         $P_i \leftarrow P_{i-4} + P_4$
(14) return $\{P_1, P_3, \ldots, P_{2^{w-1}-1}\}$
```

Reading $P_4$ in Step 11 causes the second processor to be the last to finish, because both processors perform the same number of additions after $P_4$ is obtained. Assuming

$A > D$, the time requirement for Algorithm 26 is

$$D + A + (2^{w-3} - 1)A,$$

with communication time $2s_1$. This is nearly twice as fast as the precomputation using one processor in Algorithm 17, which requires $(D + (2^{w-2} - 1)A)$.

This algorithm can be generalized to $N = 2^n$ processors as long as $w > n$. On processors $2^i$ for $2 \le i \le n$, compute $2^{i+1}P$. Using these values, compute $(2i-1)P$ on processor $i$. The points

$$P, 3P, \ldots, (2i-1)P, \ldots, (2^{n+1} - 1)P$$

will form a set similar to a spanning set or basis for the points $P, 3P, \ldots, (2^{w-1}-1)P$. Once $2^{n+1}P$ is computed, we compute all $kP$ where $k \equiv 2i-1 \bmod 2^{n+1}$ by repeatedly adding $2^{n+1}P$ to $(2i-1)P$ on processor $i$. Continue computing $kP$ for $k \le 2^{w-1} - 1$. When $n = 1$, the points $P$, $3P$, and $4P$ are computed, then $4P$ is added to $P$ and $3P$ repeatedly to compute all the $kP$ where $k \equiv 2i - 1 \bmod 4$, just like in Algorithm 26. Algorithm 27 performs this operation.

```
Algorithm 27: $2^n$-Processor Table Computation
Input: Affine point $P$, integer $w$
Output: Chudnovsky Jacobian points $P_i = iP$ for $i \in \{1, 3, \ldots, 2^{w-1}-1\}$
(1)  $P_1 \leftarrow P$
(2)  processor $i$:
(3)  for $i \in \{1, \ldots, 2^n\}$ do:
(4)      if $i \ge 2^l$ and $i < 2^{l+1}$ for some integer $l > 0$
(5)          $P_{2^{l+1}} \leftarrow 2^{l+1}P$
(6)      find the number of digits in $2i+1$, $d_i \leftarrow \lfloor \log_2(2i+1) \rfloor + 1$
(7)      $P_{2i+1} \leftarrow \infty$
(8)      wait for processor $i - 2^{d_i - 1}$ to write $(2i+1-2^{d_i})P$ to memory
         then read
```

```
(9)         $P_{2i+1} \leftarrow 2^{d_i}P + (2i+1-2^{d_i})P \ (= (2i+1)P)$
(10)        write $P_{2i+1}$ to shared memory
(11)        wait for $2^{n+1}P$ to be written to memory by processor $2^n$, read
(12)        foreach $j \in \{(2i+1)+2^{n+1}, \dots, (2i+1)+(2^{w-n-1}-1)2^{n+1}\}$
(13)            $P_j \leftarrow P_{j-2^{n+1}} + 2^{n+1}P$
(14)    return $\{P_1, P_3, \dots, P_{2^{w-1}-1}\}$
```

Algorithm 27 is equivalent to Algorithm 26 when $n = 1$. The time that processor $i$ will have to wait in Step 8 is maximal when $i = 2^n - 1$ since it needs to wait for processor $2^{n-1} - 1$, which in turn needs to wait for processor $2^{n-2} - 1$, etc. which has to wait for processor 3 which needs to wait for processor 2. Processor $2^n$ only needs to compute $n + 1$ doublings and one addition. Processor 2 adds a doubling and each other processor adds an addition. For this case, the time needed to wait is thus $D + (n-1)A$. Therefore the worst case running time of Steps 1 through 9 is at most

$$D + nA,$$

with $(n-1)s_1$ reads/writes. Step 11 will have running time

$$(2^{w-2-n} - 1)A$$

for each processor and therefore the last processor to finish calculations will do so in

$$D + nA + (2^{w-2-n} - 1)A,$$

with communication cost $((n+1) + (2^{w-2-n} - 1))s_1$. Algorithm 27 is fast when you disregard the communication cost since it essentially reduces the number of additions by a factor of nearly $N$ while adding only $\log_2 N$ additions in comparison with the serial version.

Just as in the serial version of this algorithm, the point representation used in Algorithm 27 is designed to take advantage of the fact that the most common operation is point addition. Arithmetic using Chudnovsky Jacobian coordinates allows for the most efficient addition, and Jacobian coordinates allow for the fastest doubling, therefore the scheme we use is a combined affine-Chudnovsky-Jacobian scheme. The formula used for the first doubling is $(2\mathcal{A} \rightarrow \mathcal{J})$, the repeated doubling to $2^nP$ will be $2\mathcal{J} \rightarrow \mathcal{J}$, and the final double to $2^{n+1}P$ will be $2\mathcal{J} \rightarrow \mathcal{J}^c$. The first addition will be either $(\mathcal{A} + \mathcal{J}^c \rightarrow \mathcal{J}^c)$ or $(\mathcal{J}^c + \mathcal{J}^c \rightarrow \mathcal{J}^c)$, and the repeated addition will be $\mathcal{J}^c + \mathcal{J}^c \rightarrow \mathcal{J}^c$. This results in an effective field cost of

$$\begin{aligned}(2M + 4S) + n(11M + 3S) &+ (2^{w-2-n} - 1)(11M + 3S) \\ &= (11n + 11 \cdot 2^{w-2-n} - 9)M + (3n + 3 \cdot 2^{w-2-n} + 1)S.\end{aligned}$$

If we consider the time it takes to send information from one processor to another, there is an additional factor of $(n+1)s_1$, with $(2^{w-2-n} - 1) \cdot s_1$ if all the precomputed values are subsequently written to shared memory.

It may be useful to convert the table of precomputed values from the previous section to affine form, as was done in Section 3.2.3. This can be parallelized to $2^n$ processors by splitting the set of points into $2^n$ equal sets and applying Algorithm 54 to each set using a different processor. Notice that at the end of Algorithm 27, the set of points computed in parallel is already conveniently partitioned into these sets.

The effective field cost of converting $2^{w-2}-1$ points $P_i$ from Chudnovsky Jacobian to affine form for $i \in \{3, \ldots, 2^{w-1} - 1\}$ on $2^n$ processors is the cost of Algorithm 54 on the largest partition. The set of points is partitioned into $2^n$ sets, so the largest set is of size $2^{w-2-n}$. Therefore the effective field cost is

$$I + \left(6 \cdot 2^{w-2-n} - 3\right) M + \left(2^{w-2-n}\right) S.$$

This method for the parallel precomputation of a table will be useful in the next section, which deals with the parallelization of scalar multiplication.

## 4.3 Parallel Unknown Point Scalar Multiplication

Unknown point scalar multiplication was introduced in Section 3.2. In this section we examine parallel algorithms for this operation derived mostly from variations of the binary algorithm.

In order to create a parallelized version of a task, we must examine the sequence of dependencies in the algorithm's execution. In the case of the binary algorithm for unknown point scalar multiplication, we can not overcome the need for performing a large number of doublings in sequence. The gain that can be achieved by parallelization in this case stems from the parallelization of additions. Since not all of the algorithm can be parallelized, Amdahl's law [2] implies that massive parallelization brings diminishing returns. We therefore limit our parallelization to a small number of processors. Specifically, we will examine algorithms on 2 to 8 processors.

The algorithms we examine are the $p^{th}$ order binary method (Section 4.3.1), several right-to-left and left-to-right algorithms (Sections 4.3.2 to 4.3.7), the double-base $n$-chain method (Section 4.3.9) and Montgomery's method (Section 4.3.10). Many of the left-to-right, right-to-left and double-base $n$-chain methods are new and can not be found in the literature. The two results we are concerned with in this section are effective field cost and granularity. A summary of the results of this section are presented in Section 4.3.11.

### 4.3.1 $p^{th}$ Order Binary

This method was introduced by Garcia and Garcia [38] as a way to reduce the number of additions needed to calculate a scalar multiplication using additional processors. The term $p^{th}$ order is in reference to using $p$ processors. In this thesis we use the variable $N$ to represent the number of processors. This method is based on splitting the multiplier $k$ into a number of integers with a small number of non-zero digits that sum to $k$. We present a novel algorithm that improves on that of Garcia and Garcia by using the NAF of the multiplier rather than the binary representation.

Let $k$ be the multiplier and $w$ be a parameter denoting window size. Suppose $d$ is the length of NAF($k$). Define $l := \lceil d/w \rceil$. Then NAF($k$) $= K_{l-1}||\ldots||K_0$ where each $K_i$ is a signed bitstring of length $w$, padded with zeros if needed.

Each $K_i$ can be written as $K_{i,0} + K_{i,1}2 + \cdots K_{i,w-1}2^{w-1}$ for $K_{i,j} \in \{-1, 0, 1\}$. Notice that

$$k = \sum_{\substack{0 \le i \le l-1, \\ 0 \le j \le w-1}} K_{i,j} 2^{iw+j}.$$

If we define

$$g_j = \sum_{0 \le i \le l-1} K_{i,j} 2^{iw+j}$$

for $0 \le j \le w - 1$, then

$$k = \sum_{0 \le j \le w-1} g_j.$$

Each $g_j$ is zero for all but $1/w$ of the digits of $k$, and therefore has approximately $w$ times fewer non-zero digits in its binary representation than $k$. For example, if

$w = 4$ and $\mathrm{NAF}(k) = (1, 0, \bar{1}, 0\,, 0, 0, 0, \bar{1}\,, 0, 1, 0, 1)$, we obtain

$$g_0 = (1, 0, 0, 0,\ 0, 0, 0, 0, 0, 0, 0, 0)$$

$$g_1 = (0, 0, 0, 0,\ 0, 0, 0, 0, 0, 1, 0, 0)$$

$$g_2 = (0, 0, \bar{1}, 0,\ 0, 0, 0, 0,\ 0, 0, 0, 0)$$

$$g_3 = (0, 0, 0, 0,\ 0, 0, 0, \bar{1},\ 0, 0, 0, 1).$$

Every $w^{\mathrm{th}}$ digit of each $g_j$ agrees with $k$, and the rest of the digits are 0. Calculating $g_jP$ using the binary NAF method should require around the same number of doublings as computing $kP$ but with $w$ times fewer additions.

The algorithm involves two main stages:

1. *Bit scattering*: Compute the set $\{g_0, \ldots, g_{w-1}\}$ as described above. Using $N = w$ processors, compute the set

$$\{g_0P, g_1P, \ldots, g_{w-1}P\}.$$

2. *Recombination*: Using $\lceil w/2 \rceil$ processors, calculate $\sum_{0 \le j \le w-1} g_jP$. This can be done in the time it takes for $\lceil \log_2 w \rceil$ additions using a recursive algorithm. Calculate $P_{1,j} = g_{2j}P + g_{2j+1}P$ on processor $j$ for $0 \le j \le \lfloor (w-1)/2 \rfloor$, (if there is no $g_jP$ to add, add $\infty$). Then calculate $P_{2,j} = P_{1,2j} + P_{1,2j+1}$ for $0 \le 2j \le \lfloor (w-1)/2 \rfloor$ when defined. Continue with $P_{i,j} = P_{i-1,2j} + P_{i-1,2j+1}$ for $0 \le 2^i j \le \lfloor (w-1)/2 \rfloor$. Once $2^i j > \lfloor (w-1)/2 \rfloor$, then $P_{i,1} = kP$.

Algorithm 28 performs this by first computing every $g_i$ from the bit-scattering step, then computing $g_iP$ with the binary NAF algorithm (Algorithm 15) on processor $i$. The recombination stage is performed in successive steps by processors with numbers divisible by powers of 2. Algorithm 28 is adapted almost directly from the ideas from Garcia and Garcia [38], with the NAF replacing the standard binary representation.

```
Algorithm 28: $p^{th}$ Order Binary
Input: Affine point $P$ and signed bitstrings $K_0, \ldots, K_{l-1}$ of length
$w(= N)$
Output: Affine point $kP$
(1)   $g_m \leftarrow K_{l-1,m}|| \cdots ||K_{0,m}$ where $K_{i,j}$ matches $K_i$ at binary digit
      $j$ and has 0 for every other digit.
(2)   foreach processor $j \in \{0, \ldots, N-1\}$ do:
(3)       calculate $P_j \leftarrow g_j P$ using binary NAF, write to memory.
(4)   for $i = 1$ to $\lceil \log_2 w \rceil$
(5)       processor $j$:
(6)       foreach $j \in \{0, \ldots, \lfloor (w-1)/2 \rfloor 2^{-i}\}$
(7)           if $2^i j + 2^{i-1} \le l$ then $Q_j \leftarrow P_{2^i j + 2^{i-1}}$ read from memory
(8)                                  else $Q_j \leftarrow \infty$
(9)           $P_{2^i j} \leftarrow P_{2^i j} + Q_j$, write to memory
(10)  return $P_0$
```

The average running time of Algorithm 28 is approximately

$$\left( (d-1)D + \frac{d-1}{3w} A \right) + \left( \lceil \log_2 w \rceil A \right),$$

with communication overhead of $\left( (2\lceil \log_2 w \rceil) + 1 \right) s_1$. The first term comes from computing $g_i P$ and the second from the recombination stage. This reduces the number of additions in the main section by a factor of $w$ in exchange for $\lceil \log_2 w \rceil$ additions.

For a detailed analysis of Algorithm 28 we will chose an appropriate coordinate system. The binary NAF portion of the algorithm uses the mixed Jacobian-Affine coordinate system described in the analysis of Algorithm 15, with the exception that the final result is converted to Chudnovsky Jacobian rather than affine. This is preferable because the next steps are all additions and Chudnovsky Jacobian coordinates allow for the fastest addition.

With these choices for coordinates, the average cost for Step 3 is

$$\left(\frac{d-3}{3w}\right)(12M+7S)+(2M+4S)+\left(d-2-\frac{d-3}{3w}\right)(4M+4S)+(M+S)$$

for each processor with communication overhead $s_1$. Each execution of Steps 6 to 8 will cost $(\mathcal{J}^c+\mathcal{J}^c\rightarrow\mathcal{J}^c)$ and $(2s_1)$, and the return step costs $(I+3M+S)$. This results in an effective field cost of

$$\begin{aligned}&\left(\frac{d-3}{3w}\right)(12M+7S)+(2M+4S)\\&\qquad+\left(d-2-\frac{d-3}{3w}\right)(4M+4S)+(M+S)\\&\qquad\qquad+(\lceil\log_2 w\rceil)(12M+4S)+(I+3M+S)\end{aligned}$$

$$\begin{aligned}&=I+\left(4d+8\cdot\frac{d-3}{3w}+12\lceil\log_2 w\rceil-2\right)M+\\&\qquad\qquad\left(4d+3\cdot\frac{d-3}{3w}+4\lceil\log_2 w\rceil-2\right)S,\end{aligned}$$

with communication overhead of $((2\lceil\log_2 w\rceil)+1)\,s_1$.

Tables 4.1 describes the field costs and $M$-costs of Algorithm 28 with $d$ corresponding to the length of the NIST prime $P192$. Recall that the $M$-cost is the field cost with substitutions $S\leftarrow(4/5)M$ and $I\leftarrow 80M$. The tables for $d$ chosen to correspond to $P224$, $P256$, $P384$ and $P521$ are similar and can be found in Appendix B. The advantages of this algorithm are that it outperforms the fastest one-processor algorithms and it has a relatively low amount of communication overhead. The next algorithm takes an orthogonal view to splitting up the additions.

Table 4.1: $p^{th}$ Order Binary Method Average Cost ($d = 192$)

| $N$ | Field Cost | $M$-cost | $s_1$ |
|---|---|---|---|
| 2 | I+1030.0M+864.5S | 1801.6M | 3 |
| 3 | I+958.0M+837.0S | 1707.6M | 5 |
| 4 | I+916.0M+821.2S | 1653.0M | 5 |
| 5 | I+902.8M+815.8S | 1635.4M | 7 |
| 6 | I+886.0M+809.5S | 1613.6M | 7 |
| 7 | I+874.0M+805.0S | 1598.0M | 7 |
| 8 | I+865.0M+801.6S | 1586.3M | 7 |

### 4.3.2 Right-to-Left Parallel

The troublesome part of the $p^{th}$ order method is the recombination stage (Steps 4 to 8). This stage can be avoided by taking an alternative approach to bit-scattering that is reminiscent of the right-to-left binary algorithm. In this section, we present a new algorithm that implements this idea.

As in the previous section, suppose $d$ is the length of NAF($k$). Define $w \in \mathbb{N}$ to be the word length, and define $l := \lceil d/w \rceil$ to be the number of words in NAF($k$). Then NAF($k$) $= K_{l-1}||\dots||K_0$ where each $K_i$ is a signed bitstring of length $w$, padded with zeros if needed.

The key observation for this algorithm is that $k = K_0 + K_1||0_w + \cdots + K_{l-1}||0_{(l-1)w}$ where $0_k$ represents $k$ consecutive zeros. Rather than splitting the multiplier into a set of long sparse integers, it can be split into $K_0, K_1||0_w, \cdots, K_{l-1}||0_{(l-1)w}$. Each of these integers has a low Hamming weight, but they are not all the same length as $k$.

Each $K_iP$ is calculated in parallel for $0 \leq i \leq l-1$ with $N = l$ processors. Then a repeated doubling algorithm is used to calculate $(K_j||0_w)P$ on processor $j$. Once $K_0P$ is calculated, it is written to shared memory to be read by the processor calculating $(K_1||0_w)P$. That processor then calculates $K_0P + (K_1||0_w)P = (K_1||K_0)P$ and sends the result to the next processor, which computes $(K_1||K_0)P + (K_2||0_{2w}) = (K_2||K_1||K_0)P$, and so on. In this arrangement, each processor is in use only as long as doubling is needed, so the recombination is performed by processors that have finished early, eliminating the need for a recombination stage. The implicit assumption we have made is that $j$ doublings take longer than 2 reads from shared memory. Algorithm 29, which does not appear in the literature, performs this operation.

```
Algorithm 29: Right-to-Left Parallel
Input: Affine point $P$, $l(= N)$ signed bitstrings $K_0, \dots, K_{l-1}$ of length
$w$
Output: Affine point $kP$
(1)     foreach processor $i \in \{0, \dots, l-1\}$, simultaneously perform
        the following:
(2)        $P_i \leftarrow K_iP$ using the binary NAF algorithm
(3)        $P_i \leftarrow 2^{iw}P_i$ using consecutive doubling
(4)        if $i = 0$
(5)           write $P_i$ to memory
(6)        else
(7)           wait until $P_{i-1}$ is written to memory, read
(8)           calculate $P_i \leftarrow P_i + P_{i-1}$
(9)           if $i < l-1$ then write $P_i$ to memory
(10)       return $P_{l-1}$
```

The average running time of Algorithm 29 is

$$\left((w-1)D + \frac{w-3}{3}A\right) + ((l-1)wD + A).$$

The first term comes from computing $K_{l-1}P$ and the second from the repeated doubling and final addition. The coordinate system used for Algorithm 29 is the standard Jacobian-affine for the binary NAF algorithm and Jacobian for the doubling. Before the final point is sent to processor $l-1$, it is converted to Chudnovsky Jacobian coordinates so $\mathcal{J} + \mathcal{J}^c \rightarrow \mathcal{J}$ is used for the final addition.

Table 4.2: Right-to-Left Parallel Method Average Cost ($d = 192$)

| $N$ | Field Cost | $M$-cost | $s_1$ |
|---|---|---|---|
| 2 | I+1024.0M+861.0S | 1792.8M | 1 |
| 3 | I+938.7M+829.0S | 1681.9M | 1 |
| 4 | I+896.0M+813.0S | 1626.4M | 1 |
| 5 | I+884.0M+816.0S | 1616.8M | 1 |
| 6 | I+853.3M+797.0S | 1570.9M | 1 |
| 7 | I+858.7M+809.0S | 1585.9M | 1 |
| 8 | I+832.0M+789.0S | 1543.2M | 1 |

With this configuration, Step 2 has average field cost

$$\left(\frac{w-3}{3}\right)(12M+7S)+(2M+4S)+\left(w-2-\frac{w-3}{3}\right)(4M+4S).$$

For processor $l-1$, Step 3 requires $(d-1)w(2\mathcal{J}\to\mathcal{J})$ and Step 7 $\mathcal{J}+\mathcal{J}^c\to\mathcal{J}$. The return step costs $(I+3M+S)$. This results in an effective field cost of

$$\left(\frac{w-3}{3}\right)(12M+7S)+(2M+4S)+\left(w-2-\frac{w-3}{3}\right)(4M+4S)+$$
$$(l-1)w(4M+4S)+(11M+3S)+(I+3M+S),$$
$$=I+\left(\frac{8w}{3}+4lw\right)M+(w+1+4lw)S$$

with communication overhead $s_1$.

Table 4.2 describes the field costs and $M$-costs of Algorithm 28 with $d$ corresponding to the length of the NIST prime $P192$. The tables for $d$ chosen to correspond with $P224$, $P256$, $P384$ and $P521$ are similar and can be found in Appendix B.

This new algorithm has all the advantages of the $p^{th}$ order algorithm as well as a lower cost and communication time due to the elimination of the recombination step. Note that it might be possible to reduce the effective cost of this algorithm by using the parallel precomputation algorithm from Section 4.2. In the next algorithm, we will attempt to reduce the total running time of this algorithm.

### 4.3.3 Right-to-Left Parallel With Hedging

In Algorithm 29, the processors finished in succession with the last processor containing the final result. This allowed the additions performed on the other processors to essentially have no net effect on the total running time of the operation. The only time spent on additions on the first processor was in calculating $K_{l-1}P$ for the most significant block $K_{l-1}$ and performing the final addition. One way to reduce this first cost is to vary the size of the blocks in such a way that the processors still finish in sequence but the most significant block is small. We introduce a new algorithm in this section based on this idea.

The key part of the algorithm is changing the sizes of the blocks $K_i$ so that each processor will finish almost immediately before the result is needed. This is achieved by making the lower order blocks longer so that the computation takes longer on the processors that compute them.

Let us suppose that performing a scalar multiplication of $P$ by an $n$ digit number takes time $nC$ and the calculation of $2^nP$ takes time $nD$. If the largest block $K_1$ has length $L_1$, then the calculation of $K_1P$ will take time $L_1C$. The next processor must compute $2^{L_1}(K_2P)$. For this to take the same amount of time, $K_2$ should be length $L_2$ where $L_2C + L_1D \approx L_1C$. Solving this gives $L_2 \approx L_1(1 - D/C)$.

The next processor needs to compute $2^{L_2}2^{L_1}(K_3P)$ in the same time as the previous processor computed $2^{L_1}(K_2P) + K_1P$. Therefore, we need $L_3C + (L_1 + L_2)D \approx$

$L_1D + L_2C + A$. Solving this, we obtain

$$L_3 \approx (1 - D/C)L_2 + A/C \approx (1 - D/C)^2 L_1 + A/C.$$

By comparing processors $n$ and $n-1$, we obtain

$$L_n \approx (1 - D/C)L_{n-1} + A/C.$$

Evaluating this recursively, we arrive at:

$$\begin{aligned}
L_n &\approx \left(1 - \frac{D}{C}\right)^{n-1} L_1 + \sum_{i=0}^{n-3} \left(1 - \frac{D}{C}\right)^{i} \frac{A}{C} \\
&= \left(1 - \frac{D}{C}\right)^{n-1} L_1 + \frac{1 - \left(1 - \frac{D}{C}\right)^{n-2}}{1 - \left(1 - \frac{D}{C}\right)} \frac{A}{C} \\
&= \left(1 - \frac{D}{C}\right)^{n-1} L_1 + \frac{C}{D}\left(1 - \left(1 - \frac{D}{C}\right)^{n-1}\right) \frac{A}{C} \\
&= \left(1 - \frac{D}{C}\right)^{n-1} L_1 + \left(1 - \left(1 - \frac{D}{C}\right)^{n-1}\right) \frac{A}{D} \\
&= \left(1 - \frac{D}{C}\right)^{n-1} L_1 + \frac{A}{D} - \left(1 - \frac{D}{C}\right)^{n-1} \frac{A}{D} \\
&= \left(1 - \frac{D}{C}\right)^{n-2} \left(\left(1 - \frac{D}{C}\right) L_1 - \frac{A}{D}\right) + \frac{A}{D}
\end{aligned}$$

The relationship between the maximal length of the multiplier $L$ and the number of processors $N$ is

$$L = L_1 + \sum_{i=2}^{N} \left\lfloor (1 - \frac{D}{C})^{i-2} \left(\left(1 - \frac{D}{C}\right) L_1 - \frac{A}{D}\right) + \frac{A}{D} \right\rfloor . \qquad (4.1)$$

If we assume that $D = 7.2M$, $A = 15.2M$ from the $M$-cost of $2\mathcal{J} \rightarrow \mathcal{J}$ and $\mathcal{J} + \mathcal{J} \rightarrow \mathcal{J}$ and $C = 10.7M$ from $(2\mathcal{J} \rightarrow \mathcal{J}) + (1/3)(\mathcal{J} + \mathcal{A} \rightarrow \mathcal{J})$, we obtain some estimates on the number of processors needed for the final word to be of length 1 for the NIST primes.

By choosing the length of the largest block $L_1$ in Equation (4.1), we can calculate when the block size $L_i$ is 2 and when the total length of the multiplier, $L$, is larger than that of a given NIST prime. The value of $L_1$ can be determined by trial and error. If we set $L_1 = 123$, then $L_8 = 2$ and $L = 192$, $L_1 = 146$ gives $L_8 = 2$ and $L = 225$, $L_1 = 168$ gives $L_8 = 2$ and $L = 257$, $L_1 = 253$ gives $L_8 = 2$ and $l = 384$, and finally $L_1 = 352$ gives $L_8 = 2$ and $L = 521$. These values are estimates, but they indicate that even given a small cost for writing to memory, 7 or 8 processors should be enough for this algorithm. If we consider communication cost, each addition step comes with one read from memory and one write to memory, therefore we set $A = 15.2M + 2s_1$. Except for very large values of $s_1$, the block sizes can still be chosen so that at most 8 processors are needed.

Once the NAF of the integer multiplier $k$ is split into appropriate length signed bit-strings, Algorithm 30 computes the scalar multiple of a given point in the same manner as Algorithm 29. Algorithm 30 is new and does not appear in the literature.

```
Algorithm 30: Right-to-Left Parallel with Hedging
Input: Affine point $P$, $l = N$ signed bitstrings $K_i$ of length $L_i$, $i \in \{0, \ldots, l-1\}$ such that $\sum_{i=0}^{l-1} 2^{L_i} K_i = k$ and $\sum_{i=0}^{l-1} L_i = d$
Output: Affine point $kP$
(1)  foreach processor $i \in \{0, \ldots, l-1\}$
(2)      $P_i \leftarrow K_i P$ using the binary NAF algorithm
(3)      $P_i \leftarrow 2^{\sum_{i=0}^{i-1} L_i} P_l$ using consecutive doubling
(4)      if $i = 0$
(5)          write $P_i$ to memory
(6)      else
(7)          wait until $P_{i-1}$ is written to memory, read
(8)          $P_i \leftarrow P_i + P_{i-1}$
(9)          if $i < l-1$ then write $P_i$ to memory
(10) return $P_{l-1}$
```

If the parameters are correctly set, then processor $l-1$ will most likely not need to wait, and the running time of the whole algorithm is the running time of processor $l-1$. Thus the average running time of Algorithm 30 is

$$(d-1)D+A.$$

Using the same coordinate choices as in Section 4.3.2 results in an effective field cost of

$$(d-2)(4M+4S)+(2M+4S)+(12M+4S)+(I+3M+S) = I+(4d+9)M+(4d+1)S,$$

with communication overhead of $s_1$. Table 4.3 describes the field costs and $M$-costs of Algorithm 30 with $d$ corresponding to the NIST primes. Since there is only one addition contributing to the effective cost of the algorithm, this seems to be a reasonable lower bound for computing an unknown point scalar multiple in parallelizing. It must be noted that if the processor needs to wait, then this cost may be slightly higher. The next algorithms will attempt to approach the effective cost of this algorithm with fewer processors.

### 4.3.4 Two-Processor Right-to-Left

The parallelization achieved in Algorithms 29 and 30 works by spreading the addition steps throughout the processors. The two-processor algorithm takes a different

Table 4.3: Right-to-Left Parallel Method Average Cost (8 Processors)

| $d$ | Field Cost | $M$-cost | $s_1$ |
|---|---|---|---|
| 192 | I+777.0M+769.0S | 1472.2M | 1 |
| 224 | I+905.0M+897.0S | 1702.6M | 1 |
| 256 | I+1033.0M+1025.0S | 1933.0M | 1 |
| 384 | I+1545.0M+1537.0S | 2854.6M | 1 |
| 521 | I+2093.0M+2085.0S | 3841.0M | 1 |

approach. All the doubling steps are performed on one processor and the results are periodically sent to the other processor in order to combine them using additions.

In an NAF scalar multiplication, there are on average three times as many doublings as additions. If a doubling takes around half the time of that of an addition, the addition processor should not be slower than the the doubling processor.

Algorithm 31 performs scalar multiplication in this manner. One processor computes the points $2P, 4P, 8P, \ldots$ by repeated addition and writes $2^iP$ to memory when $k_i \neq 0$ while the other adds $k_i2^iP$ whenever it is written to memory. This algorithm is not found in this form in the literature.

```
Algorithm 31: Two-Processor Right-to-Left NAF
Input: Affine point $P$, integer $k$ with signed binary representation
($k_{d-1}, \ldots, k_0$)
Output: Affine point $kP$
(1)  processor 1:
(2)      $Q_1 \leftarrow P$
(3)      for $i = 1$ to $d - 1$
(4)          if $k_i = 1$ then write $Q_1$ to memory
(5)          $Q_1 \leftarrow 2Q_1$
(6)  processor 2:
(7)      $Q_2 \leftarrow \infty$
(8)      for every $Q_1$ written to memory: $Q_2 \leftarrow Q_2 + Q_1$
(9)  return $Q_2$
```

A feature of the NAF representation of integers is that no two consecutive bits in the representation are non-zero. This provides a worst-case scenario in which the representation of the integer $k$ alternates between zero and non-zero values. In such a case, the second processor would compute $d/2$ additions and the first would compute $d - 1$ doubles. In the typical case, only $d/3$ additions would be required, including a final addition that needs the result of the last doubling on the other processor in

Table 4.4: Two-Processor Right-to-Left Average Cost

| $d$ | Field Cost | $M$-cost | $s_1$ |
|---|---|---|---|
| 192 | I+777.0M+769.0S | 1472.2M | 63 |
| 224 | I+905.0M+897.0S | 1702.6M | 74 |
| 256 | I+1033.0M+1025.0S | 1933.0M | 84 |
| 384 | I+1545.0M+1537.0S | 2854.6M | 127 |
| 521 | I+2093.0M+2085.0S | 3841.0M | 173 |

order to continue. Algorithm 31 has an average-case running time of

$$(d-1)D + A,$$

Since the bulk of the effective cost is being performed by the doubling, we take $D$ to be $2\mathcal{J} \rightarrow \mathcal{J}$ and $A$ to be $\mathcal{J} + \mathcal{J} \rightarrow \mathcal{J}$. This results in a best case field cost of

$$(d-2)(4M+4S)+(2M+4S)+(12M+4S)+(I+3M+S) = I+(4d+9)M+(4d+1)S$$

There is also $(d/3-1)s_1$ in communication overhead on average. Table 4.4 describes the average cost of Algorithm 31.

Aside from the communication overhead, Algorithm 31 has the same effective field cost as Algorithm 30. This algorithm is therefore well suited to a shared-memory parallel implementation where $s_1$ is negligible. However, when $s_1$ is more costly, this algorithm will be significantly slower as it has rather fine granularity. The next section examines an algorithm that lowers the communication overhead at the cost of more elliptic curve operations.

### 4.3.5 Two-Processor Window Right-to-Left

Recall the windowing technique for known point scalar multiplication from Section 3.3.1. This algorithm can be modified so that the precomputation is included in the computation time. Möller [72] introduced a variant of this algorithm that can be parallelized to two processors and has some useful properties that protect against side-channel attacks (see Section 5.1.2). We will present a simplified version of his algorithm.

In Algorithm 31, to compute $kP$ for $\mathrm{NAF}(k) = (k_{d-1}, \ldots, k_0)$, the first processor computes $2^i P$ for all $1 \leq i \leq d-1$ and sends $2^i P$ for $k_i \neq 0$ to the other processor. The second processor then adds the points it receives together to obtain $kP$. In this algorithm, the digits of $k$ are grouped into windows and the second processor computes the sums of the $2^i P$ corresponding to each digit separately.

Suppose $k \in \mathbb{N}$ is written as

$$k = \sum_{i=0}^{l-1} b_i 2^{wi},$$

for $b_i \in B$, for some digit set $B$. For simplicity, assume

$$B = \{-2^{w-1}, \ldots, -1, 0, 1, \ldots, 2^{w-1}\}.$$

Let $B'$ denote the set $\{|b| \mid b \in B\}$ of absolute values of digits. We can write

$$k = \sum_{b \in B'} b \left( \sum_{i:b_i=b}^{l-1} 2^{wi} - \sum_{i:b_i=-b}^{l-1} 2^{wi} \right).$$

Just as $kP$ is calculated in Algorithm 31, the points

$$P_b = \sum_{i:b_i=b}^{l-1} 2^{wi} - \sum_{i:b_i=-b}^{l-1} 2^{wi}$$

can be computed with two processors, one performing the point doublings and the other performing the point additions and subtractions. We can then calculate

$$kP = \sum_{b=0}^{2^{w-1}} bP_b = \sum_{b=0}^{2^{w-1}} \sum_{j=1}^{b} P_b.$$

The algorithm proceeds in two steps, the right-to-left stage and the result stage. The right-to-left stage involves the first processor computing $2^{iw}P$ for $1 \leq i \leq l-1$ and sending the results to processor 2, which computes the points $P_b$ for $b \in B'$. In the result stage, the point $\sum_{b\in B'} bP_b$ is computed. The result stage is performed exclusively on the second processor.

Algorithm 32 performs this operation; it is a simplified version of that described by Möller [72] in that it does not use randomization to protect from side-channel attacks.

**Algorithm 32:** Two-Processor Window Right-to-Left
**Input:** Affine point $P$, integer $k$ with signed $2^w$-ary representation $(k_{l-1}, \ldots, k_0)_{2^w}$
**Output:** Affine point $kP$

```
(1)   processor 1:
(2)     $Q \leftarrow P$
(3)     for $i = 1$ to $l-1$
(4)       if $k_i \neq 0$ then write $Q, k_i$
(5)       $Q \leftarrow 2^w Q$
(6)   processor 2:
(7)     foreach $0 \leq b \leq 2^{w-1}$
(8)       $P_b \leftarrow \infty$
(9)     foreach $Q, k_i$ written in memory, read
(10)      if $k_i > 0$
(11)        $P_{k_i} \leftarrow P_{k_i} + Q$
(12)      else if $k_i < 0$
(13)        $P_{-k_i} \leftarrow P_{-k_i} - Q$
(14)    for $i = 2^{w-1} - 1$ down to $2$
(15)      $P_i \leftarrow P_i + P_{i+1}$
(16)    for $i = 2$ to $2^{w-1}$
```

```
(17)        $P_1 \leftarrow P_1 + P_i$
(18)    **return** $P_1$
```

Similarly to the previous algorithm, the second processor is the last to finish. As long as $w \geq 2$, the running time of Algorithm 32 is approximately

$$((l-1)w)D + (2^w - 2)A,$$

with communication overhead $(l-1)s_1$. The algorithm also requires the storage of $2^{w-1}$ temporary points. Assuming $D$ is performed with the $2\mathcal{J} \rightarrow \mathcal{J}$ operation and $A$ with $\mathcal{J} + \mathcal{J} \rightarrow \mathcal{J}$, the average effective running time of this algorithm is

$$(w(l-1)-1)(4M+4S) + (2M+4S) + (2^w-2)(11M+3S) + (I+3M+S)$$

$$= I + (4wl - 4w + 11 \cdot 2^w - 21)M + (4wl - 4w + 3 \cdot 2^w - 5)S.$$

In addition, $(l-1)s_1$ is required for passing points between processors.

Table 4.5 describes the field costs and $M$-costs of Algorithm 32 with $d$ corresponding to the length of the NIST prime $P192$. The tables for $d$ chosen to correspond to $P224$, $P256$, $P384$ and $P521$ are similar and can be found in Appendix B.

The results show that Algorithm 32 has a higher effective $M$-cost than Algorithm 31, but a lower communication overhead when $w \geq 3$. The main advantages of the windowed method are the savings in the number of transfers between processors and

Table 4.5: Right-to-Left Windowing Method Average Cost ($d = 192$)

| $w$ | Field Cost | $M$-cost | $s_1$ |
|---|---|---|---|
| 2 | I+783.0M+767.0S | 1476.6M | 95 |
| 3 | I+823.0M+775.0S | 1523.0M | 63 |
| 4 | I+907.0M+795.0S | 1623.0M | 47 |
| 5 | I+1091.0M+851.0S | 1851.8M | 38 |

the measures that can be taken against side-channel attacks. However, the side-channel resistant version of this algorithm presented by Möller is more costly than the version presented here.

### 4.3.6 Left-to-Right Parallel

Algorithm 32 used a technique analogous to the right-to-left binary algorithm for modular exponentiation. In Sections 4.3.1 to 4.3.5, we will examine algorithms that take an alternative approach analogous to the left-to-right binary algorithm.

Let $k$ be an integer multiplier with $\mathrm{NAF}(k) = (k_{d-1}, \ldots, k_0)$. Then we can write $k$ in terms of blocks of length $w$ as follows

$$K_0 = (k_{w-1}, \ldots, k_0), \ldots, K_{l-1} = (0, \ldots, k_{d-1}, \ldots, k_{(l-1)w}),$$

where $K_{l-1}$ is padded with zeros as needed.

The first step of the $w$-bit left-to-right algorithm uses $l = N$ processors to calculate the values $K_iP$ for $0 \le i < l-1$. The point $kP$ is calculated by taking $K_{l-1}P$, doubling it $w$ times, adding the value $K_{l-2}P$, then doubling $w$ times and repeating for each $K_iP$ until finally $K_0$ is added, obtaining $kP$.

Algorithm 33 is a new algorithm that performs scalar multiplication in this manner.

```
Algorithm 33: Left-to-Right Parallel
Input: Affine point $P$, $l(= N)$ signed bitstrings $K_0, \ldots, K_{l-1}$ of length
$w$
Output: Affine point $kP$
(1)   foreach processor $i \in \{0, \ldots, l-1\}$
(2)       $P_i \leftarrow K_iP$ using the binary NAF algorithm
(3)       if $i < l-1$ then write $P_l$ to memory
(4)       if $i = l-1$
```

```
(5)        foreach $j$ from $l-2$ down to 0
(6)            wait until $P_i$ is written and read
(7)            $P_{l-1} \leftarrow 2^w P_{l-1}$ using consecutive doubling
(8)            $P_{l-1} \leftarrow P_{l-1} + P_j$
(9)    return $P_{l-1}$
```

The average running time of Algorithm 33 is

$$\left(wD + \frac{w-3}{3}A\right) + ((l-1)wD + (l-1)A)\,,$$

with communication overhead $(l-1)s_1$. The first term comes from computing $K_{l-1}P$, the second term is from the repeated doubling and the addition of the other $K_iP$ to the total. The drawback of this algorithm compared to Algorithm 29 is that the additions used to combine the values computed on each processor are performed on the main processor $l-1$ and therefore contribute to the total running time. Specifically, $l-2$ additional additions are needed in comparison with Algorithm 33.

After the initial computation of the values $K_iP$, all the processors except for $l-1$ are idle. This observation leads to the next algorithm, which has the same running time but only uses 3 processors.

### 4.3.7 Left-to-Right (3 Processors)

In Algorithm 33, all but one of the processors are idle for most of the computation. Also, the values computed by the other processors are not needed immediately. These two facts allow the number of processors to be reduced to three.

The algorithm below consists of the same operations as the previous one except that the values $K_0P, \ldots, K_{l-2}P$ are computed by two processors concurrently while another processor performs the doubling. Three processors are needed rather than two because the time required to compute $K_iP$ is on average more than the time it takes to compute $w$ doubles and one add.

Algorithm 34 is a new algorithm that performs this operation.

```
Algorithm 34: Left-to-Right (3 Processors)
Input: Affine point $P$, $l$ signed bitstrings $K_0, \ldots, K_{l-1}$ of length $w$
Output: Affine point $kP$
(1)   foreach processor $j \in \{0, 1\}$ simultaneously perform
(2)      for $i = l - 3 + j$ down to 0 or 1 by 2
(3)         $P_i \leftarrow K_i P$ using the binary NAF algorithm, write to mem-
            ory
(4)   for processor 2 simultaneously perform
(5)      $P_i \leftarrow K_{l-1} P$ using the binary NAF algorithm
(6)      for $i = l - 2$ to 0
(7)         wait until $P_i$ is written to memory, read
(8)         $P_{l-1} \leftarrow 2^w P_{l-1}$ using consecutive doubling
(9)         $P_{l-1} \leftarrow P_{l-1} + P_i$
(10)  return $P_{l-1}$
```

The average running time of Algorithm 34 is

$$\left(wD + \frac{w}{3}A\right) + ((l-1)wD + (l-1)A),$$

the same as Algorithm 33. The first term comes from computing $K_{l-1}P$, the second term is from the repeated doubling and the addition of the other $K_iP$ to the total. Assuming that each block takes approximately the same amount of time, the first processor should not have to wait.

The coordinate system used is affine-Jacobian for the binary NAF algorithm, with the return value in Chudnovsky Jacobian coordinates. The repeated doubling is performed with $2\mathcal{J} \to \mathcal{J}$ and the addition is $\mathcal{J} + \mathcal{J}^c \to \mathcal{J}$.

The average cost for Step 3 is

$$\left(\frac{w-3}{3}\right)(12M + 7S) + (2M + 4S) + \left(w - 2 - \frac{w-3}{3}\right)(4M + 4S) + (M + S),$$

Table 4.6: Left-to-Right (3 Processors) Average Cost ($d = 192$)

| $l$ | Field Cost | $M$-cost | $s_1$ |
|---|---|---|---|
| 3 | I+951.7M+834.0S | 1698.9M | 2 |
| 4 | I+921.0M+822.0S | 1658.6M | 3 |
| 5 | I+921.0M+829.0S | 1664.2M | 4 |
| 6 | I+902.3M+814.0S | 1633.5M | 5 |
| 7 | I+919.7M+830.0S | 1663.7M | 6 |
| 8 | I+905.0M+814.0S | 1636.2M | 7 |
| 9 | I+935.7M+840.0S | 1687.7M | 8 |

with $s_1$ for communication. Since each processor only performs this step at most $\lceil \frac{l-3}{2} \rceil$ times, they should all finish before processor two. The cost of Step 5 for processor 2 is

$$\left(\frac{w-3}{3}\right)(12M+7S)+(2M+4S)+\left(w-2-\frac{w-3}{3}\right)(4M+4S).$$

Step 8 costs $w(4M+4S)$ and Step 9 costs $12M+4S$ with $s_1$ to read from memory. This results in an effective cost for Algorithm 34 of

$$\begin{aligned}&\left(\frac{w-3}{3}\right)(12M+7S)+(2M+4S)+\left(w-2-\frac{w-3}{3}\right)(4M+4S)+\\&\qquad w(l-1)(4M+4S)+(l-1)(12M+4S)+(I+3M+S)\\&=I+\left(\frac{8w}{3}+4lw+12l-23\right)M+(w+4lw+4l-10)S,\end{aligned}$$

with communication overhead $(d-1)\cdot s_1$.

Table 4.6 describes the field costs and $M$-costs of Algorithm 34 with $d$ corresponding to the length of the NIST prime $P192$. The tables for $d$ chosen to correspond with $P224$, $P256$, $P384$ and $P521$ are similar and can be found in Appendix B.

The advantage of this algorithm is that is is relatively fast compared to other parallel implementations with a small number of processors and less communication time. Since the number of blocks $l$ is variable, a trade-off can be made between the effective field cost (the cost on processor 2) and the communication time, providing more flexibility. In the next section, we will examine a version of this algorithm that only needs two processors but uses precomputation.

### 4.3.8 Two-Processor Left-to-Right Parallel with Precomputation

In the previous algorithm, scalar multiplication is split into two parts. The multiplier $k$ is split into blocks $K_0, \ldots, K_{l-1}$ of length $w$. Computation of each $K_iP$ is performed on two processors and the results are sent to another processor which computes $2^wQ + R$. The reason that the previous algorithm required three processors instead of two is that $K_iP$ in general takes more time to compute than $2^wQ + R$. With precomputed values available, it is possible to make sure that only one processor is needed to compute the values $K_i$.

To achieve this, we must look at the width-$v$ NAF of the multiplier. Suppose that $v$-NAF$(k) = (k_{d-1}, \ldots, k_0)$. Let $(k_{d-1} = k_{v_s}, k_{v_{s-1}}, \ldots, k_{v_1})$ be the non-zero digits of $v$-NAF$(k)$. The value $s$ is the number of non-zero digits. Let $w$ be a small positive integer and $r = \lfloor s/w \rfloor$, then define

$$K_0 = (k_{v_s}, \ldots, k_{v_{s-w}}),$$

$$K_1 = (k_{v_{s-w-1}}, \ldots, k_{v_{s-2w}}),$$

$$\vdots$$

$$K_r = (k_{v_{s-rw-1}}, \ldots, k_{v_1}).$$

These values are blocks of consecutive digits of $k$. The values needed for $v$-NAF are precomputed with Algorithm 27. The algorithm will proceed by computing $K_0P$ on processor 1, then doubling $v_{s-w} - v_{s-2w}$ times and adding $K_1P$, $K_2P, \ldots, K_rP$. while processor 2 computes $K_1P, K_2P, \ldots, K_rP$. Each $K_iP$ is computed with $v_{s-iw-1} - v_{s-(i+1)w}$ doublings and $w-1$ additions. The goal is to select $w$ and $v$ appropriately so that processor 2 finishes before processor 1.

In the worst case, the number of zeros between two non-zero values in a $v$-NAF representation of an integer is $v-1$. For now, we will assume that this is always the case. This gives us worst case estimates for our choices of $w$ and $v$, so that the parameter choices work for any input. In this case, we assume $s = d/v$, and $r = d/wv$; we are looking at the average results, so these do not have to be integers.

After precomputation, processor 1 uses $(w-1)A + (w-1)vD$ to compute $K_0P$, then $(d-wv)D$ and $(d/wv)A$. Processor 2 uses $(d/wv)((w-1)A + (w-1)vD)$. In the computation of $K_iP$, the additions are $\mathcal{J} + \mathcal{A} \rightarrow \mathcal{J}$. The doublings are $2\mathcal{J} \rightarrow \mathcal{J}$ and all other additions are $\mathcal{J} + \mathcal{J} \rightarrow \mathcal{J}$. The total field cost for processor 1 is

$$(d-wv)(4M+4S) + \left(\frac{d}{wv}\right)(12M+4S) + \\ ((w-1)v(4M+4S) + (w-1)(8M+3S)).$$

The total field cost for processor 2 is

$$\left(\frac{d}{wv}\right)((w-1)v(4M+4S) + (w-1)(8M+3S)).$$

We need to find the values for $v$ and $w$ such that the field cost for processor 1 is minimal, while being larger than that of processor 2. Keep in mind that a higher value for $v$ translates into a larger precomputation, so we keep $v$ small. Also, lower values of $w$ translate into more additions on processor 1, so we wish to keep $w$ large. Table 4.7 shows the smallest $w$ for a given $v$ and $d$ that satisfies the above equation

Table 4.7: Largest Values of $w$ for a given $v$, $d$

| $d$ | 192 | | | 224 | | | 256 | | | 384 | | | 521 | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $v$ | 3 | 4 | 5 | 3 | 4 | 5 | 3 | 4 | 5 | 3 | 4 | 5 | 3 | 4 | 5 | 6 |
| $w$ | 4 | 5 | 6 | 4 | 5 | 6 | 4 | 5 | 6 | 4 | 5 | 6 | 4 | 5 | 6 | 6 |

for given bit length $d$.

Algorithm 35 is a new algorithm that computes the scalar multiple of a point in the manner described above given the values $K_0, \ldots, K_r$ as above with bit lengths $l_0, \ldots, l_r$.

**Algorithm 35:** Two-Processor Left-to-Right Parallel with Precomputation
**Input:** Affine point $P$, $r+1$ signed binary digit strings $K_0, \ldots, K_r$ of lengths $l_0, \ldots, l_r$
**Output:** Affine point $kP$

```
(1)  precomputation:
(2)      compute $P_i = iP$ for $i \in \{1, 3, \ldots, 2^v - 1\}$ using 2-processor
         precomputation (Algorithm 26).
(3)  **for** processor 1 perform the following:
(4)      $Q_0 \leftarrow K_0 P$ using binary $v$-NAF algorithm
(5)      **for** $j = 1$ **to** $r$
(6)          $Q \leftarrow 2^{l_j} Q$
(7)          wait until $Q_j$ is written to memory, read
(8)          $Q \leftarrow Q + Q_j$
(9)  **for** processor 2 do the following:
(10)     **for** $j = 1$ **to** $r$
(11)         $Q_j \leftarrow K_j P$ using binary $v$-NAF algorithm
(12)         write $Q_j$ to memory
(13) **return** $Q$
```

This algorithm will finish without waiting with appropriate choices for $w$,$v$ and $K_i$. As described above, in the worst case $r \approx \lceil t/(vw) \rceil$. In the average case, the number of zero terms between successive points is $v$, not $v-1$, so we can assume

$r = d/((v+1)w)$. The effective field cost of the non-precomputation portion of Algorithm 32 is

$$(d - w(v+1))D + (d/((v+1)w))A + ((w-1)(v+1)D + (w-1)A)$$

on average.

The repeated doubling in Step 6 is performed with Jacobian coordinates. Each block $K_iP$ is computed with the binary $v$-NAF algorithm by doubling a point in Jacobian coordinates and adding the precomputed affine points. The points $K_iP$ are given in Jacobian coordinates to processor 1 and added.

With these coordinate systems, the effective field cost of Algorithm 35 without the precomputation is

$$\begin{aligned}(d - w(v+1))(4M + 4S) + ((d/((v+1)w))(12M + 4S)+ \\ (w-1)(v+1)(4M+4S) + (w-1)(8M+3S) + (S + 3M + I), \\ = I + \left(4d - 4v + 8w + \frac{12d}{vw+w} - 9\right) M + \left(4d - 4v + 3w + \frac{4d}{vw+w} - 2\right) S,\end{aligned}$$

with communication overhead $d/((w+1)v)s_1$. The generation of the precomputation table has field cost

$$(2M + 4S) + (11M + 3S) + (2^{v-3} - 1)(11M + 3S)$$

$$= (2 + 11 \cdot 2^{w-3})M + (4 + 3 \cdot 2^{w-3})S,$$

and converting the table to affine coordinates costs

$$I + \left(6 \cdot 2^{v-3} - 3\right) M + \left(2^{v-3}\right) S,$$

with a communication overhead of $(3 + 2^{v-3} - 1)s_1$.

Table 4.8: Left-to-Right (2 Processors) Average Cost

| $d$ | $v$ | $w$ | Field Cost | $M$-cost | $s_1$ |
|---|---|---|---|---|---|
| 192 | 3 | 4 | 2I+922.0M+820.0S | 1738.0M | 16 |
| | 4 | 5 | 2I+891.2M+805.7S | 1695.7M | 12 |
| | 5 | 6 | 2I+901.0M+803.3S | 1703.7M | 12 |
| 224 | 3 | 4 | 2I+1074.0M+956.0S | 1998.8M | 18 |
| | 4 | 5 | 2I+1034.5M+938.8S | 1945.6M | 14 |
| | 5 | 6 | 2I+1039.7M+934.9S | 1947.6M | 13 |
| 256 | 3 | 4 | 2I+1226.0M+1092.0S | 2259.6M | 21 |
| | 4 | 5 | 2I+1177.9M+1072.0S | 2195.4M | 15 |
| | 5 | 6 | 2I+1178.3M+1066.4S | 2191.5M | 14 |
| 384 | 3 | 4 | 2I+1834.0M+1636.0S | 3302.8M | 29 |
| | 4 | 5 | 2I+1751.3M+1604.4S | 3194.9M | 20 |
| | 5 | 6 | 2I+1733.0M+1592.7S | 3167.1M | 17 |
| 521 | 3 | 4 | 2I+2484.8M+2218.2S | 4419.4M | 38 |
| | 4 | 5 | 2I+2365.1M+2174.4S | 4264.6M | 26 |
| | 5 | 6 | 2I+2326.7M+2155.9S | 4211.4M | 21 |
| | 6 | 6 | 2I+2365.9M+2159.6S | 4253.6M | 23 |

Table 4.8 describes the field costs and $M$-costs of Algorithm 35 with $d$ corresponding to the NIST primes and $v, w$ determined from Table 4.7. The results of Table 4.8 demonstrate that this algorithm improves upon using a single processor, at the expense of a small number of communications. This is the last algorithm we examine based on the binary algorithm.

### 4.3.9 Parallel Double-Base Representation

Double-base chain multiplication introduced in Section 3.2.6 can be generalized to multiple processors in an analogous way to the $p^{th}$ order binary algorithm using a generalization of a double-base chain. The material in this section is new and is not found in the literature. The key concept is a *double-base $n$-chain*.

**Definition 4.3.1.** *A* double-base $n$-chain *for an integer $k$ is a set of $n$ double-base chains for integers $g_1, \ldots, g_n$ such that $\sum_{i=1}^{n} g_i = k$.*

Double-base $n$-chains can be generated in a number of ways. They can be computed by applying a relaxed version of Algorithm 21 or by taking a double-base chain and partitioning the chain into $n$ sub-chains. There has not been any extensive study of the density of double-base $n$-chains.

Computing a multiple of a point using double-base $n$-chains can be performed in a manner analogous to the $p^{th}$ order binary method. If $g_1, \ldots, g_n$ form a double-base $n$-chain for $k$, then $kP$ can be computed with $n$ processors in parallel by computing $g_iP$ on processor $i$. The point $kP$ is the sum of each of the points $g_iP$. If the integers $g_i$ are chosen correctly, then the technique introduced in Section 4.3.2 can be applied. With this method, the integers $g_1, \ldots, g_n$ are arranged so that $g_1P$ finishes first, then $g_2P$, and so forth with $g_nP$ finishing last. With this arrangement, processor 1 sends $g_1P$ to processor 2, which then computes $g_1P + g_2P$ and sends the result to the next processor. The final processor receives $\sum_{k=1}^{n-1} g_iP$ and computes $kP = g_nP + \sum_{k=1}^{n-1} g_iP$ with one addition.

Algorithm 36 is a new algorithm that computes the scalar multiple of an integer on $N$ processors when given a double-base $N$-chain by computing the scalar multiple of each chain and then adding them together once they are all completed.

```
Algorithm 36: $n$-Chain Scalar Multiplication
Input: Affine point $P$, positive integer $k = \sum_{i=1}^{N} g_i$ with DBNS representations $g_i = \sum_{j=1}^{m_i} g_{i_j} 2^{b_{i_j}} 3^{t_{i_j}}$
Output: Affine point $kP$
(1)   foreach processor $i \in \{1, \ldots, N\}$ do:
(2)       calculate $P_i \leftarrow g_iP$ using Algorithm 22.
(3)       if $N > i > 1$
```

```
(4)            read $P_{i-1}$ from memory
(5)            $P_i \leftarrow P_i + P_{i-1}$
(6)            write $P_i$ to memory
(7)        if $i = N$
(8)            read $P_{i-1}$ from memory
(9)            $P_i \leftarrow P_i + P_{i-1}$
(10)           return $P_n$
```

The cost of Algorithm 36 depends on the chains that are given. One possibility is to take the double-base chain derived from the greedy algorithm and to split it into $N$ different chains. For instance, if the chain has $m_{max}$ terms, take the smallest $m_{max}/N$ terms for the first chain, the next smallest $m_{max}/N$ terms for the second, etc. In this case, if $b_{max}$, $t_{max}$, and $m_{max}$ represent the maximal binary exponent, the maximal ternary exponent and the number of terms, then the average running time is

$$\left\lceil \frac{m_{max}-1}{N} + 1 \right\rceil A + b_{max}D + t_{max}T.$$

Algorithm 36 requires a large number of doublings and triplings and a smaller number of additions. The fastest coordinate system for doubling and tripling is Jacobian. We will therefore use the formulas for $\mathcal{J} + \mathcal{A} \to \mathcal{J}$, $2\mathcal{J} \to \mathcal{J}$, and $3\mathcal{J} \to \mathcal{J}$ to execute Algorithm 36. The field cost of Algorithm 36 is on average

$$\left\lceil \frac{m_{max}-1}{N} \right\rceil (8M+3S) + (2M+4S) + (b_{max}-1)(4M+4S) + \\ t_{max}(9M+5S) + (12M+4S) + (I+3M+S)$$

$$= I + \left(8\left\lceil \frac{m_{max}-1}{N} \right\rceil + 4b_{max} + 9t_{max} + 13\right) M + \\ \left(3\left\lceil \frac{m_{max}-1}{N} \right\rceil + 4b_{max} + 5t_{max} + 5\right) S,$$

and a communication overhead of $s_1$.

Table 4.9: Double-Base $n$-Chain Average Cost ($d = 192$)

| $N$ | Field Cost | $M$-cost | $s_1$ |
|---|---|---|---|
| 2 | I+1085.0M+775.0S | 1785.0M | 1 |
| 3 | I+1029.0M+754.0S | 1712.2M | 1 |
| 4 | I+997.0M+742.0S | 1670.6M | 1 |
| 5 | I+981.0M+736.0S | 1649.8M | 1 |
| 6 | I+973.0M+733.0S | 1639.4M | 1 |
| 7 | I+965.0M+730.0S | 1629.0M | 1 |
| 8 | I+957.0M+727.0S | 1618.6M | 1 |

Table 4.9 describes the field costs and $M$-costs of the variant of Algorithm 36 with no recombination stage. For each $d$ corresponding to a NIST prime, $b_{max}, t_{max}, m_{max}$ are chosen optimally. For multipliers of length 192 the optimal value of $(b_{max}, t_{max})$ is $(116, 48)$, see Table 3.10. The tables for $d$ chosen to correspond with $P224$, $P256$, $P384$ and $P521$ are similar and can be found in Appendix B.

Using the double-base chain results from Table 3.9, some estimates can be made about the effective field cost of Algorithm 36. Note that finding a double-base $n$-chain by splitting a double-base chain is not necessarily the best method of finding one, and that it is almost always possible to find a double-base $n$-chain that is less dense and therefore faster by using a modified greedy algorithm.

### 4.3.10 Parallel Montgomery Ladder

Montgomery [74] introduced a technique for scalar multiplication of points on curves in a form called Montgomery form now referred to as *Montgomery's Ladder*. The technique involves special formulas for addition and doubling that rely on only the $X$ and $Z$ coordinates of a point in projective form. The $Y$ coordinate is derived from the $X$ and $Z$ coordinates after the calculation.

Brier and Joye [15] generalized Montgomery's formulas to any curve with short Weierstraß equation $y^2 = x^3 + ax + b$. If we write $nP$ in projective coordinates as $(X_n : Y_n : Z_n)$, then the following formulas hold:

Addition:

$$\begin{aligned} X_{m+n} &= Z_{m-n}(-4bZ_mZ_n(X_mZ_n + X_nZ_m) + (X_mX_n - aZ_mZ_n)^2), \\ Z_{m+n} &= X_{m-n}(X_mZ_n - X_nZ_m)^2, \end{aligned}$$

Doubling:

$$\begin{aligned} X_{2n} &= (X_n^2 - aZ_n^2)^2 - 8bX_nZ_n^3, \\ Z_{2n} &= 4Z_n(X_n(X_n^2 + aZ_n^4) + bZ_n^3). \end{aligned}$$

The advantage of these formulas is that the $Y$-coordinate of a point is never needed. If $a = -3$, then an addition requires $8M + 2S$ and a doubling requires $5M + 3S$. We want to recover the affine coordinates for $nP = (x_n, y_n)$. Given the $x$ coordinates $x_n, x_{n+1}$ of $nP$ and $(n+1)P$, respectively, $y_n$ is recovered by

$$y_n = \frac{2b + (x_1x_n + a)(x_1 + x_n) - (x_1 - x_n)^2 x_{n+1}}{2y_1}.$$

To find these values, we compute $T = (Z_nZ_{n+1}(2y_1))^{-1}$, then $(Z_n)^{-1} = TZ_{n+1}(2y_1)$ and $Z_{n+1}^{-1} = TZ_n(2y_1)$,$(2y_1)^{-1} = TZ_nZ_{n+1}$. This allows us to compute $x_n = X_nZ_n^{-1}$ and $x_{n+1} = X_{n+1}Z_{n+1}^{-1}$. Therefore, given $x_1$, $(X_n, Z_n)$ and $(X_{n+1}, Z_{n+1})$, computing $y_n$ takes

$$(6M + I) + 4M + S.$$

The left-to-right binary method can be applied with these formulas for doubling and addition. Doubling only requires the point $nP$, but addition requires the points

$nP, mP$ and $(n-m)P$. In Montgomery's ladder, two points are stored: $P_1$ and $P_2$. The point $P_1$ corresponds to the temporary multiple $nP$ and $P_2$ corresponds to $(n+1)P$. Given a multiplier $k = (k_{d-1}, \ldots, k_0)_2$, the left-to-right binary method proceeds by scanning the digits from $k_{d-1}$ down to $k_0$. Starting with $P_1 = P$, if $k_i = 0$, then $P_1$ is doubled, and if $k_i = 1$, then $P_1$ is doubled and $P$ is added. Since $P_2 = P_1 + P$, the double and add step can be computed by adding $P_2$ to $P_1$. Since we are adding $nP$ to $(n+1)P$, we need the point $(n+1-n)P = P$. Notice that $P_2$ needs to be kept as $P_1 + P$ for the new $P_1$. When $k_i = 0$, $P_1$ is set to $2P_1$, so setting $P_2$ to $P_1 + P_2$ results in $P_2 = 2P_1 + P$. When $k_i = 0$, $P_1$ is set to $2P_1 + P$, so setting $P_2$ to $2P_2$ results in $P_2 = (2P_1 + P) + P$.

If this algorithm is implemented with the special Montgomery operations, then $P_1 - P_2$ is equal to $P$ at each step, so that $Z_{m-n} = Z_1$. Since $P$ is given in affine form, $Z_1 = 1$; therefore, the addition requires one less multiplication $(7M + 2S)$.

Algorithm 37 computes a scalar multiple of a point using the operations; it is adapted from Brier and Joye [15]. Processor 1 computes the successive values of $P_1$ and processor 2 computes the successive values of $P_2$.

**Algorithm 37:** Parallel Montgomery Ladder Scalar Multiplication
**Input:** Affine point $P$, positive integer $k$ with binary representation $k = (k_{d-1}, \ldots, k_0)_2$
**Output:** Affine point $kP$

```
(1)  $P_1 \leftarrow P$
(2)  $P_2 \leftarrow 2P$
(3)  for $i = d-2$ to $0$
(4)      if $k_i = 0$
(5)          processor 1:
(6)              $P_1 \leftarrow 2P_1$
(7)              write $P_1$ to shared memory
(8)          processor 2:
```

```
(9)              read $P_1$ from shared memory
(10)             $P_2 \leftarrow P_1 + P_2$
(11)             write $P_2$ to shared memory
(12)       else
(13)          processor 1:
(14)             read $P_2$ from shared memory
(15)             $P_1 \leftarrow P_1 + P_2$
(16)             write $P_1$ to shared memory
(17)          processor 2:
(18)             $P_2 \leftarrow 2P_2$
(19)             write $P_2$ to shared memory
(20)    compute $(x_n, y_n)$ from $P_1, P_2$
(21)    return $P_1$
```

For each step of the loop, the processor performing addition will take the longest time. The effective running time of Algorithm 37 in terms of elliptic curve operations is $(d-1)A$.

Using Montgomery operations for each double and add, the effective field cost of Algorithm 37 is

$$(d-2)(7M+3S)+(5M+3S)+(I+10M+S)$$
$$= I+(7d+1)M+(3d-2)S,$$

with $2(d-1)s_1$ in communication overhead. Table 4.10 describes the field costs and $M$-costs of Algorithm 37 on two processors with $d$ corresponding to the length of the NIST primes.

The main advantage of this algorithm is that it has good side-channel resistance (see Section 5.1.2). However, the algorithm has very fine granularity in that it requires many communications and hence is not practical when $s_1$ is large.

### 4.3.11 Summary

This section compares the running times of the various algorithms described in this chapter. Tables 4.11 and 4.12 describe these results for parameters corresponding to the NIST primes $P192$ and $P521$ in terms of their field costs and their $M$-costs. The tables for $d$ chosen to correspond with $P224$, $P256$ and $P384$ are similar and can be found in Appendix B. The amount of shared memory needed is one Chudnovsky point for every algorithm except 2P L2R, in which $2^{v-2}-1$ Chudnovsky points need to be stored.

If we take a more in depth look at Tables 4.11 and 4.12, we can determine the theoretical speedup of these algorithms in comparison with their serial counterparts. The theoretical speedup is computed as $c_1/c_N$ where $c_1$ is the $M$-cost of the comparable serial algorithm and $c_N$ is the $M$-cost of the algorithm on $N$ processors in question.

Let us first look at Table 4.11. The first case to consider is when the communication time is very low ($s_1 < 2M$). In this case, the two-processor right-to-left algorithm (Algorithm 31) has cost less than $1472.2M + 2M \cdot 63 = 1598.2M$, which is lower than that of any algorithm using 5 or fewer processors and is comparable with

Table 4.10: Parallel Montgomery Ladder Average Cost

| $d$ | Field Cost | $M$-cost | $s_1$ |
|---|---|---|---|
| 192 | I+1345.0M+574.0S | 1884.2M | 382 |
| 224 | I+1569.0M+670.0S | 2185.0M | 446 |
| 256 | I+1793.0M+766.0S | 2485.8M | 510 |
| 384 | I+2689.0M+1150.0S | 3689.0M | 766 |
| 521 | I+3648.0M+1561.0S | 4976.8M | 1040 |

Table 4.11: Parallel Scalar Multiplication ($P192$)

| Processors | Algorithm | Variables | Field Cost | $M$-cost | $s_1$ |
|---|---|---|---|---|---|
| 2 | $p^{th}$ (28) | – | I+1030.0M+864.5S | 1801.6M | 3 |
| | R2L Par. (29) | – | I+1024.0M+861.0S | 1792.8M | 1 |
| | 2P R2L (31) | – | I+777.0M+769.0S | 1472.2M | 63 |
| | 2P Win R2L (32) | $w = 4$ | I+907.0M+795.0S | 1623.0M | 47 |
| | | $w = 5$ | I+1091.0M+851.0S | 1851.8M | 38 |
| | 2P L2R (35) | $v = 3, w = 4$ | 2I+922.0M+820.0S | 1738.0M | 16 |
| | | $v = 4, w = 5$ | 2I+891.2M+805.7S | 1695.7M | 12 |
| | | $v = 5, w = 6$ | 2I+901.0M+803.3S | 1703.7M | 12 |
| | DB2Chain (36) | – | I+1085.0M+775.0S | 1785.0M | 1 |
| | MontLad (37) | – | I+1345.0M+574.0S | 1884.2M | 382 |
| 3 | $p^{th}$ (28) | – | I+958.0M+837.0S | 1707.6M | 5 |
| | R2L Par. (29) | – | I+938.7M+829.0S | 1681.9M | 1 |
| | 3P L2R (34) | $l = 3$ | I+951.7M+834.0S | 1698.9M | 2 |
| | | $l = 4$ | I+921.0M+822.0S | 1658.6M | 3 |
| | | $l = 5$ | I+921.0M+829.0S | 1664.2M | 4 |
| | | $l = 6$ | I+902.3M+814.0S | 1633.5M | 5 |
| | DB3Chain (36) | – | I+1029.0M+754.0S | 1712.2M | 1 |
| 4 | $p^{th}$ (28) | – | I+916.0M+821.2S | 1653.0M | 5 |
| | R2L Par. (29) | – | I+896.0M+813.0S | 1626.4M | 1 |
| | DB4Chain (36) | – | I+997.0M+742.0S | 1670.6M | 1 |
| 5 | $p^{th}$ (28) | – | I+902.8M+815.8S | 1635.4M | 7 |
| | R2L Par. (29) | – | I+884.0M+816.0S | 1616.8M | 1 |
| | DB5Chain (36) | – | I+981.0M+736.0S | 1649.8M | 1 |
| 6 | $p^{th}$ (28) | – | I+886.0M+809.5S | 1613.6M | 7 |
| | R2L Par. (29) | – | I+853.3M+797.0S | 1570.9M | 1 |
| | DB6Chain (36) | – | I+973.0M+733.0S | 1639.4M | 1 |
| 7 | $p^{th}$ (28) | – | I+874.0M+805.0S | 1598.0M | 7 |
| | R2L Par. (29) | – | I+858.7M+809.0S | 1585.9M | 1 |
| | DB7Chain (36) | – | I+965.0M+730.0S | 1629.0M | 1 |
| 8 | $p^{th}$ (28) | – | I+865.0M+801.6S | 1586.3M | 7 |
| | R2L Par. (29) | – | I+832.0M+789.0S | 1543.2M | 1 |
| | R2L Hed. (30) | – | I+777.0M+769.0S | 1472.2M | 1 |
| | DB8Chain (36) | – | I+957.0M+727.0S | 1618.6M | 1 |

Table 4.12: Parallel Scalar Multiplication ($P521$)

| Processors | Algorithm | Variables | Field Cost | $M$-cost | $s_1$ |
|---|---|---|---|---|---|
| 2 | $p^{th}$ (28) | – | I+2784.7M+2345.0S | 4740.7M | 3 |
| | R2L Par. (29) | – | I+2784.0M+2346.0S | 4740.8M | 1 |
| | 2P R2L (31) | – | I+2093.0M+2085.0S | 3841.0M | 173 |
| | 2P Win R2L (32) | $w = 4$ | I+2235.0M+2123.0S | 4013.4M | 130 |
| | | $w = 5$ | I+2411.0M+2171.0S | 4227.8M | 104 |
| | 2P L2R (35) | $v = 3, w = 4$ | 2I+2484.8M+2218.2S | 4419.4M | 38 |
| | | $v = 4, w = 5$ | 2I+2365.1M+2174.4S | 4264.6M | 26 |
| | | $v = 5, w = 6$ | 2I+2326.7M+2155.9S | 4211.4M | 21 |
| | | $v = 6, w = 6$ | 2I+2365.9M+2159.6S | 4253.6M | 23 |
| | DB2Chain (36) | – | I+2967.0M+2042.0S | 4680.6M | 1 |
| | MontLad (37) | – | I+3648.0M+1561.0S | 4976.8M | 1040 |
| 3 | $p^{th}$ (28) | – | I+2566.4M+2262.7S | 4456.6M | 5 |
| | R2L Par. (29) | – | I+2552.0M+2259.0S | 4439.2M | 1 |
| | 3P L2R (34) | $l = 6$ | I+2369.0M+2189.0S | 4200.2M | 5 |
| | | $l = 7$ | I+2361.0M+2193.0S | 4195.4M | 6 |
| | | $l = 9$ | I+2327.7M+2172.0S | 4145.3M | 8 |
| | DB3Chain (36) | – | I+2815.0M+1985.0S | 4483.0M | 1 |
| 4 | $p^{th}$ (28) | – | I+2451.3M+2219.5S | 4306.9M | 5 |
| | R2L Par. (29) | – | I+2445.3M+2224.0S | 4304.5M | 1 |
| | DB4Chain (36) | – | I+2743.0M+1958.0S | 4389.4M | 1 |
| 5 | $p^{th}$ (28) | – | I+2394.3M+2197.6S | 4232.3M | 7 |
| | R2L Par. (29) | – | I+2380.0M+2202.0S | 4221.6M | 1 |
| | DB5Chain (36) | – | I+2695.0M+1940.0S | 4327.0M | 1 |
| 6 | $p^{th}$ (28) | – | I+2348.2M+2180.3S | 4172.5M | 7 |
| | R2L Par. (29) | – | I+2320.0M+2172.0S | 4137.6M | 1 |
| | DB6Chain (36) | – | I+2663.0M+1928.0S | 4285.4M | 1 |
| 7 | $p^{th}$ (28) | – | I+2315.3M+2168.0S | 4129.7M | 7 |
| | R2L Par. (29) | – | I+2300.0M+2172.0S | 4117.6M | 1 |
| | DB7Chain (36) | – | I+2647.0M+1922.0S | 4264.6M | 1 |
| 8 | $p^{th}$ (28) | – | I+2290.7M+2158.8S | 4097.7M | 7 |
| | R2L Par. (29) | – | I+2288.0M+2175.0S | 4108.0M | 1 |
| | R2L Hed. (30) | – | I+2093.0M+2085.0S | 3841.0M | 1 |
| | DB8Chain (36) | – | I+2631.0M+1916.0S | 4243.8M | 1 |

Table 4.13: Scalar Multiplication Speedup ($P192$)

| Processors | Algorithm | $s_1$ | $M$-cost | 1P Alg | Speedup |
|---|---|---|---|---|---|
| 2 | 2P R2L | 0M–2M | 1472.2M–1598.2M | 1946.3M | 1.32–1.22 |
| | | 2M–4M | 1598.2M–1734.2M | 1946.3M | 1.22–1.12 |
| | 2P L2R ($v = 4, w = 5$) | 3M–10M | 1731.7M–1815.7M | 1946.3M | 1.12–1.07 |
| | DB2Chain | >10M | >1795M | 1946.3M | 1.08–1.00 |
| 3 | 3P L2R ($l = 6$) | 0M–12M | 1633.5M–1693.5M | 1946.3M | 1.19–1.15 |
| | R2L Par. | >10M | >1691.9M | 1946.3M | 1.15–1.00 |

the fastest algorithms for 6 or 7 processors. The speedup is at least 1.22 versus the fractional window algorithm (Algorithm 20) in this case. If 8 processors are available, the right-to-left with hedging algorithm (Algorithm 30) is always faster with a speedup of 1.32, as long as the value of $s_1$ is not too large.

The more complicated case is when there is a limited number of processors and $s_1 > 2M$. Table 4.13 describes the resulting cost of the fastest algorithms using two or three processors for the possible ranges of $s_1$. In this case, we assume that there are 3 available spaces of storage for precomputation. With a smaller amount of storage space, the fractional window algorithm to which we are comparing would not be available and the two-processor left-to-right algorithm with $v = 4$ would not be available. The fastest serial scalar multiplication algorithm with no storage space is the double-base chain algorithm (Algorithm 22) with a cost of $1988.9M$.

Table 4.13 indicates that if two processors are available, 2P R2L is the fastest algorithm for $s_1 = 0$ up to around $s_1 \approx 3M$, 2P L2R is faster for $s_1 \approx 3M$ up to $s_1 \approx 10M$ and DB2Chain is faster for $s_1 > 10M$. If three processors are available then 2P R2L is the fastest algorithm for $s_1 = 0$ up to around $s_1 \approx 3M$, 3P L2R is faster for $s_1 \approx 3M$ up to $s_1 \approx 11M$, and R2L Parallel is faster for $s_1 > 10M$.

Table 4.14: Scalar Multiplication Speedup ($P521$)

| Processors | Algorithm | $s_1$ | $M$-cost | 1P Alg | Speedup |
|---|---|---|---|---|---|
| 2 | 2P R2L | 0M–2M | 3841M–4187M | 4940.9M | 1.29–1.18 |
| | | 2M–4M | 4187M–4533M | 4940.9M | 1.18–1.09 |
| | 2P L2R ($v = 4, w = 5$) | 3M–25M | 4274.4M–4631.4M | 4940.9M | 1.16–1.04 |
| | DB2Chain | >25M | >4705.6M | 4940.9M | 1.05–1.00 |
| 3 | 3P L2R ($l = 9$) | 2M–40M | 4157.3M–4461.3M | 4940.9M | 1.19–1.11 |
| | R2L Par. | >40M | >4479.2M | 4940.9M | 1.10–1.00 |

Table 4.12 gives similar results when the communication time is very low ($s_1 < 2M$). In this case, the two-processor right-to-left algorithm (Algorithm 31) still has cost $3841.0M + 173s_1$ which is less than $4187M$, and lower than that of any algorithm using 5 or fewer processors and is comparable with the fastest algorithms for 6 or 7 processors. The speedup is at least 1.18 versus the fractional window algorithm (Algorithm 20) in this case. If 8 processors are available, the right-to-left with hedging algorithm (Algorithm 30) is always faster with a speedup of 1.29.

Again, the complicated case is when there is a limited number of processors and $s_1 > 2M$. Table 4.14 describes the resulting cost of the fastest algorithms using two or three processors for the possible ranges of $s_1$. In this case, we assume that there are 7 available spaces of storage for precomputation. With a smaller amount of storage space, the fractional window algorithm we are comparing to would not be available and the two processor right-to-left algorithm with $v = 4$ would not be available. The fastest serial scalar multiplication algorithm with no storage space is the double-base chain algorithm (Algorithm 22) with a cost of $5241.9M$.

This table indicates that if two processors are available, 2P R2L is the fastest algorithm for $s_1 = 0$ up to around $s_1 \approx 3M$, 2P L2R is faster for $s_1 \approx 3M$ up to

$s_1 \approx 25M$, and DB2Chain is faster for $s_1 > 25M$. If three processors are available, then 2P R2L is the fastest algorithm for $s_1 = 0$ up to around $s_1 \approx 3M$, 3P L2R is faster for $s_1 > 3M$ up to $s_1 \approx 40M$, and R2L Parallel is faster for $s_1 > 40M$.

The consistency of the results of these two tables suggest that the following general conclusions hold for all the NIST primes:

- 2P R2L is the fastest algorithm for fewer than 8 processors when $s_1$ is very small.
- R2L Hed. is the fastest algorithm for 8 processors.
- 2P L2R and 3P L2R are the fastest algorithms for 2 and 3 processors, respectively, when $s_1$ is small.
- DB2Chain and R2L Par. are the fastest algorithms for 2 and 3 processors respectively when $s_1$ is large.
- R2L Par. is the fastest algorithm for 3 to 7 processors when $s_1$ is not very small.
- Using more than 8 processors does not offer any advantage over using 8 processors.

It must be noted that these estimates are based on very specific assumptions. Implementations of the algorithms are needed to determine the practical speedup that is obtained.

The results of this chapter suggest that a speedup can be obtained as long as the communication time $s_1$ is small. The maximum speedup is around 1.30, and this can be achieved with at most 8 processors. If speed of computation is a high priority and multiple processors are available, it may be useful to implement one of these parallel algorithms.

# Chapter 5

# Conclusion

In this thesis, we presented results on the comparative speed of several serial and parallel scalar multiplication algorithms for elliptic curves over prime fields. A metric called the $M$-cost was developed and used to compare the running time of different algorithms using estimates of $S = (4/5)M$ and $I = 80$ for the relative speed of prime field squaring, multiplication and inversion. These assumptions are standard (Okeya and Sakurai [80] and Lim and Hwang [65]) and hold for most software implementations of prime field arithmetic.

Using this metric, we examined elliptic curve scalar multiplication in three situations: unknown point, known point and known multiplier scalar multiplication. For all three scenarios, we performed a detailed analysis of the performance of scalar multiplication algorithms. In the unknown point case, we found that the fastest algorithm requiring precomputation and storage space is the fractional window algorithm (Algorithm 20), and using the new tripling formulas, the fastest algorithm not requiring storage space is the double-base chain algorithm (Algorithm 22). In the known point case, we examined fixed-base windowing and the comb method and found that the comb method is faster for a given amount of storage space except for a small range of parameters where fixed-base windowing is faster. The results for unknown point and known point scalar multiplication were shown to hold for scalars with lengths corresponding with all five NIST primes. In the known multiplier case, there are no definitive results in the literature on computing optimal addition-subtraction chains, so a full analysis could not be made.

In Chapter 4, we presented several algorithms for unknown point scalar multiplication using multiple processors. We used a computation model in which there are a small number of processors that each have their own memory and access to a shared memory bank. Combining this model with the $M$-cost metric, we were able to determine the fastest algorithms given different memory access costs. Using two processors, the two processor right-to-left algorithm (Algorithm 31) was shown fastest when memory access is fast, two processor left-to-right (Algorithm 35) is fastest when memory access is slower, and double-base $n$-chain (Algorithm 36) is fastest when writing to memory is very slow. For three processors, three processor right-to-left (Algorithm 34) is fastest when memory access is fast, and right-to-left parallel is fastest otherwise. For 4 to 7 processors, the right-to-left parallel algorithm (Algorithm 29) is fastest and for 8 processors, right-to-left parallel with hedging (Algorithm 30) is fastest. The results of this chapter show that a speedup can be obtained for scalar multiplication by using multiple processors, but the speedup is not linear in the number of processors. Instead, it appears to approach an upper bound as the number of processors increases.

These results are significant because they show that there are high level parallel algorithms that can increase the speed of scalar multiplication. Also, using the modified tripling formulas, the double-base chain algorithm becomes the most efficient serial algorithm for scalar multiplication that does not need extra storage space. This thesis demonstrates that although much progress has been made, there is still more work to be done in the area of fast elliptic curve arithmetic.

## 5.1 Further Work

In this section, we will examine several topics that are relevant to elliptic curve arithmetic and ECC but were not examined in this thesis.

### 5.1.1 Simultaneous Multiplication

One operation that we did not discuss in this thesis is simultaneous scalar multiplication. Simultaneous multiplication is a specialized algorithm for computing $aP + bQ$ that is used in signature verification, for example ECDSA [46]. We now briefly mention some of the standard techniques for performing simultaneous multiplication and discuss further avenues of study.

In some elliptic curve signature schemes, the value $aP + bQ$ needs to be computed, where $a, b \in \mathbb{N}$ and $P, Q$ are points on an elliptic curve. One method that was suggested by ElGamal [32] is what is often referred to as Shamir's trick. The idea is similar to that of the left-to-right binary algorithm. NAF representations $(a_{k-1}, \ldots, a_0), (b_{k-1}, \ldots, a_0)$ of $a$ and $b$ are taken (padding $a$ or $b$ with zeros if necessary), and the points $P + Q, P - Q, -P - Q$ are precomputed. The algorithm begins by examining the highest order digits of $a$ and $b$, and setting an accumulator $R = a_{k-1}P + b_{k-1}Q$. The algorithm proceeds by doubling the accumulator and adding $a_iP + b_iQ$ for each $i$ from $k - 2$ down to 0.

We now discuss the current methods for encoding the digits of $P$ and $Q$ to increase the efficiency of Shamir's trick. The joint sparse form is an optimal signed encoding that reduces the average number of digits in which $(a_i, b_i) \neq (0, 0)$. Interleaving is a method that allows window NAF and other techniques to be combined by performing the doublings jointly.

## Joint Sparse Form

For simplicity, we write the NAF representations of $a$ and $b$ jointly in a $2\times n$ coefficient matrix:

$$\begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} a_{k-1} & \dots & a_0 \\ b_{k-1} & \dots & b_0 \end{pmatrix}.$$

The non-zero columns of this matrix correspond to additions when computing the joint scalar multiplication. The joint sparse form is a representation of the two integers chosen to minimize the number of non-zero columns. The following definition was introduced by Solinas [90].

**Definition 5.1.1.** *The* joint sparse form *(or* JSF*) of the $l$-bit integers $a$ and $b$ is a representation of the form*

$$\begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} a_{k-1} & \dots & a_0 \\ b_{k-1} & \dots & b_0 \end{pmatrix}$$

*such that $a_i, b_i \in \{0, \pm 1\}$ and*

1. *At least one of any three consecutive columns is zero.*
2. *Consecutive terms in a row do not have opposite signs.*
3. *If $a_{j+1}a_j \neq 0$ then $b_{j+1} \neq 0$ and $b_j = 0$.*

Note that the JSF of two given integers is unique and the average number of non-zero columns is $k/2$. There is also a variant on the JSF called *simple JSF* that is due to Grabner *et al.* [40] that is easier to compute in practice.

### Interleaving

Interleaving is a method for simultaneous multiplication in which window methods are used for computing additions and doublings jointly. It is due to Gallant *et al.* [37] and Möller [69].

Suppose $a$ and $b$ have base 2 representations

$$a = (a_{k-1}, \dots, a_0)_2, \quad b = (b_{l-1}, \dots, b_0)_2$$

where the coefficients $a_i \in A, b_i \in B$ and $A, B$ are specific digit sets. These representations can be taken from window NAF, fractional window or other representations. Interleaving involves precomputing all the multiples of $P$ in $A$ and all the multiples of $Q$ in $B$, and then applying Shamir's trick. Interleaving is discussed in Hankerson *et al.* [43, Sec. 3.3.3].

### Further Research

It would be useful to have a full analysis of the JSF and interleaving algorithms for elliptic curve scalar multiplication in order to do a full analysis signature verification by comparing different multiple point multiplication algorithms.

There are possible generalizations of the JSF that have yet to be fully explored, including window and fractional window analogues. There is also work to be done to create efficient parallel versions of multiple point multiplication. Another potentially relevant issue specific to ECDSA [46] is finding algorithms for computing $aP + bQ$ when one or both of the points are known.

### 5.1.2 Side Channel Attacks

Another point of interest that was briefly mentioned but not discussed in detail is the issue of side channel attacks. This has become a popular area of research in recent years. Implementations of cryptographic algorithms can be susceptible to leaking data through what are called side channels.

An overview of side-channel attacks on curves and their countermeasures can be found in a report by Avanzi [5]. We briefly discuss the different types of side channel attacks and describe a number of available mathematical solutions.

#### Simple Power Analysis

Power analysis is a technique in which the power consumption of a cryptographic device is measured while it is in use in order to determine the secret key, first introduced by Kocher *et al.* [56]. With statistical tools, the information obtained from the raw power consumption graph is used to determine which operations were performed by the device. A simple power analysis attack involves analyzing the power trace of one cryptographic operation.

Since different operations (multiplication, addition, cyclic shift, etc.) require different amounts of power, an attacker may be able to derive information about the operation performed. For example, in binary scalar multiplication, a sequence of additions and doublings is performed based on the binary representation of the scalar multiple. If an attacker can distinguish the power curve of an addition from a doubling, he can determine the sequence of operations performed and hence the binary representation of the secret key.

One possible countermeasure to this attack is the introduction of dummy elliptic curve operations. The simplest version of this is the *double-and-always-add* algorithm. This is similar to the double-and-add algorithm, except that when a zero digit occurs, a dummy addition is performed. This causes the power curve to look like an alternating sequence of doubles and additions. Lange mentions a method of Giessmann [6, page 689] for introducing dummy operations to the NAF method that does not introduce as many additions.

There are related methods that do not require dummy operations but present the same sequence of elliptic curve operations to a side channel attacker. The Montgomery ladder method presented in Section 4.3.10 is one of these methods. Möller [70] also presented some ideas based on windowing methods for which there is no digit 0 in the digit set. Okeya and Tagagi [82] achieve expansions without the digit 0 in a window NAF type expansion.

Another countermeasure is the introduction of dummy *finite field* operations. Chevallier-Mames *et al.* [17] introduced the concept of *side-channel atomicity.* In side channel atomicity, instead of making the sequence of elliptic curve operations uniform, each operation is split up into a number of identical blocks. To a side-channel attacker, a sequence of operations made up of atomic blocks looks uniform.

Brier *et al.* [15] presented a set of unified formulas for addition and doubling. Using these formulas, addition and doubling are indistinguishable in terms of simple power analysis. Joye and Quisquater [47] examined similar unified formulas for Hessian curves, and Liardet and Smart [62] examined such formulas in the Jacobi model.

The simplest and least costly mathematical defence against simple power analysis seems to be that of side-channel atomicity, since it only introduces a few field additions and no multiplications to the formulas for Jacobian, Chudnovsky Jacobian, and mixed affine-Jacobian operations.

**Differential Side Channel Attacks**

Differential power analysis was also introduced by Kocher *et al.* [56] and involves taking a large number of power traces of a cryptographic device in order to determine information about the secret key.

To defend against differential power analysis, the representation of the scalar must be varied from one execution to another. This makes the information obtained from each operation unrelated, diffusing the effect of differential attacks to obtain information.

Coron's first countermeasure [22] is a method for varying the scalar as long as the group order is known. If $l$ is the group order and $k_1, k_2$ are random integers, then $kP = (k + k_1 + k_2 l)P - k_1 P$. The performance penalty is linear in the bit-lengths of $k_1$ and $k_2$.

Coron's second countermeasure [22] is called *blinding of group elements*. Let $k$ be the hidden exponent, and store $(Q, kQ)$ for a set of points $Q$. To compute $kP$, compute $k(Q + P) - kQ$ for some stored $Q$. Changing $Q$ each time ensures that the computation of $kP$ is hidden. The cost is two additions.

Projective randomization is another technique by which the base point is given in Jacobian (or projective) coordinates $(X : Y : Z)$ and replaced with an equivalent element $(\lambda^2 X : \lambda^3 Y : \lambda Z)$ (or $(\lambda X : \lambda Y : \lambda Z)$, respectively) for some random $\lambda$. See Joye and Tymen [23].

Another option is the randomization of the curve equation. In Section 2.3.1, we discussed curve isomorphism. Joye and Tymen [23] introduced a way to work under a different isomorphic curve equation for each execution of the curve in order to counter differential analysis.

It should be noted that differential side channel analysis requires a large number of power traces of scalar multiplications by the same exponent in order to be effective. In many cryptographic primitives (such as ECDH [96]), the private key is ephemeral and used only once. In these cases, differential side channel analysis is not effective.

#### Other attacks

Simple and differential power analysis attacks are the most common attacks referred to in the literature, but there are many more side-channel attacks. See Cohen *et al.* [6, Ch. 29] for a details about Goubin type attacks, higher order differential attacks, timing attacks and fault attacks; some of the mathematical countermeasures are also discussed.

#### Further Work

There are many possible countermeasures to side channel attacks. Choosing which countermeasure is useful is determined by the implementation, by the protocol, and by the required security parameters. Integrating the computational cost of the side channel protection measures with the cost of the algorithms presented in this thesis is an avenue for further research.

### 5.1.3 Binary Curves

Elliptic curves over prime fields are often preferred to elliptic curves over binary fields for software implementations. The reason is that prime field arithmetic is often faster than binary field arithmetic in software implementations (see Brown *et al.* [16] versus Hankerson *et al.* [42]). Binary fields are still in use, however. A useful endeavour would be to revisit the questions of this thesis with a focus on binary field arithmetic.

One major difference to the prime field scenario is the ratios of the time it takes for finite field operations in binary field arithmetic. The relative speed of multiplication, squaring and inversion is different in binary fields: rather than $S = .8M$, and $I = 80M$, in binary fields the ratios are closer to $S = .1M$, and $I = 8M$. For these ratios, squaring cost can be neglected.

Chapter 2 provided a general introduction to elliptic curves that also applies to binary curves up to Section 2.3.3. The general group law formulas (2.10) and (2.11) can be customized for elliptic curves over binary fields. Projective coordinates (López-Dahab [66]) are also useful for binary curves; however, they are not always preferable because affine coordinates are competitive with the low cost of inversion.

Most of the algorithms in Chapter 3 are generic in the sense that they do not require the elliptic curve to be defined over a prime field. The choices for coordinate systems and the field cost analysis could be redone for binary fields. The same could be done for many of the algorithms from Chapter 4.

### 5.1.4 Parallelization

The algorithms in Chapter 4 are essentially generic, and rely on properties about the speed of doubling and addition. Most of this framework could possibly be improved in some ways. For example, the parallel precomputation algorithm on two, four or eight processors could be combined with a version of the right-to-left parallel algorithm modified to use window NAF.

A possible two-processor parallel system could be derived from a radix-2 system for integers, so that the higher order digits are small and grow progressively larger towards the lower order digits. The first processor would do the work of the precomputation by computing multiples of the base point while the other processor would compute the scalar multiple of the point using the left-to-right binary algorithm, reading precomputed points from shared memory as needed. Since the digits are initially small and grow progressively larger, the points will have been precomputed by the time they are needed to add to the total.

Chapter 4 only dealt with algorithms for unknown point scalar multiplication. There is potential to parallelize known point methods such as the comb method.

We have not dealt with parallelizing elliptic curve scalar multiplication on a lower level although there have been some successful attempts to do this in the literature. Mishra *et al.* [68] introduced a method for pipelining the elliptic curve operations on two processors. Aoki *et al.* [3] introduced algorithms to reduce the computation time of certain elliptic curve operations using SIMD operations. One avenue of research is to combine low level parallelizations with the high level parallel algorithms from Chapter 4.

Another aspect that we have not examined is the implementation of these parallel algorithms. Our analysis of the parallel algorithms in Chapter 4 is based on many assumptions about the parallel model. It would be interesting to implement the algorithms from Chapter 4 in order to see if estimates for $s_1$ can be obtained and how well the theoretical assumptions bear out in reality.

# Bibliography

[1] L. M. Adleman and H. W. Lenstra, *Finding irreducible polynomials over finite fields*, STOC '86: Proceedings of the eighteenth annual ACM symposium on Theory of computing (New York, NY, USA), ACM Press, 1986, pp. 350–355.

[2] G. M. Amdahl, *Validity of the single processor approach to achieving large scale computing capabilities*, Proc. AFIPS, vol. 30, 1967, pp. 483–485.

[3] K. Aoki, F. Hoshino, T. Kobayashi, and H. Oguro, *Elliptic curve arithmetic using SIMD*, ISC '01: Proceedings of the 4th International Conference on Information Security (London, UK), Springer-Verlag, 2001, pp. 235–247.

[4] M. Artin, *Algebra*, Prentice Hall, 1991.

[5] R. Avanzi, *Side channel attacks on implementations of curve-based cryptographic primitives*, preprint, http://eprint.iacr.org/2005/017/, 2005.

[6] R. M. Avanzi, H. Cohen, C. Doche, G. Frey, T. Lange, K. Nguyen, and F. Vercauteren, *Handbook of Elliptic and Hyperelliptic Curve Cryptography*, Chapman and Hall/CRC, 2006.

[7] D. Bailey and C. Paar, *Optimal extension fields for fast arithmetic in public-key algorithms*, Advances in Cryptology – CRYPTO '98, Lecture Notes in Computer Science, vol. 1462, 1998, pp. 472–485.

[8] P. Barrett, *Implementing the Rivest, Shamir and Adleman public key encryption algorithm on a standard digital signal processor*, Proceedings on Advances in

Cryptology—CRYPTO '86 (London, UK), Lecture Notes in Computer Science, vol. 263, Springer-Verlag, 1987, pp. 311–323.

[9] M. Bellare and P. Rogaway, *Minimizing the use of random oracles in authenticated encryption schemes*, Information and Communications Security '97, vol. 1334, Lecture Notes in Computer Science, no. 188, 1997, pp. 1–16.

[10] F. Bergeron, J. Berstel, and S. Brlek, *Efficient computation of addition chains*, J. Théor. Nombres Bordeaux **6** (1994), 21–38.

[11] F. Bergeron, J. Berstel, S. Brlek, and C. Duboc, *Addition chains using continued fractions*, J. Algorithms **10** (1989), no. 3, 403–412.

[12] V. Berthé and L. Imbert, *On converting numbers to the double base number system*, Advanced Signal Processing Algorithms, Architecture and Implementations XIV, Proceedings of SPIE, vol. 5559, 2004, pp. 70–78.

[13] W. Bosma, *Signed bits and fast exponentiation*, J. Théor. Nombres Bordeaux **13** (2001), 27–41.

[14] E. F. Brickell, D. M. Gordon, K. S. McCurley, and D. B. Wilson, *Fast exponentiation with precomputation*, Advances in Cryptology - EUROCRYPT '92, Lecture Notes in Computer Science, vol. 657, 1992, pp. 200–217.

[15] É. Brier, I. Déchène, and M. Joye, *Unified point addition formulae for elliptic curve cryptosystems*, Embedded Cryptographic Hardware: Methodologies and Architectures (2004), 247–256.

[16] M. Brown, D. Hankerson, J. López, and A. Menezes, *Software implementation of the NIST elliptic curves over finite fields*, Lecture Notes in Computer Science, vol. 1965, 2001, pp. 250–265.

[17] B. Chevallier-Mames, M. Ciet, and M. Joye, *Low-cost solutions for preventing simple side-channel analysis: Side-channel atomicity*, IEEE Trans. on Computers **53** (2004), 760–768.

[18] D. Chudnovsky and G. Chudnovsky, *Sequences of numbers generated by addition in formal groups and new primality and factoring tests*, Advances in Applied Mathematics **6** (1987), 385–434.

[19] M. Ciet, M. Joye, K. Lauter, and P. Montgomery, *Trading inversions for multiplications in elliptic curve cryptography*, Designs, Codes and Cryptography **39** (2006), no. 2, 189 – 206.

[20] H. Cohen, A Miyaji, and T. Ono, *Efficient elliptic curve exponentiation using mixed coordinates*, Advances in Cryptology – ASIACRYPT '98: International Conference on the Theory and Application of Cryptology, Lecture Notes in Computer Science, vol. 1514, January 1998, p. 51.

[21] P. G. Comba, *Exponentiation cryptosystems on the IBM PC*, IBM Syst. J. **29** (1990), no. 4, 526–538.

[22] J.-S. Coron, *Resistance against differential power analysis for elliptic curve cryptosystems*, Cryptographic Hardware and Embedded Systems – CHES '99, Lecture Notes In Computer Science, vol. 1717, 1999, pp. 392–402.

[23] ———, *Protections against differential analysis for elliptic curve cryptography – An algebraic approach*, Cryptographic Hardware and Embedded Systems – CHES '99, Lecture Notes In Computer Science, vol. 2162, 2002, pp. 377–390.

[24] R. Crandall, *Method and apparatus for public key exchange in a cryptographic system*, United States Patent 5, 159, 632, Oct. 27th 1992.

[25] R. Crandall and C. Pomerance, *Prime Numbers, A Computational Perspective*, Springer-Verlag, Berlin, 2001.

[26] J.-F. Dhem, *Design of an efficient public key cryptographic library for RISC-based smart cards*, Ph.D. thesis, Université Catholique de Louvain-la-Neuve, Belgique, 1998.

[27] W. Diffie and M. Hellman, *New directions in cryptography*, IEEE Transactions on Information Theory **22** (1976), 644–654.

[28] V. Dimitrov, L. Imbert, and P. K. Mishra, *Efficient and secure elliptic curve point multiplication using double-base chains*, Advances in Cryptology – ASIACRYPT '05, Lecture Notes In Computer Science, vol. 3788, 2005, pp. 59–78.

[29] V. Dimitrov, G. Jullien, and W. Miller, *An algorithm for modular exponentiation*, Information Processing Letters **66** (1998), no. 3, 155–159.

[30] ———, *Theory and applications of the double base number system*, IEEE Transactions on Computers **48** (1999), no. 10, 1098–1106.

[31] K. Eisenträger, K. Lauter, and P. Montgomery, *Fast elliptic curve arithmetic and improved weil pairing evaluation*, Topics in Cryptography–CT-RSA 2003,

vol. 2612, Lecture Notes in Computer Science, no. 230, 2003, pp. 343–354.

[32] T. ElGamal, *A public key cryptosystem and a signature scheme based on discrete logarithms*, IEEE Trans. Inform. Theory **IT-31** (1985), 469–472.

[33] E. Fujisaki and T. Okamoto, *Secure integration of asymmetric and symmetric encryption schemes*, Advances in Cryptology–CRYPTO '99, vol. 1666, Lecture Notes in Computer Science, no. 480, 1999, pp. 537–554.

[34] W. Fulton, *Algebraic Curves*, Benjamin, New York, 1969.

[35] S. D. Galbraith, *Supersingular curves in cryptography*, Advances in Cryptology - ASIACRYPT 2001, Lecture Notes In Computer Science, vol. 2248, 2001, p. 495.

[36] S. D. Galbraith and J. McKee, *The probability that the number of points on an elliptic curve over a finite field is a prime*, J. London Math. Soc. **63** (2000), no. 3, 671–684.

[37] R. Gallant, R. Imbert, and S. Vanstone, *Faster point multiplication on elliptic curves with efficient endomorphisms*, Advances in Cryptology – CRYPTO 2001, vol. 2139, Lecture Notes in Computer Science, no. 241, 2001, pp. 190–200.

[38] J. M. G. Garcia and R. M. Garcia, *Parallel algorithm for multiplication on elliptic curves*, Cryptology ePrint Archive, Report 2002/179, http://eprint.iacr.org.

[39] D. M. Gordon, *A survey of fast exponentiation method*, Journal of Algorithms **27** (1998), 129–146.

[40] P. Grabner, C. Heuberger, and H. Prodinger, *Distribution results for low weight binary representations for pairs of integers*, Theorer. Comput. Sci. **319** (2004), 307–331.

[41] T. Granlund, *The GNU multiple precision arithmetic library*, http://www.swox.com/gmp, 2004, version 4.1.4.

[42] D. Hankerson and A. Menezes J. L. Hernandez, *Software implementation of elliptic curve cryptography over binary fields*, Lecture Notes in Computer Science, vol. 1965, 2001.

[43] D. Hankerson, A. Menezes, and S. Vanstone, *Guide to Elliptic Curve Cryptography*, Springer-Verlag New York, Inc., 2004.

[44] Y. Hitchcock, E. Dawson, A. Clark, and P. Montague, *Implementing an efficient elliptic curve cryptosystem over GF(p) on a smart card*, Proc. of 10th Computational Techniques and Applications Conference CTAC-2001, ANZIAM J., vol. 44, 2003, pp. C354–C377.

[45] K. Ireland and M. Rosen, *A Classical Introduction to Modern Number Theory*, 2 ed., Springer-Verlag, 1990.

[46] D. Johnson, A. Menezes, and S. Vanstone, *The elliptic curve digital signature algorithm (ECDSA)*, International Journal of Information Security **1** (2001), 36–63.

[47] M. Joye and J.-J. Quisquater, *Hessian elliptic curves and side-channel attacks*, Cryptographic Hardware and Embedded Systems - CHES 2001, Lecture Notes

in Computer Science, vol. 2162, 2001, p. 402.

[48] D. Jungnickel, *Finite Fields*, B.I.-Wissenschaftverlag, 1993.

[49] A. Karatsuba, *The complexity of computations*, Trudy Mat. Inst. Steklov. **211** (1995), 186–202.

[50] A. Karatsuba and Y. Ofman, *Multiplication of multiplace numbers on automata*, Dokl. Acad. Nauk SSSR **145** (1962), no. 2, 293–294.

[51] R. M. Karp and V. Ramachandran, *Handbook of theoretical computer science (vol. a): algorithms and complexity*, ch. Parallel algorithms for shared-memory machines, MIT Press, Cambridge, MA, 1991.

[52] D. Knuth, *The Art of Computer Programming – Seminumerical Algorithms*, 3 ed., Addison-Wesley, 1998.

[53] N. Koblitz, *Elliptic curve cryptosystems*, Mathematics of Computation **48** (1987), no. 177, 203–209.

[54] ______, *Algebraic Aspects of Cryptography*, Springer-Verlag, Berlin, 1998.

[55] C. K. Ko c, T. Acar, and B. S. Kalinski, *Analyzing and comparing Montgomery multiplication algorithms*, IEEE Micro **16** (1996), no. 3, 26–33.

[56] P. Kocher, J. Jaffe, and B. Jun, *Differential power analysis*, Advances in Cryptology – CRYPTO '99, Lecture Noted In Computer Science, vol. 1666, 1999, pp. 388–397.

[57] N. Kunihiro and H. Yamamoto, *Window and extended window methods for addition chain and addition-subtraction chain*, IEICE Trans. Fundamentals **E81-A** (1998), no. 1902, 72–81.

[58] S. Lang, *Algebra*, 3 ed., Graduate Texts In Mathematics, vol. 211, Springer-Verlag Berlin, 2001.

[59] L. Law, A. Menezes, M. Qu, J. Solinas, and S. Vanstone, *An efficient protocol for authenticated key agreement*, Designs, Codes and Cryptography **28** (2003), 119–134.

[60] D. Lehmer, *Euclid's algorithm for large numbers*, American Mathematical Monthly **45** (1938), 227–233.

[61] A. K. Lenstra, H. W. Lenstra Jr., M. S. Manasse, and J. M. Pollard, *The number field sieve*, STOC '90: Proceedings of the twenty-second annual ACM symposium on theory of computing (New York, NY, USA), ACM Press, 1990, pp. 564–572.

[62] P.-Y. Liardet and N. P. Smart, *Preventing SPA/DPA in ECC systems using the Jacobi form*, Cryptographic Hardware and Embedded Systems - CHES 2001, Lecture Notes in Computer Science, vol. 2162, 2001, p. 391.

[63] R. Lidl and H. Neiderreiter, *Finite Fields*, 2 ed., Cambridge University Press, 1997.

[64] C. Lim and P. Lee, *More flexible exponentiation with precomputation*, CRYPTO '94, Lecture Notes in Computer Science, vol. 839, 1994, pp. 95–107.

[65] C. H. Lim and H. S. Hwang, *Fast implementation of elliptic curve arithmetic in GF*($p^m$), Public Key Cryptography (PKC2000), Lecture Notes In Computer Science, vol. 1751, 2000, pp. 405–421.

[66] J. Lopez and R. Dahab, *Improved algorithms for elliptic curve arithmetic in GF(*$2^n$*)*, Selected Areas in Cryptography (SAC98), 1998, pp. 201–212.

[67] V. Miller, *Use of elliptic curves in cryptography*, Advances in Cryptology-CRYPTO '85, Lecture Notes in Computer Science, vol. 218, 1985, pp. 203–209.

[68] P. K. Mishra, *Pipelined computation of scalar multiplication in elliptic curve cryptosystems*, Cryptographic Hardware and Embedded Systems - CHES 2004, Lecture Notes in Computer Science, vol. 3156, January 2004, pp. 328–342.

[69] B. Möller, *Algorithms for multi-exponentiation*, Selected Areas in Cryptography – SAC 2001, vol. 2259, Lecture Notes in Computer Science, no. 468, 2001, pp. 165–180.

[70] B. Möller, *Securing elliptic curve point multiplication against side-channel attacks*, Information Security - ISC 2001, Lecture Notes In Computer Science, vol. 2200, 2001, pp. 324–333.

[71] ———, *Improved techniques for fast exponentiation*, Information Security and Cryptology - ISICS 2002, Lecture Notes in Computer Science, vol. 2587, 2002, pp. 282–290.

[72] ———, *Parallelizable elliptic curve point multiplication method with resistance against side-channel attacks*, Information Security - ISC 2002, Lecture Notes in

Computer Science, vol. 2433, 2002, pp. 402–413.

[73] P. Montgomery, *Modular multiplication without trial division*, Mathematics of Computation **44** (1985), no. 170, 519–521.

[74] ———, *Modular multiplication without trial division*, Mathematics of Computation **48** (1987), 243–264.

[75] ———, *Speeding the Pollard and elliptic curve methods of factorization*, Mathematics of Computation **48** (1987), no. 177, 243–264.

[76] J. A. Muir and D. R. Stinson, *Minimality and other properties of the width-w nonadjacent form*, Mathematics of Computation **75** (2006), 369–384.

[77] N. Nedjah and L. de Macedo Mourelle, *Minimal addition chain for modular exponentiation using genetic algorithms*, Proceedings of the 2002 Industrial and Engineering, Applications of Artificial Intelligence and Expert Systems Conference, Lecture Notes In Computer Science, vol. 2358, 2002, pp. 88–89.

[78] National Institute of Standard and Technology, *Recommended elliptic curves for federal government use*, 1999, http://www.csrc.nist.gov/encryption/.

[79] K. Okeya, H. Kurumatani, and K. Sakurai, *Elliptic curves with the Montgomery-form and their cryptographic applications*, Public Key Cryptography: Third International Workshop on Practise and Theory in Public Key Cryptosystems, PKC 2000, Lecture Notes in Computer Science, vol. 1751, 2000, pp. 238–257.

[80] K. Okeya and K. Sakurai, *Efficient elliptic curve cryptosystems from a scalar multiplication algorithm with recovery of the y-coordinate on a Montgomery-*

*form elliptic curve*, Proceedings of the Third International Workshop on Cryptographic Hardware and Embedded Systems – CHES '01, vol. 2020, Lecture Notes In Computer Science, no. 338, 2001, pp. 126–141.

[81] K. Okeya, K. Schmidt-Samoa, C. Spahn, and T. Takagi, *Signed binary representations revisited*, Advances in Cryptology - CRYPTO 2004, Lecture Notes in Computer Science, vol. 3152, 2004, pp. 123–139.

[82] K. Okeya and T. Tagagi, *The width-w NAF provides small memory and fast elliptic scalar multiplication secure against side channel attacks*, Topics In Cryptology – CT-RSA 2003, Lecture Notes In Computer Science, vol. 2612, 2003, pp. 328–342.

[83] S. C. Pohlig and M. E. Hellman, *An improved algorithm for computing logarithms over GF($p$) and its cryptographic significance*, IEEE Trans. Inf. Theory **IT-24** (1978), 106–110.

[84] J. Pollard, *Monte carlo methods for index computation (mod p)*, Mathematics of Computation **32** (1978), 918–924.

[85] J. Protic, M. Tomasevic, and V. Milutinovic, *Distributed shared memory: Concepts and systems*, Parallel and Distributed Technology: Systems and Applications, IEEE **4** (1996), no. 2, 63–71.

[86] Victor Shoup, *Fast construction of irreducible polynomials over finite fields*, SODA '93: Proceedings of the fourth annual ACM-SIAM Symposium on Discrete algorithms (Philadelphia, PA, USA), Society for Industrial and Applied Mathematics, 1993, pp. 484–492.

[87] I. E. Shparlinski, *Finite Fields: Theory and Computation*, Kluwer Academic Publishers, 1999.

[88] J. H. Silverman, *The Arithmetic of Elliptic Curves*, GTM, no. 106, Springer-Verlag, 1986.

[89] J. H. Silverman and J. Tate, *Rational Points on Elliptic Curves*, Springer-Verlag New York Inc., 1992.

[90] J. Solinas, *Low-weight binary representations for pairs of integers*, Tech. Report CORR 2001-41, Department of Combinatorics and Optimization, University of Waterloo, Canada, 2001.

[91] J. Stein, *Computational problems associated with Racah algebra*, Journal of Computational Physics **1** (1967), 397–405.

[92] J. J. Thomas, J. M. Keller, and G. N. Larsen, *The calculation of multiplicative inverses over GF(p) efficiently where p is a Mersenne prime*, IEEE Transactions on Computers **35** (1986), no. 5, 478–482.

[93] R. Tijdeman, *On the maximal distance between integers composed of small primes*, Compositio Mathematica **28** (1974), 159–162.

[94] L. Washington, *Elliptic Curves: Number Theory and Cryptography*, CRC Press, 2003.

[95] W. Waterhouse, *Abelian varieties over finite fields*, Ann. Sci. École Norm. Sup. $4^e$ **série** (1969), no. 2, 521–560.

[96] ANSI X9.42-2000, *Public key cryptography for the financial services industry: Agreement of symmetric keys using discrete logarithm cryptography*, December 1999.

[97] Y. Yacobi, *Fast exponentiation using data compression*, SIAM J. Comput. **28** (1998), no. 2, 700–703.

[98] A. C. Yao, *On the evaluation of powers*, SIAMJC **5** (1976), 100–103.

# Appendix A

# Finite Fields

## A.1 Introduction

This appendix provides an introduction to the theory of finite fields. We will focus on the application of finite fields to elliptic curve arithmetic.

A *field* is a triple consisting of a non-empty set in conjunction with two binary operations. These two operations are called addition ($+$) and multiplication ($\times$). The operations must satisfy certain field axioms.

Examples of fields include the rational numbers $\mathbb{Q}$ and the real numbers $\mathbb{R}$, both with standard addition and multiplication. Another example of a field is $\mathbb{Z}/(p)$, the set of integers modulo $p$ where $p$ is a prime number. A field is called finite when the underlying set is finite. For every prime number $p$, we know that $\mathbb{Z}/(p)$ is a field with $p$ elements. In fact, we will show that $\mathbb{Z}/(p)$ is the only field of size $p$ up to isomorphism. There is a unique field of size $p^k$ for all prime numbers $p$ and positive integers $k$, denoted by $\mathbb{F}_{p^k}$.

This appendix provides an introduction to finite fields and will single out prime fields ($\mathbb{F}_p$ for $p$ prime) for special consideration. There are other finite fields that are relevant for elliptic curve cryptography, including binary fields (see Hankerson *et al.* [42]) and optimal extension fields (see Bailey and Paar [7]), but these are beyond the scope of this thesis.

To implement a finite field for use in cryptography, an efficient representation of elements is needed, as well as efficient algorithms for performing addition, subtraction, multiplication, squaring, inversion and modular reduction. Elements of a prime field are be represented by multi-precision integers. Multi-precision integers are stored as an array of word-sized integers, where the word size is often chosen to correspond with the hardware word size.

In Section A.3, we present algorithms for addition, subtraction, multiplication, inversion and modular reduction for field elements given in this form. Operations such as addition and multiplication are computed on the integer representatives of the field elements and then reduced to canonical form with a modular reduction algorithm. We describe algorithms for addition and subtraction on the multi-precision integer representatives in Section A.3.1 and we describe multiplication and squaring for multi-precision integers in Section A.3.2.

Modular reduction algorithms are examined in Section A.3.3. The operation is performed for all moduli with generic algorithms such as Barrett reduction and Montgomery reduction. For special moduli, such as those suggested by NIST, we present specialized algorithms.

Inversion is performed with a number of variants of the Euclidean algorithm. We will present these algorithms and an algorithm for simultaneous inversion in Section A.3.4.

The algorithms presented in this section are well suited to software implementation and are necessary precursors to the algorithms for elliptic curve arithmetic presented in Chapter 2.

## A.2 Finite Field Basics

The results in this appendix can be found in most introductory algebra books such as Artin [4] or Lang [58]. To define a finite field, we must first recall the definition of a field.

**Definition A.2.1.** *A* field *is a triple* $(F, +_F, \times_F)$ *consisting of a set* $F$ *and two binary operations* $+_F$ *and* $\times_F$ *on* $F$ *that satisfy the following properties:*

- $(F, +_F)$ *forms an Abelian group with identity denoted by* $0_F$.
- $(F \setminus \{0_F\}, \times_F)$ *forms an Abelian group with identity denoted by* $1_F$.
- *The distributive law holds:* $(a +_F b) \times_F c = (a \times_F c) +_F (a \times_F b)$ *for all* $a, b, c \in F$.

*The field is often denoted by the set alone. If* $F$ *is finite, then it is called a* finite field *and the size of the set* $F$*, denoted by* $|F|$*, is the* order *of the field.*

Every field has an associated number called the *characteristic* of the field.

**Definition A.2.2.** *The* characteristic *of a field* $F$ *is the smallest integer* $p$ *such that* $\underbrace{1 +_F 1 +_F \cdots +_F 1}_{p \text{ times}} = 0$. *If there is no such* $p$*, then the characteristic of the field is* $0$.

For example, $\mathbb{Z}/(p)$, the set of integers modulo $p$ with standard addition and multiplication modulo $p$, is a field of characteristic $p$ when $p$ is prime. Proposition A.2.3 is from Ireland and Rosen [45, Ch. 7] and demonstrates that a field can be constructed with a prime power order $p^k$ for every prime $p$ and positive integer $k$.

**Proposition A.2.3.**

- *For any prime $p$, the set $\mathbb{Z}/(p) = \{0, 1, \ldots, p-1\}$ along with the operations $+_{\mathbb{Z}/(p)}$ and $\times_{\mathbb{Z}/(p)}$ form a field. The operations are defined for $a, b \in \mathbb{Z}/(p)$ as follows:*

$$a +_{\mathbb{Z}/(p)} b = a + b \bmod p,$$
$$a \times_{\mathbb{Z}/(p)} b = a \times b \bmod p.$$

- *Given an irreducible polynomial $f \in (\mathbb{Z}/(p))[x]$ of degree $k$, define the set $(\mathbb{Z}/(p))[x] \,/\, (f)$ to be the set of polynomials of degree less than $k$ with the operations $+_{(\mathbb{Z}/(p))[x] \,/\, (f)}$ and $\times_{(\mathbb{Z}/(p))[x] \,/\, (f)}$ forms a field. The operations are defined for $a, b \in (\mathbb{Z}/(p))[x] \,/\, (f)$ as follows:*

$$a +_{(\mathbb{Z}/(p))[x] \,/\, (f)} b = a + b \bmod f,$$
$$a \times_{(\mathbb{Z}/(p))[x] \,/\, (f)} b = a \times b \bmod f.$$

The field $(\mathbb{Z}/(p))[x]/(f)$ is a vector space of dimension $k$ over $\mathbb{Z}/(p)$. Note that there are irreducible polynomials over $\mathbb{Z}/(p)$ of every positive degree, see Adleman and Lenstra [1] and Shoup [86] for methods to find such polynomials. Since irreducible polynomials of every degree are easy to find, there exist explicitly computable finite fields of order $p^k$ for every prime $p$ and positive integer $k$. We now define field isomorphism.

**Definition A.2.4.** *Let $F$ and $G$ be fields. A map $\phi$ from $F$ to $G$ is called an* isomorphism *if it satisfies the following properties:*

- $\phi(a +_F b) = \phi(a) +_G \phi(b)$,
- $\phi(a \times_F b) = \phi(a) \times_G \phi(b)$, *and*

- *$\phi$ is onto*

*for all $a, b \in F$. Two fields $F$ and $G$ are called* isomorphic *if there exists an isomorphism from $F$ onto $G$.*

With this notion of isomorphism, it is possible to characterize all finite fields with Theorem A.2.5 taken from Artin [4, Thm. 6.4].

**Theorem A.2.5.** *Every finite field $F$ has characteristic $p$ for some prime integer $p$ and order $p^k$ for some positive integer $k$. Any two finite fields with the same order are isomorphic. The "unique" finite field of order $p^k$ will be denoted by $\mathbb{F}_{p^k}$.*

Theorem A.2.5 demonstrates that every finite field is isomorphic to one of the fields defined in Proposition A.2.3. If $p$ is prime, then $\mathbb{F}_p$ is called a *prime field.* For the remainder of this appendix, we will drop the subscripts on $+_F, \times_F, 0_F$ and $1_F$ if the context is clear.

**Definition A.2.6.** *For a given field $F$, a* subfield *$G$* of *$F$ is a subset of $F$ such that $G$ is a subgroup of $F$ under $+$ and $G \setminus \{0\}$ is a subgroup of $F \setminus \{0\}$ under $\times$, i.e.*

- *$1 \in G$ and*
- *if $a, b \in G \setminus \{0\}$, then $a - b \in G \setminus \{0\}$ and $a \times b^{-1} \in G \setminus \{0\}$*

*If $G$ is a subfield of $F$, we say that $F$ is an* extension *of $G$.*

Another relevant fact is that the multiplicative group of a finite field is cyclic. This is stated as Theorem A.2.7 and is taken from Artin [4, Thm. 6.4].

**Theorem A.2.7.** *The multiplicative group of a finite field $\mathbb{F}_{p^k}$, denoted by $\mathbb{F}_{p^k}^* = (\mathbb{F}_{p^k} \setminus \{0\}, \times)$ is a cyclic group of order $p^k - 1$.*

## A.3 Prime Fields

In this section, we describe explicit algorithms for performing calculations in prime fields and specialized reduction algorithms for certain finite fields that are relevant to elliptic curve cryptography. For this discussion, we only consider finite fields $\mathbb{F}_p$ where $p$ is prime. Here we describe a method to represent prime fields efficiently on a computer and the algorithms to perform the basic field operations using this representation. This subject has been widely examined (see [48], [63], [87]) and is of great importance to elliptic curve arithmetic.

The field elements are represented by their residues modulo $p$, namely, the numbers $\{0, 1, \ldots, p-1\}$. These field element representatives are stored on a computer as multi-precision integers. A multi-precision integer is a representation of an integer as a sequence of integers from 0 to $W-1$, where $W$ is a value called the *word size*. If

$$k = (k_{n-1}, \ldots, k_0)_W$$

is the base $W$ representation of $k$, then the multi-precision integer representation of $k$ is the sequence $(k_{n-1}, \ldots, k_0)$.

In practice, the word length $W$ is chosen to agree with the physical word size of the processor, but in general, $W$ could be any positive integer. We will assume henceforth that the word size is the $w$th power of 2 for an arbitrary positive integer $w$. An integer $k$ with binary representation

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

is stored as a number of words. For word-size $2^w$, the integer $k$ is represented by

$d = \lceil \frac{l}{w} \rceil$ integers $K_i$ with bit-length at most $w$, or

$$(K_{d-1}, \cdots, K_0),$$

where

$$K_0 = (k_{w-1}, \ldots, k_0),$$

$$K_1 = (k_{2w-1}, \ldots, k_w),$$

$$\vdots$$

$$K_{d-1} = (0, \ldots, 0, k_{l-1}, \ldots, k_{(d-1)w-1}).$$

To perform operations on multi-precision integers, we will assume that the following operations can be performed:

1. Addition of two one-word integers.

2. Subtraction of two one-word integers.

3. Multiplication of two one-word integers resulting in a two-word integer.

An overview of some of the algorithms in this section can be found in Knuth [52, Ch. 4] and Cohen *et al.* [6, Ch. 10 and 11].

Addition and subtraction is performed on field elements by adding or subtracting their integer representatives word-by-word and keeping track of carry bits. If the total is larger than $p$, then $p$ is subtracted to obtain a representative in $\{0, \ldots, p-1\}$. In the subtraction of $a$ from $b$, if $a < b$, the result is obtained by subtracting $b$ from $p$ and adding the result to $a$.

Classical integer multiplication is performed in one of two modes, operand scanning and product scanning. Karatsuba-Ofman multiplication can also be used for large integers. Squaring is performed in much the same way as classical multiplication with a modification to take advantage of the fact that both multiplicands are identical. The results of integer multiplication and squaring will be around twice the length of the multiplicands, so a modular reduction step is performed to return an element in $\{0, \dots, p-1\}$.

Barrett reduction or Montgomery reduction are used to perform modular reduction in the general case. We will also introduce a reduction techniques for primes of a special form, such as those proposed by NIST [78].

Inversion is performed using the binary extended GCD algorithm. Simultaneous inversion can be performed to invert a set of elements at the cost of one inversion and extra multiplications.

With this set of algorithms, we have explicit methods to compute every finite field operation needed for elliptic curve arithmetic.

### A.3.1 Addition

Multi-precision addition and subtraction are algorithms for arithmetic on multi-precision integers. Addition and subtraction are performed in the same manner as addition by hand, with the difference that each digit is a word rather than a decimal digit. The lowest order words are added modulo the word size, and if their sum exceeds the word size, a carry bit is set to one, otherwise the carry bit is zero. The next words are added together modulo the word size again, the carry is added and the new carry bit is set to one or zero as before. This process is repeated for all the words to obtain the sum.

We are working modulo a prime $p$, so if the total exceeds the prime modulus $p$, then $p$ is subtracted once to obtain an integer in the correct range.

Algorithm 38 performs multi-precision addition on two integers. If both integers are different lengths, then the smaller integer is padded with zeros so that their lengths are equal, taken from Cohen *et al.* [6, Alg. 10.3].

```
Algorithm 38: Multi-Precision Addition
Input: $a, b \in \{0, \ldots, 2^{dw} - 1\}$, given as $a = (a_{d-1}, \ldots, a_0)_{2^w}, b = (b_{d-1}, \ldots, b_0)_{2^w}$, $a_{d-1} \neq 0$
Output: $(c_{d-1}, \ldots, c_0)_{2^w} = a + b \bmod 2^{dw}$, carry bit $c$
(1)    $c \leftarrow 0$
(2)    for $i = 0$ to $d - 1$
(3)        $c_i \leftarrow (a_i + b_i + c) \bmod 2^w$
(4)        $c \leftarrow \lfloor (a_i + b_i + c)/2^w \rfloor$
(5)    return $(c, c_{d-1}, \ldots, c_0)_{2^w}$
```

If the resulting number exceeds $p$, then a subtraction by $p$ is necessary in order to obtain the canonical representation.

```
Algorithm 39: Multi-Precision Subtraction
Input: $a, b \in \{0, \ldots, 2^{dw} - 1\}$, given as $a = (a_{d-1}, \ldots, a_0)_{2^w}, b = (b_{d-1}, \ldots, b_0)_{2^w}$ and $a \geq b$
Output: $(c_{d-1}, \ldots, c_0)_{2^w} = a - b \bmod 2^{dw}$
(1)    $c \leftarrow 0$
(2)    for $i = 0$ to $d - 1$
(3)        $c_i \leftarrow (a_i - b_i + c) \bmod 2^w$
(4)        $c \leftarrow \lfloor (a_i - b_i + c)/2^w \rfloor$
(5)    return $(c_{d-1}, \ldots, c_0)_{2^w}$
```

Addition and subtraction require a number of word operations that is linear in the word size of the largest input value.

### A.3.2 Multiplication

Multiplication of elements in $\mathbb{F}_p$ is computed in two ways; either the two integer representatives are multiplied in $\mathbb{Z}$ and the result is reduced modulo $p$, or the reduction is performed at every step of the multiplication.

There are a number methods for multiplication in prime fields that we will not examine in depth. Dhem [26] describes a method called *interleaving* for prime field multiplication. Montgomery [73] described a different representation of integers that can be used to perform multiplication and reduction. Koç *et al.* [55] provide a summary of Montgomery multiplication techniques. Modular reduction can be expensive to repeat after every step, so we only deal with multiplying the integer representatives.

In this section we present some algorithms for multiplying two multi-precision integers. These are two forms of schoolbook multiplication and Karatsuba multiplication. Reducing the product modulo a prime $p$ is dealt with in Section A.3.3.

### Schoolbook Multiplication

The first type of multiplication is schoolbook multiplication in operand scanning mode. A one-word number $a$ can be multiplied with a multi-word integer $b$ by multiplying the lowest order word of $b$ by $a$ and saving the carry, then multiplying the next lowest order integer by $a$ and adding the carry and continuing until all the words are calculated. In operand scanning mode, each word of the first number is multiplied by the second number. The sum of these products gives the product of the two multi-precision numbers. Algorithm 40 carries out this operation, taken from Cohen *et al.* [6, Alg. 10.8].

```
Algorithm 40: Multi-Precision Multiplication (Operand Scanning)
Input: $a \in \{0, \dots, 2^{d_1 w} - 1\}, b \in \{0, \dots, 2^{d_2 w} - 1\}$, given as $a = (a_{d_1-1}, \dots, a_0)_{2^w}, b = (b_{d_2-1}, \dots, b_0)_{2^w}$ and $a \geq b$
Output: $(c_{d_1+d_2-1}, \dots, c_0)_{2^w} = a \times b$
(1)    for $i = 0$ to $d_2 - 1$
```

```
(2)         $c_i \leftarrow 0$
(3)     for $i = 0$ to $d_2 - 1$
(4)         $c \leftarrow 0$
(5)         if $a_i = 0$
(6)             $b_{d_2-1} \leftarrow 0$
(7)         else
(8)             for $j = 0$ to $m - 1$
(9)                 $t \leftarrow a_i b_j + c_{i+j} + c$
(10)                $c_{i+j} \leftarrow t \bmod 2^w$
(11)                $c \leftarrow \lfloor t/2^w \rfloor$
(12)            $c_{d_2+1} \leftarrow c$
(13)    return $(c_{d_1+d_2-1}, \ldots, c_0)_{2^w}$
```

The second mode for integer multiplication is product scanning, sometimes called Comba's method [21]. This method is analogous to polynomial multiplication, each word of the *product* is computed in sequence.

The product of two $t$-word integers will have at most $2t - 1$ words. The lowest order word of the product is computed as the product of the lowest order words of the multiplicands modulo the word size. Anything exceeding the word size is carried on to later words. The $k^{\text{th}}$ word of the product is computed from the sum of the products of the $i^{\text{th}}$ and $j^{\text{th}}$ words of the multiplicands for all $i + j = k$, plus the carries from the previous words. This is taken modulo the word size and the excess is carried on to the next word. Algorithm 41 carries out this operation, adapted from Hankerson *et al.* [43, Alg. 2.10].

**Algorithm 41:** Multi-Precision Multiplication (Product Scanning)
**Input:** $a \in \{0, \ldots, 2^{d_1 w} - 1\}, b \in \{0, \ldots, 2^{d_2 w} - 1\}$, given as $a = (a_{d_1-1}, \ldots, a_0)_{2^w}, b = (b_{d_2-1}, \ldots, b_0)_{2^w}$ and $a \geq b$
**Output:** $(c_{d_1+d_2-1}, \ldots, c_0)_{2^w} = a \times b$

```
(1)     for $i = 0$ to $d_2 - 1$
(2)         $c_i \leftarrow 0$
(3)     $r_0 \leftarrow 0, r_1 \leftarrow 0, r_2 \leftarrow 0$
(4)     $d \leftarrow \max(d_1, d_2)$
```

```
(5)     for $k = 0$ to $2d - 2$
(6)        foreach $0 \le i \le \min(d-1, k)$
(7)           $t \leftarrow a_i b_{k-i}$
(8)           $r_0 \leftarrow t + r_0 \bmod 2^w$
(9)           $c \leftarrow \lfloor (t \bmod 2^w + r_0)/2^w \rfloor$
(10)          $r_1 \leftarrow r_1 + \lfloor t/2^w \rfloor + c \bmod 2^w$
(11)          $c \leftarrow \lfloor (r_1 + \lfloor t/2^w \rfloor + c)/2^w \rfloor$
(12)          $r_2 \leftarrow r_2 + c \bmod 2^w$
(13)       $c_k \leftarrow r_0, r_0 \leftarrow r_1, r_1 \leftarrow r_2,\ r_2 \leftarrow 0$
(14)    $c_{2d-1} \leftarrow r_0$
(15)    return $(c_{2d-1}, \ldots, c_0)_{2^w}$
```

Comba [21] compared these two modes of multiplication for IBM PCs and found that in practice product scanning is often faster than operand scanning.

## Karatsuba-Ofman Multiplication

Karatsuba-Ofman multiplication is an algorithm to multiply two $n$-bit numbers that takes less time asymptotically than the previous algorithms.

Suppose that $j, k$ are $2l$-word integers and $k = k_1 2^{wl} + k_0, j = j_1 2^{wl} + j_0$ where $k_1, k_0, j_1, j_0$ are $l$ word integers. The key observation is that

$$\begin{aligned} jk =& (j_1 2^{wl} + j_0)(k_1 2^{wl} + k_0), \\ =& j_1 k_1 2^{2wl} + j_0 k_0 + (j_0 k_1 + j_1 k_0) 2^{wl}, \\ =& j_1 k_1 2^{2wl} + j_0 k_0 + \\ & ((j_0 + j_1)(k_0 + k_1) - j_0 k_0 - j_1 k_1) 2^{wl}. \end{aligned}$$

The value $jk$ can therefore be computed using only three multiplications of $l$ word integers, namely the products $j_0 k_0, j_1 k_1$ and $(j_0 + j_1)(k_0 + k_1)$. These are combined to form the product using a few single-word additions, subtractions and two multiplications by powers of $2^w$. Multiplying one multi-precision integer by the word size is done by appending a word consisting of zeros on as the low order word.

Karatsuba [50, 49] introduced a method to use this decomposition recursively to reduce the complexity of integer multiplication. Instead of computing the product using schoolbook multiplication, requiring the equivalent of four $l$-word integer multiplications, we compute the three $l$-word multiplications $j_0k_0, j_1k_1$ and $(j_0+j_1)(k_0+k_1)$ as described above. Moreover, compute each of these smaller multiplications recursively, reducing each to three $\lceil l/2 \rceil$-word integer multiplications.

The complexity of Karatsuba-Ofman multiplication is $O(d^{\log_2 3})$ for $d$-word integers, compared to $O(d^2)$ for classical multiplication (see Knuth [52, Ch. 4]). However, the crossover point where this algorithm is faster than classical multiplication can be quite high, depending on the implementation and hardware. Some of these considerations are described in the manual for the GMP library [41]; the results suggest that the crossover for $d$ can range from 8 up to more than 100. In Brown *et al.* [16], it was found that Karatsuba multiplication is not efficient for elliptic curve cryptography for current ranges, although this could change in the future. Since classical multiplication is faster for smaller words, a threshold is set so that Karatsuba-Ofman is used recursively for multiplications of integers larger than the threshold and schoolbook multiplication is used for the rest. This threshold is set experimentally. Algorithm 42 is a recursive algorithm for Karatsuba-Ofman multiplication from Cohen *et al.* [6, Alg. 10.11].

```
Algorithm 42: Multi-Precision Multiplication (Karatsuba-Ofman)
Input: $a, b \in \{0, \dots, 2^{dw} - 1\}$, given as $a = (a_{d_1-1}, \dots, a_0)_{2^w}, b = (b_{d_2-1}, \dots, b_0)_{2^w}$, threshold $t$, $d = \max(d_1, d_2)$
Output: $(c_{d_1+d_2-1}, \dots, c_0)_{2^w} = a \times b$
(1)    if $d \leq t$ then compute $ab$ using Algorithm 40 or 41
(2)    $p \leftarrow \lfloor d/2 \rfloor$
(3)    $q \leftarrow \lceil d/2 \rceil$
```

```
(4)   $A_0 \leftarrow (a_{q-1}, \ldots, a_0)_2$
(5)   $B_0 \leftarrow (b_{q-1}, \ldots, b_0)_2$
(6)   $A_1 \leftarrow (a_{p+q-1}, \ldots, a_q)_2$
(7)   $B_1 \leftarrow (b_{p+q-1}, \ldots, b_q)_2$
(8)   $A_s \leftarrow A_0 + A_1$
(9)   $B_s \leftarrow B_0 + B_1$
(10)  Compute $A_0B_0$, $A_1B_1$, $A_sB_s$ recursively
(11)  return $A_1B_12^{2q} + (A_sB_s - A_1B_1 - A_0B_0)2^q + A_0B_0$
```

Algorithm 42 can be very efficient in practice, especially for large inputs.

## Squaring

Squaring is a special case of multiplication. The fact that both multiplicands are the same allows for time-saving modifications.

Both the operand scanning and product scanning modes of multiplication translate to squaring algorithms. Algorithms 43 (derived from Algorithm 40) and 44 (from Hankerson *et al.* [43, Alg. 2.13]) are the multi-precision squaring algorithms in operand scanning mode and product scanning mode, respectively. They follow Algorithms 40 and 41 with modifications to reduce the complexity. Since $a_ib_j = a_jb_i$, only one of these products needs to be computed. Instead of looping through all $i$ and $j$, the modified algorithm loops through all $i \leq j$ and computes $2a_ia_j$ when $i < j$. This reduces the number of single-word multiplications needed by a factor of roughly two.

```
Algorithm 43: Multi-Precision Squaring (Operand Scanning)
Input: $a \in \{0, \ldots, 2^{dw} - 1\}$, given as $a = (a_{d-1}, \ldots, a_0)_{2^w}$
Output: $(c_{2d-1}, \ldots, c_0)_{2^w} = a^2$
(1)   for $i = 0$ to $2d - 1$
(2)       $c_i \leftarrow 0$
(3)   for $i = 0$ to $d - 1$
(4)       $t \leftarrow a_i^2 + b_{2i}$
(5)       $b_{2i} \leftarrow c \bmod 2^w$
```

```
(6)       $c \leftarrow \lfloor t/2^w \rfloor$
(7)       for $j = i + 1$ to $d - 1$
(8)          $t \leftarrow 2a_i a_j + c_{i+j} + c$
(9)          $c_{i+j} \leftarrow t \bmod 2^w$
(10)         $c \leftarrow \lfloor t/2^w \rfloor$
(11)      $c_{i+n} \leftarrow c$
(12)   return $(c_{2d-1}, \ldots, c_0)_{2^w}$
```

**Algorithm 44:** Multi-Precision Squaring (Product Scanning)
**Input:** $a \in \{0, \ldots, 2^{dw} - 1\}$, given as $a = (a_{d-1}, \ldots, a_0)_{2^w}$
**Output:** $(c_{2d-1}, \ldots, c_0)_{2^w} = a^2$

```
(1)    for $i = 0$ to $d - 1$
(2)       $c_i \leftarrow 0$
(3)    $r_0 \leftarrow 0, r_1 \leftarrow 0, r_2 \leftarrow 0$
(4)    for $k = 0$ to $2d - 2$
(5)       foreach $0 \leq i \leq \min(d - 1, k - i)$
(6)          if $i = k - i$
(7)             $t \leftarrow a_i^2$
(8)          else
(9)             $t \leftarrow 2a_i b_{k-i}$
(10)            $c \leftarrow \lfloor t/2^{2w} \rfloor$
(11)            $r_2 \leftarrow r_2 + c \bmod 2^w$
(12)            $t \leftarrow t \bmod 2^{2w}$
(13)         $r_0 \leftarrow t + r_0 \bmod 2^w$
(14)         $c \leftarrow \lfloor (t \bmod 2^w + r_0)/2^w \rfloor$
(15)         $r_1 \leftarrow r_1 + \lfloor t/2^w \rfloor + c \bmod 2^w$
(16)         $c \leftarrow \lfloor (r_1 + \lfloor t/2^w \rfloor + c)/2^w \rfloor$
(17)         $r_2 \leftarrow r_2 + c \bmod 2^w$
(18)      $c_k \leftarrow r_0, r_0 \leftarrow r_1, r_1 \leftarrow r_2, r_2 \leftarrow 0$
(19)   $c_{2d-1} \leftarrow r_0$
(20)   return $(c_{2d-1}, \ldots, c_0)_{2^w}$
```

Algorithm 45 is the squaring variant of Karatsuba-Ofman multiplication, derived from Algorithm 42.

**Algorithm 45:** Multi-Precision Squaring (Karatsuba-Ofman)
**Input:** $a \in \{0, \ldots, 2^{dw} - 1\}$, given as $a = (a_{d-1}, \ldots, a_0)_{2^w}$, threshold $t$
**Output:** $(c_{2d-1}, \ldots, c_0)_{2^w} = a^2$

```
(1)    if $d \leq t$ then compute $a^2$ using Algorithm 43 or 44
```

```
(2)    $p \leftarrow \lfloor d/2 \rfloor$
(3)    $q \leftarrow \lceil d/2 \rceil$
(4)    $A_0 \leftarrow (a_{q-1}, \ldots, a_0)_2$
(5)    $A_1 \leftarrow (a_{p+q-1}, \ldots, a_q)_2$
(6)    $A_s \leftarrow A_0 + A_1$
(7)    Compute $A_0^2$, $A_1^2$, $A_s^2$ recursively
(8)    **return** $A_1^2 2^{2q} + (A_s^2 - A_1^2 - A_0^2)2^q + A_0^2$
```

As with general multiplication, squaring produces a result that is generally larger than the modulus. In order to obtain a canonical representative of $\mathbb{F}_q$, the result must be reduced modulo the prime modulus.

### A.3.3 Modular Reduction

Modular reduction takes an integer and a modulus and returns the non-negative remainder of the integer divided by the modulus. For prime fields $\mathbb{F}_p$, modular reduction of an integer by $p$ returns the unique representative of that integer in $\{0, 1, \ldots, p-1\}$.

The results after multiplications of $l$-word integers from Section A.3.2 will be $2l$-word integers. If the integer returned is larger than $p$, then a modular reduction is necessary to return the corresponding representative in $\mathbb{F}_p$. Formally, the problem of modular reduction can be stated as follows: for a given $k$, find $0 \leq k' < p$ such that $k = qp + k'$ for some integer $q$.

Barrett [8] introduced a method now known as Barrett reduction. Modular reduction is computed by estimating the value of the quotient $q = \lfloor k/p \rfloor$. The algorithm requires the precomputation of one division, the value $R = \lfloor 2^{2wd}/p \rfloor$. The value we are trying to estimate is

$$\begin{aligned} q &= \left\lfloor \frac{k}{p} \right\rfloor \\ &= \left\lfloor \frac{\frac{k}{2^{(w-1)d}} \frac{2^{2wd}}{p}}{2^{(w+1)d}} \right\rfloor \\ &\approx \left\lfloor \frac{\left\lfloor \frac{k}{2^{(w-1)d}} \right\rfloor R}{2^{(w+1)d}} \right\rfloor . \end{aligned}$$

The estimate

$$\hat{q} = \left\lfloor \lfloor a/2^{w(d-1)} \rfloor R/2^{w(d+1)} \right\rfloor ,$$

is guaranteed to satisfy $q \leq \hat{q} \leq q + 2$, therefore there are only three values of $q$ to check. A short proof of this fact is described in Hankerson *et al.* [43, Sec. 2.2.4].

Algorithm 46 performs reduction of a $2d$-word number modulo a $d$-word modulus by first guessing $\hat{q}$ for $q$. If this does not produce a value $r$ in the correct range, then $r - p$ or $r - 2p$ are in the correct range. This algorithm is adapted from Barrett [8].

```
Algorithm 46: Barrett Reduction
Input: $d$-word modulus $p$, $0 \leq a \leq 2^{2dw}$, $R = \lfloor 2^{2wd}/p \rfloor$
Output: $a \bmod p$
(1)   $\hat{q} \leftarrow \lfloor \lfloor a/2^{w(d-1)} \rfloor R/2^{w(d+1)} \rfloor$
(2)   $r \leftarrow (a \bmod 2^{w(d+1)}) - (\hat{q}p \bmod 2^{w(d+1)})$
(3)   if $r < 0$ then $r \leftarrow r + 2^{w(d+1)}$
(4)   while $r \geq p$
(5)       $r \leftarrow r - p$
(6)   return $r$
```

Step 5 will only be executed at most twice. For a proof of correctness, see Hankerson *et al.* [43, Sec. 2.2.4].

There are other general reduction algorithms that are of cryptographic interest, notably the Montgomery method mentioned briefly in Section A.3.2. See Crandall and Pomerance [25, Sec. 9.2] for a comparison of the Barrett and Montgomery methods. We will not examine Montgomery reduction in depth because in this thesis we focus more on fields over special primes.

NIST recommends a set of primes that are of a specific form that permits fast reduction. The fast reduction algorithms for the five NIST primes and a complexity analysis is given in this section.

A *Mersenne* prime is a prime of the form $2^p - 1$. Reduction of a number $n = (n_{k-1}, \ldots, n_0)_2$ modulo a Mersenne prime $M_p = 2^p - 1$ can be performed easily. Since $2^p - 1 \equiv 0 \bmod M_p$, we have $2^p \equiv 1 \bmod M_p$. This means

$$\begin{aligned} n &\equiv (n_{k-1}, \ldots, n_{k-1-p})_2 2^{k-p-1} + (n_{k-p-2}, \ldots, n_0)_2 \bmod M_p \\ &\equiv (n_{k-1}, \ldots, n_{k-1-p})_2 2^{k-2p-1} + (n_{k-p-2}, \ldots, n_0)_2 \bmod M_p \\ &\equiv (n_{k-p-2} + n_{k-1}, \ldots, n_{k-2p-2} + n_{k-1-p}, n_{k-2p-3}, \ldots, n_0)_2 \bmod M_p. \end{aligned}$$

This reduction step reduces the number of digits in the representative of $n$ by $p-1$. To reduce completely, this reduction is applied until the representative for $n$ has fewer than $p$ digits.

Mersenne primes are very rare, there are none with size in the cryptographically interesting range from 127 to 521 bits. However, Crandall [24] introduced the notion of a pseudo-Mersenne prime that has similar reduction properties. A *pseudo-Mersenne* prime is a number of the form $2^p - c$ where $c$ is a $t$-bit integer with $p < t$.

Given such a number $n$,

$$\begin{aligned} n &\equiv (n_{k-1}, \ldots, n_{k-1-p})_2 2^{k-p-1} + (n_{k-p-2}, \ldots, n_0)_2 \bmod (2^p - c), \\ &\equiv (n_{k-1}, \ldots, n_{k-1-p})_2 2^{k-2p-1} c + (n_{k-p-2}, \ldots, n_0)_2 \bmod (2^p - c). \end{aligned}$$

The resulting reduction is used in the same way to reduce the number of digits in the representative for $n$ by $p - t$.

In 1999, NIST [78] published a list of primes for use in elliptic curve cryptography. They are, as follows:

$$\begin{aligned} &P192 = 2^{192} - 2^{64} - 1, \\ &P224 = 2^{224} - 2^{96} + 1, \\ &P256 = 2^{256} - 2^{224} + 2^{192} + 2^{96} - 1, \\ &P384 = 2^{384} - 2^{128} - 2^{96} + 2^{32} - 1, and \\ &P521 = 2^{521} - 1. \end{aligned}$$

The NIST prime $P521$ is a Mersenne prime and the others are pseudo-Mersenne primes with low Hamming weight. Algorithms 47 to 51 perform modular reduction by the NIST primes $P192$ through $P521$. These algorithms are from Hankerson *et al.* [43, Sec. 2.2.6] with a change in notation to conform with the style of this thesis.

```
Algorithm 47: Fast Reduction Modulo $P192$
Input: Integer $a = (a_5, a_4, a_3, a_2, a_1, a_0)_{2^{64}} < (P192)^2$
Output: $a \bmod P192$
(1)   $s_1 = (a_2, a_1, a_0)_{2^{64}}$
(2)   $s_2 = (0, a_3, a_3)_{2^{64}}$
(3)   $s_3 = (a_4, a_4, 0)_{2^{64}}$
(4)   $s_4 = (a_5, a_5, a_5)_{2^{64}}$
(5)   return $s_1 + s_2 + s_3 + s_4 \bmod P192$
```

```
**Algorithm 48:** Fast Reduction Modulo $P224$
**Input:** Integer $a = (a_{13}, \ldots, a_0)_{2^{32}} < (P224)^2$
**Output:** $a \bmod P224$
(1)   $s_1 = (a_6, a_5, a_4, a_3, a_2, a_1, a_0)_{2^{32}}$
(2)   $s_2 = (a_{10}, a_9, a_8, a_7, 0, 0, 0)_{2^{32}}$
(3)   $s_3 = (0, a_{13}, a_{12}, a_{11}, 0, 0, 0)_{2^{32}}$
(4)   $s_4 = (a_{13}, a_{12}, a_{11}, a_{10}, a_9, a_8, a_7)_{2^{32}}$
(5)   $s_5 = (0, 0, 0, 0, a_{13}, a_{12}, a_{11})_{2^{32}}$
(6)   **return** $s_1 + s_2 + s_3 - s_4 - s_5 \bmod P224$
```

```
**Algorithm 49:** Fast Reduction Modulo $P256$
**Input:** Integer $a = (a_{15}, \ldots, a_0)_{2^{32}} < (P256)^2$
**Output:** $a \bmod P256$
(1)   $s_1 = (a_7, a_6, a_5, a_4, a_3, a_2, a_1, a_0)_{2^{32}}$
(2)   $s_2 = (a_{15}, a_{14}, a_{13}, a_{12}, a_{11}, 0, 0, 0)_{2^{32}}$
(3)   $s_3 = (0, a_{15}, a_{14}, a_{13}, a_{12}, 0, 0, 0)_{2^{32}}$
(4)   $s_4 = (a_{15}, a_{14}, 0, 0, 0, a_{10}, a_9, a_8)_{2^{32}}$
(5)   $s_5 = (a_8, a_{13}, a_{15}, a_{14}, a_{13}, a_{11}, a_{10}, a_9)_{2^{32}}$
(6)   $s_6 = (a_{10}, a_9, 0, 0, 0, a_{13}, a_{12}, a_{11})_{2^{32}}$
(7)   $s_7 = (a_{11}, a_9, 0, 0, a_{15}, a_{14}, a_{13}, a_{12})_{2^{32}}$
(8)   $s_8 = (a_{12}, 0, a_{10}, a_9, a_8, a_{15}, a_{14}, a_{13})_{2^{32}}$
(9)   $s_9 = (a_{13}, 0, a_{11}, a_{10}, a_9, 0, a_{15}, a_{14})_{2^{32}}$
(10)  **return** $s_1 + 2s_2 + 2s_3 + s_4 + s_5 - s_6 - s_7 - s_8 - s_9 \bmod P256$
```

```
**Algorithm 50:** Fast Reduction Modulo $P384$
**Input:** Integer $a = (a_{23}, \ldots, a_0)_{2^{32}} < (P384)^2$
**Output:** $a \bmod P384$
(1)   $s_1 = (a_{11}, a_{10}, a_9, a_8, a_7, a_6, a_5, a_4, a_3, a_2, a_1, a_0)_{2^{32}}$
(2)   $s_2 = (0, 0, 0, 0, 0, a_{23}, a_{22}, a_{21}, 0, 0, 0, 0)_{2^{32}}$
(3)   $s_3 = (a_{23}, a_{22}, a_{21}, a_{20}, a_{19}, a_{18}, a_{17}, a_{16}, a_{15}, a_{14}, a_{13}, a_{12})_{2^{32}}$
(4)   $s_4 = (a_{20}, a_{19}, a_{18}, a_{17}, a_{16}, a_{15}, a_{14}, a_{13}, a_{12}, a_{23}, a_{22}, a_{21})_{2^{32}}$
(5)   $s_5 = (a_{19}, a_{18}, a_{17}, a_{16}, a_{15}, a_{14}, a_{13}, a_{12}, a_{20}, 0, a_{23}, 0)_{2^{32}}$
(6)   $s_6 = (0, 0, 0, 0, a_{23}, a_{22}, a_{21}, a_{20}, 0, 0, 0, 0)_{2^{32}}$
(7)   $s_7 = (0, 0, 0, 0, 0, 0, a_{23}, a_{22}, a_{21}, 0, 0, a_{20})_{2^{32}}$
(8)   $s_8 = (a_{22}, a_{21}, a_{20}, a_{19}, a_{18}, a_{17}, a_{16}, a_{15}, a_{14}, a_{13}, a_{12}, a_{23})_{2^{32}}$
(9)   $s_9 = (0, 0, 0, 0, 0, 0, 0, a_{23}, a_{22}, a_{21}, 0, a_{20}, 0)_{2^{32}}$
(10)  $s_{10} = (0, 0, 0, 0, 0, 0, 0, a_{23}, a_{23}, 0, 0, 0)_{2^{32}}$
(11)  **return** $s_1 + 2s_2 + s_3 + s_4 + s_5 + s_6 + s_7 - s_8 - s_9 - s_{10} \bmod P384$
```

**Algorithm 51:** Fast Reduction Modulo $P521$

```
Input: Integer $a = (a_{1041}, a_{1040}, \ldots, a_0)_2 < (P521)^2$
Output: $a \bmod P521$
(1)    $s_1 = (a_{1041}, \ldots, a_{522}, a_{521})_2$
(2)    $s_2 = (a_{520}, \ldots, a_1, a_0)_2$
(3)    return $s_1 + s_2 \bmod P521$
```

The NIST reduction techniques are faster than Barrett reduction in general; see Hankerson *et al.* [43, Sec. 5.1.5] for a comparison.

### A.3.4 Inversion

Finding the inverse of an element of the multiplicative group of a finite field is called inversion. The classical way to find an inverse is the extended Euclidean algorithm. However, in its standard form, the extended Euclidean algorithm requires some expensive division operations. These divisions are replaced by bit shifts in the binary version of the extended Euclidean algorithm. In this section, we describe this process.

The *greatest common divisor (GCD)* of two integers $a$ and $b$, not both 0, is the largest positive integer dividing both $a$ and $b$ and is denoted $\gcd(a, b)$. For each such pair of integers $(a, b)$, there exist unique integers $s$ and $t$ such that

$$sa + tb = \gcd(a, b).$$

The extended Euclidean algorithm computes the values $s, t$ and $\gcd(a, b)$ from $a$ and $b$. For this process, the division algorithm is needed, which takes positive integers $a, b$ and returns $q, r$ such that $a = qb + r, 0 \leq r < b$.

The key fact of the Euclidean algorithm is that if $a = bq + r$, then $\gcd(a, b) = \gcd(b, r)$. The GCD of two integers can be computed by repeatedly applying the division algorithm, first on $(a, b)$ to obtain $q_1, r_1$, then on $(b, r_1)$ to obtain $(q_2, r_2)$, on

$(r_1, r_2)$ to obtain $(q_3, r_3)$ and continue until $r_k \mid r_{k-1}$. When this process terminates, $r_{k+1} = 0$ and $r_k$ is the resulting GCD. The extended GCD algorithm keeps track of the values $q_i$ to compute $(s, t)$ so that $as + bt = \gcd(a, b)$ when the algorithm terminates.

As mentioned above, the main problem with the Euclidean algorithm is the division step. This step is at least as expensive as modular reduction. The binary Euclidean algorithm replaces the need for this expensive operation with much simpler halving operation. A halving operation is simply a bit shift of the binary representation, an operation that is easy to perform in hardware. The improved algorithm is possible due to the following facts about GCDs.

- if $a$ and $b$ are both even, then $\gcd(a, b) = 2\gcd(a/2, b/2)$,
- if $a$ is even and $b$ is odd, then $\gcd(a, b) = \gcd(a/2, b)$,
- if $a$ and $b$ are both odd, then $\gcd(a, b) = \gcd(a, |a - b|/2)$.

The binary extended Euclidean algorithm uses these facts to reduce the sizes of $a$ and $b$ until the GCD is obtained while keeping track of $s, t$ so that $as + bt = \gcd(a, b)$. If $p$ is a prime and $a$ is coprime to $p$, then $\gcd(a, p) = 1$. Here, the extended Euclidean algorithm generates integers $s, t$ such that $as + pt = 1$. We then see that $s(\mathrm{mod} p)$ is the inverse of $a(\mathrm{mod} p)$. Algorithm 52 computes $s \bmod p$ such that $as + pt = 1$ for a prime $p$ and and integer $a$, $0 \leq a < p$. For each iteration of the loop, both numbers are divided by 2 until they are both odd, then $p$ is set to $|a - p|$. By of the properties of the GCD, these transformations to $a$ and $p$ do not change $\gcd(a, p)$, so this process is repeated until either $a$ or $p$ is 0. The algorithm then returns the last non-zero remainder.

```
Algorithm 52: Binary Inversion
Input: integer $a$, prime $p$
Output: $a^{-1} \bmod p$
(1)   $u \leftarrow a$, $v \leftarrow p$
(2)   $x_1 \leftarrow 1, x_2 \leftarrow 0$
(3)   while $u \neq 1$ and $v \neq 1$
(4)      while $u$ is even
(5)         $u \leftarrow u/2$
(6)         if $x_1$ is even then $x_1 \leftarrow x_1/2$
(7)                        else $x_1 \leftarrow (x_1 + p)/2$
(8)      while $v$ is even
(9)         $v \leftarrow v/2$
(10)        if $x_2$ is even then $x_2 \leftarrow x_2/2$
(11)                       else $x_2 \leftarrow (x_2 + p)/2$
(12)     if $u \leq v$ then $u \leftarrow u - v$, $x_1 \leftarrow x_1 - x_2$
(13)               else $v \leftarrow v - u, x_2 \leftarrow x_2 - x_1$
(14)  if $u = 1$ then return $x_1 \bmod p$
(15)            else return $x_2 \bmod p$
(16)  return $s_1 + s_2 \bmod p$
```

This version of the algorithm can be found in Hankerson *et al.* [43, Alg. 2.22] and was adapted from Stein [91]. Its proof of correctness can be found in Knuth [52, Ch. 4].

Algorithm 52 works for any modulus, Thomas *et al.* [92] introduced an alternative inversion algorithm that only works for prime moduli. For every iteration of the algorithm, $q = -\lfloor p/a \rfloor$ is computed and $a$ is set to $p + qa$ which is necessarily a smaller positive value. Multiplying $a$ by $q$ gives the next $a$ value modulo $p$. When $a$ reaches 1, the product of all the values of $q$ are the inverse of the original $a$. Algorithm 53 computes the inverse of a prime field element in this manner and was adapted from the presentation in Cohen *et al.* [6, Alg. 11.9].

```
Algorithm 53: Prime Field Binary Inversion
Input: prime $p$, integer $a$ in $\{1, \ldots, p-1\}$
```

```
Output: $a^{-1} \bmod p$
(1)   $u \leftarrow 1$
(2)   while $a \neq 1$
(3)       $q \leftarrow -\lfloor p/a \rfloor$
(4)       $z \leftarrow p + qa$
(5)       $u \leftarrow q \cdot u \bmod p$
(6)   return $u$
```

Thomas *et al.* [92] provide a proof of termination. The division in Step 3 can be performed inexpensively for certain primes such as Mersenne primes according to Crandall and Pomerance [25, p. 428].

There are other inversion techniques including Lehmer's technique [60] to compute the extended GCD and Montgomery inversion [73]. We do not discuss Lehmer's technique because it provides the full results of the extended euclidean algorithm, rather than just the inverse. Montgomery inversion is not included because it is used in conjunction with Montgomery reduction, which we have not discussed. Several algorithms are also presented in Cohen *et al.* [6, Sec. 11.1.3].

### Simultaneous Inversion

Montgomery [75] introduced a method for computing the inverses of multiple elements of a prime field using only one inverse and a number of multiplications. The technique involves multiplying all of the elements together and computing the inverse of the product. The inverse of one element can be obtained by multiplying the inverse of the product of all the elements by every element except the one being inverted. For example, to invert $x$, $y$, compute $(xy)^{-1}$ then $x^{-1} = (xy)^{-1}y$ and $y^{-1} = (xy)^{-1}x$. Algorithm 54 inverts a set of elements using this trick, adapted from Montgomery [75].

**Algorithm 54:** Simultaneous Inversion

```
Input: prime $p$, elements $a_1, \ldots, a_k$ in $\mathbb{F}_p$
Output: $(b_1, \ldots, b_k) = (a_1^{-1} \bmod p, \ldots, a_k^{-1} \bmod p)$
(1)    $c_1 \leftarrow a_1$
(2)    for $i$ from 2 to $k$
(3)        $c_i \leftarrow c_{i-1} a_i \bmod p$
(4)    $c \leftarrow c_k^{-1}$
(5)    for $i$ from $k$ to 2
(6)        $b_i \leftarrow c \cdot c_{i-1} \bmod p$
(7)        $a \leftarrow c \cdot a_i \bmod p$
(8)    return $(b_1, \ldots, b_k)$
```

Algorithm 54 takes $3(k-1)$ multiplications, one inversion and $k$ temporary elements. This method will be more efficient than $k$ inversions as long as three multiplications take less time to compute than one inversion, as is usually the case in prime fields.

## A.4 Complexity Summary

The algorithms presented in this appendix are designed for implementation in software. In the this thesis, these operations are used in algorithms for elliptic curve arithmetic. In order to compare the speed of these algorithms, a comparative analysis of the time it takes to compute different finite field operations is used. Brown *et al.* [16] performed an extensive set of timings for various algorithms for arithmetic on the NIST curves.

First we must note that the relative and absolute speeds of the prime field operations discussed above depend greatly on implementation and hardware. Table A.1 presents the timings for prime field arithmetic on an 800 MHz Intel Pentium III, including reduction to canonical form from Hankerson *et al.* [43]. The variables $M$, $S$ and $I$ represent the average time it takes to perform one multiplication, squaring

Table A.1: Estimated Time of Prime Field Operations (in $\mu$s)

| Operation | $\mathbb{F}_{P192}$ | $\mathbb{F}_{P224}$ | $\mathbb{F}_{P256}$ | $\mathbb{F}_{P384}$ | $\mathbb{F}_{P521}$ |
|---|---|---|---|---|---|
| Addition | 0.07 | 0.07 | 0.08 | 0.10 | 0.10 |
| Reduction | 0.11 | 0.12 | 0.30 | 0.38 | 0.20 |
| Multiplication | 0.42 | 0.52 | 0.81 | 1.47 | 2.32 |
| Squaring | 0.36 | 0.44 | 0.71 | 1.23 | 1.87 |
| Inversion | 25.2 | 34.3 | 44.3 | 96.3 | 163.8 |
| $S/M$ | 0.86 | 0.85 | 0.88 | 0.84 | 0.81 |
| $I/M$ | 60.0 | 70.0 | 54.7 | 65.5 | 70.6 |

or inversion in $\mathbb{F}_p$, respectively.

The algorithms used to implement these operations are very similar to the algorithms presented in this appendix. Addition was performed with Algorithm 38. Reduction with fast reduction Algorithms 47 through 51. Multiplication and squaring were computed in product scanning mode similar to Algorithms 41 and 44 and inversion was computed as in Algorithm 52. The timings would be different for different variants of the specific algorithms. Inversion is often assumed to be worst case since multiplication is a more important operation and is often better optimized than inversion. Squaring is often assumed to be best case because most multiplication optimizations will affect squaring and multiplication equally. In this thesis we follow the conventions from Okeya and Sakurai [80] and Lim and Hwang [65] and assume that $S = (4/5)M$, $I = 80M$.

# Appendix B

# Algorithm Costs

In this section, we present tables describing the costs of algorithms from Chapters 3 and 4 that were not included in the text.

Tables B.1 to B.3 describe the average field costs and average $M$-costs of the window NAF method with various window sizes and $d$ chosen to correspond to the length of the NIST primes $P224$, $P384$ and $P521$. Both versions of the algorithm are included; using the regular Chudnovsky Jacobian table and converting the table to affine form. The fastest version for a given amount of storage space is written in bold. If none of the terms are bold, then there is a faster version requiring less storage space.

Tables B.4 to B.6 describe the average field costs and average $M$-costs of the sliding window method with various window sizes and $d$ chosen to correspond to the length of the NIST primes $P224$, $P256$ and $P521$. Both versions of the algorithm are included; using the regular Chudnovsky Jacobian table and converting the table

Table B.1: Window NAF Method Average Cost ($d = 224$)

| $w$ | Regular Table | | Affine Table | | Storage |
|---|---|---|---|---|---|
| | Field Cost | $M$-cost | Field Cost | $M$-cost | |
| 3 | **I+1344.0M+1066.0S** | **2276.8M** | 2I+1347.0M+1067.0S | 2360.6M | 1 |
| 4 | **I+1342.1M+1038.4S** | **2252.8M** | 2I+1291.4M+1041.4S | 2284.5M | 3 |
| 5 | **I+1342.4M+1028.0S** | **2244.8M** | 2I+1299.7M+1035.0S | 2287.7M | 7 |
| 6 | I+1387.4M+1036.0S | 2296.2M | 2I+1393.0M+1051.0S | 2393.8M | 15 |
| 7 | I+1525.9M+1072.0S | 2463.5M | 2I+1633.0M+1103.0S | 2675.4M | 31 |

Table B.2: Window NAF Method Average Cost ($d = 256$)

| $w$ | Regular Table | | Affine Table | | Storage |
|---|---|---|---|---|---|
| | Field Cost | $M$-cost | Field Cost | $M$-cost | |
| 3 | **I+1536.0M+1218.0S** | **2590.4M** | 2I+1539.0M+1219.0S | 2674.2M | 1 |
| 4 | **I+1530.9M+1185.6S** | **2559.4M** | 2I+1470.6M+1188.6S | 2581.5M | 3 |
| 5 | **I+1525.1M+1172.0S** | **2542.7M** | 2I+1470.3M+1179.0S | 2573.5M | 7 |
| 6 | I+1563.9M+1177.7S | 2586.1M | 2I+1557.6M+1192.7S | 2671.7M | 15 |
| 7 | I+1697.2M+1212.0S | 2746.8M | 2I+1793.0M+1243.0S | 2947.4M | 31 |

Table B.3: Window NAF Method Average Cost ($d = 384$)

| $w$ | Regular Table | | Affine Table | | Storage |
|---|---|---|---|---|---|
| | Field Cost | $M$-cost | Field Cost | $M$-cost | |
| 3 | **I+2304.0M+1826.0S** | **3844.8M** | 2I+2307.0M+1827.0S | 3928.6M | 1 |
| 4 | I+2286.1M+1774.4S | 3785.6M | **2I+2187.4M+1777.4S** | **3769.3M** | 3 |
| 5 | I+2255.8M+1748.0S | 3734.2M | **2I+2153.0M+1755.0S** | **3717.0M** | 7 |
| 6 | I+2270.2M+1744.6S | 3745.9M | 2I+2215.9M+1759.6S | 3783.5M | 15 |
| 7 | I+2382.2M+1772.0S | 3879.8M | 2I+2433.0M+1803.0S | 4035.4M | 31 |

Table B.4: Sliding Window Method Average Cost ($d = 224$)

| $w$ | Regular Table | | Affine Table | | Storage |
|---|---|---|---|---|---|
| | Field Cost | $M$-cost | Field Cost | $M$-cost | |
| 3 | **I+1354.0M+1050.3S** | **2274.3M** | 2I+1314.2M+1052.3S | 2316.1M | 2 |
| 4 | **I+1345.3M+1035.0S** | **2253.3M** | 2I+1291.3M+1039.0S | 2282.5M | 4 |
| 5 | I+1359.9M+1030.4S | 2264.2M | 2I+1333.1M+1040.4S | 2325.4M | 10 |
| 6 | I+1430.5M+1046.9S | 2348.0M | 2I+1467.1M+1066.9S | 2480.6M | 20 |
| 7 | I+1635.7M+1101.5S | 2596.9M | 2I+1810.8M+1143.5S | 2885.6M | 42 |

Table B.5: Sliding Window Method Average Cost ($d = 256$)

| $w$ | Regular Table | | Affine Table | | Storage |
|---|---|---|---|---|---|
| | Field Cost | $M$-cost | Field Cost | $M$-cost | |
| 3 | **I+1546.0M+1199.7S** | **2585.7M** | 2I+1499.1M+1201.7S | 2620.4M | 2 |
| 4 | **I+1533.1M+1181.3S** | **2558.1M** | 2I+1468.1M+1185.3S | 2576.3M | 4 |
| 5 | I+1540.4M+1173.5S | 2559.1M | 2I+1501.3M+1183.5S | 2608.0M | 10 |
| 6 | I+1605.4M+1188.0S | 2635.8M | 2I+1630.1M+1208.0S | 2756.5M | 20 |
| 7 | I+1805.4M+1241.0S | 2878.2M | 2I+1969.5M+1283.0S | 3155.9M | 42 |

to affine form. The fastest version for a given amount of storage space is written in bold. If none of the terms are bold, then there is an version requiring less storage space that is faster.

Tables B.7 to B.9 describe the average field costs and average $M$-costs of the fractional window method with different values of $w$ and $m$ with $d$ chosen to correspond to the length of the NIST primes $P224$, $P256$ and $P521$. Both versions of the algorithm are included; using the Chudnovsky Jacobian table and using the table converted to affine coordinates. The fastest version for a given amount of storage space is written in bold. If none of the terms are bold, then there is a faster version requiring less storage.

Table B.6: Sliding Window Method Average Cost ($d = 384$)

| $w$ | Regular Table | | Affine Table | | Storage |
|---|---|---|---|---|---|
| | Field Cost | $M$-cost | Field Cost | $M$-cost | |
| 3 | **I+2314.0M+1797.0S** | **3831.6M** | 2I+2238.7M+1799.0S | 3837.9M | 2 |
| 4 | I+2284.0M+1766.4S | 3777.1M | **2I+2175.1M+1770.4S** | **3751.5M** | 4 |
| 5 | I+2262.3M+1745.7S | 3738.8M | **2I+2173.9M+1755.7S** | **3738.4M** | 10 |
| 6 | I+2304.9M+1752.5S | 3787.0M | 2I+2282.1M+1772.5S | 3860.1M | 20 |
| 7 | I+2484.0M+1799.1S | 4003.2M | 2I+2604.2M+1841.1S | 4237.0M | 42 |

Table B.7: Fractional Window Method Average Cost ($d = 224$)

| $w$ | $m$ | Regular Table | | Affine Table | | Storage |
|---|---|---|---|---|---|---|
| | | Field Cost | $M$-cost | Field Cost | $M$-cost | |
| 1 | 1 | **I+1344.0M+1066.0S** | **2276.8M** | 2I+1347.0M+1067.0S | 2360.6M | 1 |
| 2 | 1 | **I+1354.0M+1050.3S** | **2274.3M** | 2I+1314.2M+1052.3S | 2316.1M | 2 |
| | 3 | **I+1342.1M+1038.4S** | **2252.8M** | 2I+1291.4M+1041.4S | 2284.5M | 3 |
| 3 | 1 | I+1345.3M+1035.0S | 2253.3M | 2I+1291.3M+1039.0S | 2282.5M | 4 |
| | 3 | **I+1345.3M+1032.2S** | **2251.0M** | 2I+1292.8M+1037.2S | 2282.6M | 5 |
| | 5 | **I+1344.0M+1029.9S** | **2247.9M** | 2I+1295.7M+1035.9S | 2284.3M | 6 |
| | 7 | **I+1342.4M+1028.0S** | **2244.8M** | 2I+1299.7M+1035.0S | 2287.7M | 7 |
| 4 | 1 | I+1348.6M+1028.7S | 2251.5M | 2I+1310.6M+1036.7S | 2299.9M | 8 |
| | 3 | I+1354.3M+1029.5S | 2258.0M | 2I+1321.7M+1038.5S | 2312.5M | 9 |
| | 5 | I+1359.9M+1030.4S | 2264.2M | 2I+1333.1M+1040.4S | 2325.4M | 10 |
| | 7 | I+1365.3M+1031.4S | 2270.5M | 2I+1344.7M+1042.4S | 2338.6M | 11 |

Table B.8: Fractional Window Method Average Cost ($d = 256$)

| $w$ | $m$ | Regular Table | | Affine Table | | Storage |
|---|---|---|---|---|---|---|
| | | Field Cost | $M$-cost | Field Cost | $M$-cost | |
| 1 | 1 | **I+1536.0M+1218.0S** | **2590.4M** | 2I+1539.0M+1219.0S | 2674.2M | 1 |
| 2 | 1 | **I+1546.0M+1199.7S** | **2585.7M** | 2I+1499.1M+1201.7S | 2620.4M | 2 |
| | 3 | **I+1530.9M+1185.6S** | **2559.4M** | 2I+1470.6M+1188.6S | 2581.5M | 3 |
| 3 | 1 | **I+1533.1M+1181.3S** | **2558.1M** | 2I+1468.1M+1185.3S | 2576.3M | 4 |
| | 3 | **I+1531.5M+1177.6S** | **2553.6M** | 2I+1467.4M+1182.6S | 2573.5M | 5 |
| | 5 | **I+1528.4M+1174.6S** | **2548.1M** | 2I+1468.2M+1180.6S | 2572.6M | 6 |
| | 7 | **I+1525.1M+1172.0S** | **2542.7M** | 2I+1470.3M+1179.0S | 2573.5M | 7 |
| 4 | 1 | I+1530.6M+1172.4S | 2548.5M | 2I+1480.4M+1180.4S | 2584.7M | 8 |
| | 3 | I+1535.6M+1172.9S | 2553.9M | 2I+1490.7M+1181.9S | 2596.2M | 9 |
| | 5 | I+1540.4M+1173.5S | 2559.1M | 2I+1501.3M+1183.5S | 2608.0M | 10 |
| | 7 | I+1545.0M+1174.2S | 2564.4M | 2I+1512.1M+1185.2S | 2620.2M | 11 |

Table B.9: Fractional Window Method Average Cost ($d = 384$)

| $w$ | $m$ | Regular Table | | Affine Table | | Storage |
|---|---|---|---|---|---|---|
| | | Field Cost | $M$-cost | Field Cost | $M$-cost | |
| 1 | 1 | **I+2304.0M+1826.0S** | **3844.8M** | 2I+2307.0M+1827.0S | 3928.6M | 1 |
| 2 | 1 | **I+2314.0M+1797.0S** | **3831.6M** | 2I+2238.7M+1799.0S | 3837.9M | 2 |
| | 3 | I+2286.1M+1774.4S | 3785.6M | **2I+2187.4M+1777.4S** | **3769.3M** | 3 |
| 3 | 1 | I+2284.0M+1766.4S | 3777.1M | **2I+2175.1M+1770.4S** | **3751.5M** | 4 |
| | 3 | I+2276.2M+1759.5S | 3763.7M | **2I+2165.5M+1764.5S** | **3737.1M** | 5 |
| | 5 | I+2266.2M+1753.3S | 3748.9M | **2I+2158.3M+1759.3S** | **3725.7M** | 6 |
| | 7 | I+2255.8M+1748.0S | 3734.2M | **2I+2153.0M+1755.0S** | **3717.0M** | 7 |
| 4 | 1 | I+2258.5M+1747.1S | 3736.2M | 2I+2159.6M+1755.1S | 3723.6M | 8 |
| | 3 | I+2260.6M+1746.3S | 3737.6M | 2I+2166.5M+1755.3S | 3730.8M | 9 |
| | 5 | I+2262.3M+1745.7S | 3738.8M | 2I+2173.9M+1755.7S | 3738.4M | 10 |
| | 7 | I+2263.8M+1745.2S | 3740.0M | 2I+2181.6M+1756.2S | 3746.6M | 11 |

Table B.10: Scalar Multiplication ($P224$)

| Algorithm | Variables | Storage | Field Cost | $M$-cost |
|---|---|---|---|---|
| R2L Binary (12) | – | – | I+2229.0M+1338.5S | 3379.8M |
| L2R Binary (13) | – | – | I+1772.0M+1219.0S | 2827.2M |
| L2R NAF (15) | – | – | I+1482.3M+1114.0S | 2453.5M |
| $w$-NAF (17) | $w = 5$, $\mathcal{J}^c$ | 7 | I+1342.4M+1028.0S | 2244.8M |
| swNAF (18) | $w = 4$, $\mathcal{J}^c$ | 5 | I+1345.3M+1032.2S | 2251.0M |
| fwNAF (20) | $w = 3$, $c = 5$, $\mathcal{J}^c$ | 7 | I+1342.4M+1028.0S | 2244.8M |
| DBChain (22) | $b_{max} = 121, t_{max} = 65$ | – | I+1460.0M+956.3S | 2305.0M |

Tables B.10 to B.12 describe the result of Chapter 3 for parameters corresponding to the NIST primes $P224$, $P256$ and $P384$ in terms of their storage requirements (in points), their field costs and their $M$-costs.

Tables B.14 to B.17 list the cost of fixed base windowing for values of $d$ that correspond to the length of the NIST primes $P224$, $P256$, $P384$ and $P521$. The variable $l = \lceil \frac{d}{w} \rceil$ where $d$ is the size of the input and $w$ is the fixed windowing size.

Tables B.18 to B.21 list the cost of Algorithm 24 for $d$ chosen to correspond to

Table B.11: Scalar Multiplication ($P256$)

| Algorithm | Variables | Storage | Field Cost | $M$-cost |
|---|---|---|---|---|
| R2L Binary (12) | – | – | I+2549.0M+1530.5S | 3853.4M |
| L2R Binary (13) | – | – | I+2028.0M+1395.0S | 3224.0M |
| L2R NAF (15) | – | – | I+1695.7M+1274.0S | 2794.9M |
| $w$-NAF (17) | $w = 5$, $\mathcal{J}^c$ | 7 | I+1525.1M+1172.0S | 2542.7M |
| swNAF (18) | $w = 4$, $\mathcal{J}^c$ | 5 | I+1533.1M+1181.3S | 2558.1M |
| fwNAF (20) | $w = 3$, $c = 5$, $\mathcal{J}^c$ | 7 | I+1525.1M+1172.0S | 2542.7M |
| DBChain (22) | $b_{max} = 134, t_{max} = 77$ | – | I+1676.1M+1089.3S | 2624.7M |

Table B.12: Scalar Multiplication ($P384$)

| Algorithm | Variables | Storage | Field Cost | $M$-cost |
|---|---|---|---|---|
| R2L Binary (12) | – | – | I+3829.0M+2298.5S | 5747.8M |
| L2R Binary (13) | – | – | I+3052.0M+2099.0S | 4811.2M |
| L2R NAF (15) | – | – | I+2549.0M+1914.0S | 4160.2M |
| $w$-NAF (17) | $w = 5$, $\mathcal{A}$ | 7 | 2I+2153.0M+1755.0S | 3717.0M |
| swNAF (18) | $w = 5$, $\mathcal{A}$ | 10 | 2I+2173.9M+1755.7S | 3738.4M |
| fwNAF (20) | $w = 3$, $c = 5$, $\mathcal{A}$ | 7 | 2I+2153.0M+1755.0S | 3717.0M |
| DBChain (22) | $b_{max} = 216, t_{max} = 106$ | – | I+2487.8M+1645.8S | 3884.4M |

Table B.13: Scalar Multiplication ($P521$)

| Algorithm | Variables | Storage | Field Cost | $M$-cost |
|---|---|---|---|---|
| R2L Binary (12) | – | – | I+5199.0M+3120.5S | 7775.4M |
| L2R Binary (13) | – | – | I+4148.0M+2852.5S | 6510.0M |
| L2R NAF (15) | – | – | I+3462.3M+2599.0S | 5621.5M |
| $w$-NAF (17) | $w = 5$, $\mathcal{A}$ | 7 | 2I+2883.7M+2371.5S | 4940.9M |
| swNAF (18) | $w = 5$, $\mathcal{A}$ | 10 | 2I+2893.8M+2368.2S | 4948.3M |
| fwNAF (20) | $w = 3$, $c = 5$, $\mathcal{A}$ | 7 | 2I+2883.7M+2371.5S | 4940.9M |
| DBChain (22) | $b_{max} = 269, t_{max} = 159$ | – | I+3407.6M+2207.8S | 5241.9M |

Table B.14: Fixed-Base Window Average Cost ($d = 224$)

| $w$ | Storage | Field Cost | $M$-cost |
|---|---|---|---|
| 2 | 111 | I+741.7M+275.8S | 1042.3M |
| 3 | 74 | I+603.1M+220.4S | 859.4M |
| 4 | 55 | I+538.6M+190.7S | 771.2M |
| 5 | 44 | I+583.0M+195.1S | 819.1M |
| 6 | 37 | I+759.3M+238.0S | 1029.7M |
| 7 | 31 | I+1177.8M+347.4S | 1535.8M |
| 8 | 27 | I+2062.9M+585.6S | 2611.3M |

Table B.15: Fixed-Base Window Average Cost ($d = 256$)

| $w$ | Storage | Field Cost | $M$-cost |
|---|---|---|---|
| 2 | 127 | I+844.1M+314.2S | 1175.4M |
| 3 | 85 | I+683.1M+250.4S | 963.5M |
| 4 | 63 | I+599.7M+213.6S | 850.6M |
| 5 | 51 | I+638.1M+215.8S | 890.7M |
| 6 | 42 | I+799.5M+253.0S | 1081.9M |
| 7 | 36 | I+1219.6M+362.9S | 1589.9M |
| 8 | 31 | I+2098.9M+598.7S | 2657.9M |

Table B.16: Fixed-Base Window Average Cost ($d = 384$)

| $w$ | Storage | Field Cost | $M$-cost |
|---|---|---|---|
| 2 | 191 | I+1253.7M+467.8S | 1708.0M |
| 3 | 127 | I+988.7M+365.0S | 1360.7M |
| 4 | 95 | I+843.8M+305.1S | 1167.9M |
| 5 | 76 | I+834.1M+289.2S | 1145.5M |
| 6 | 63 | I+967.3M+315.7S | 1299.9M |
| 7 | 54 | I+1366.9M+417.7S | 1781.0M |
| 8 | 47 | I+2236.3M+649.2S | 2835.7M |

Table B.17: Fixed-Base Window Average Cost ($d = 521$)

| $w$ | Storage | Field Cost | $M$-cost |
|---|---|---|---|
| 2 | 260 | I+1695.3M+633.4S | 2282.1M |
| 3 | 173 | I+1323.3M+490.5S | 1795.7M |
| 4 | 130 | I+1110.7M+405.2S | 1514.8M |
| 5 | 104 | I+1053.3M+371.4S | 1430.4M |
| 6 | 86 | I+1150.1M+384.2S | 1537.5M |
| 7 | 74 | I+1528.2M+478.0S | 1990.6M |
| 8 | 65 | I+2385.2M+704.5S | 3028.9M |

Table B.18: Fixed-Base Comb Average Cost ($d = 224$)

| $l$ | Storage | Field Cost | $M$-cost |
|---|---|---|---|
| 2 | 2 | I+1113.0M+694.8S | 1748.8M |
| 3 | 6 | I+817.0M+491.2S | 1290.0M |
| 4 | 14 | I+635.5M+375.7S | 1016.0M |
| 5 | 30 | I+520.0M+304.9S | 843.9M |
| 6 | 62 | I+442.4M+258.3S | 729.0M |
| 7 | 126 | I+373.1M+217.3S | 626.9M |
| 8 | 254 | I+326.2M+189.7S | 557.9M |

the length of the NIST primes $P224$, $P256$, $P384$ and $P521$. The variable $w = \lceil \frac{d}{l} \rceil$ where $d$ is the size of the input and $l$ is given.

Tables B.22 to B.25 describe the field costs and $M$-costs of Algorithm 28 with $d$ corresponding to the length of the NIST primes $P224$, $P256$, $P384$ and $P521$. Recall that the $M$-cost is the field cost with substitutions $S \leftarrow (4/5)M$ and $I \leftarrow 80M$. The tables for $d$ chosen to correspond with are similar and can be found in Appendix B.

Tables B.26 to B.29 describe the field costs and $M$-costs of Algorithm 28 with $d$ corresponding to the length of the NIST primes $P224$, $P256$, $P384$ and $P521$.

Tables B.30 to B.33 describe the field costs and $M$-costs of Algorithm 32 with $d$

Table B.19: Fixed-Base Comb Average Cost ($d = 256$)

| $l$ | Storage | Field Cost | $M$-cost |
|---|---|---|---|
| 2 | 2 | I+1273.0M+794.8S | 1988.8M |
| 3 | 6 | I+938.0M+564.1S | 1469.3M |
| 4 | 14 | I+727.5M+430.2S | 1151.7M |
| 5 | 30 | I+602.2M+353.2S | 964.8M |
| 6 | 62 | I+501.8M+293.0S | 816.2M |
| 7 | 126 | I+432.8M+252.2S | 714.5M |
| 8 | 254 | I+374.0M+217.6S | 628.1M |

Table B.20: Fixed-Base Comb Average Cost ($d = 384$)

| $d$ | Storage | Field Cost | $M$-cost |
|---|---|---|---|
| 2 | 2 | I+1913.0M+1194.8S | 2948.8M |
| 3 | 6 | I+1400.0M+842.4S | 2153.9M |
| 4 | 14 | I+1095.5M+648.2S | 1694.0M |
| 5 | 30 | I+896.0M+525.9S | 1396.7M |
| 6 | 62 | I+751.1M+439.0S | 1182.4M |
| 7 | 126 | I+647.6M+377.7S | 1029.8M |
| 8 | 254 | I+565.5M+329.4S | 909.1M |

Table B.21: Fixed-Base Comb Average Cost ($d = 521$)

| $d$ | Storage | Field Cost | $M$-cost |
|---|---|---|---|
| 2 | 2 | I+2603.0M+1626.0S | 3983.8M |
| 3 | 6 | I+1906.0M+1147.1S | 2903.7M |
| 4 | 14 | I+1498.0M+886.6S | 2287.3M |
| 5 | 30 | I+1225.0M+719.2S | 1880.4M |
| 6 | 62 | I+1024.2M+599.0S | 1583.4M |
| 7 | 126 | I+886.4M+517.3S | 1380.2M |
| 8 | 254 | I+781.0M+455.2S | 1225.2M |

Table B.22: $p^{th}$ Order Binary Method Average Cost ($d = 224$)

| $N$ | Field Cost | $M$-cost | $s_1$ |
|---|---|---|---|
| 2 | I+1200.7M+1008.5S | 2087.5M | 3 |
| 3 | I+1114.4M+975.7S | 1975.0M | 5 |
| 4 | I+1065.3M+957.2S | 1911.1M | 5 |
| 5 | I+1047.9M+950.2S | 1888.0M | 7 |
| 6 | I+1028.2M+942.8S | 1862.5M | 7 |
| 7 | I+1014.2M+937.6S | 1844.2M | 7 |
| 8 | I+1003.7M+933.6S | 1830.6M | 7 |

Table B.23: $p^{th}$ Order Binary Method Average Cost ($d = 256$)

| $N$ | Field Cost | $M$-cost | $s_1$ |
|---|---|---|---|
| 2 | I+1371.3M+1152.5S | 2373.3M | 3 |
| 3 | I+1270.9M+1114.3S | 2242.4M | 5 |
| 4 | I+1214.7M+1093.2S | 2169.3M | 5 |
| 5 | I+1192.9M+1084.6S | 2140.6M | 7 |
| 6 | I+1170.4M+1076.2S | 2111.4M | 7 |
| 7 | I+1154.4M+1070.1S | 2090.5M | 7 |
| 8 | I+1142.3M+1065.6S | 2074.8M | 7 |

Table B.24: $p^{th}$ Order Binary Method Average Cost ($d = 384$)

| $N$ | Field Cost | $M$-cost | $s_1$ |
|---|---|---|---|
| 2 | I+2054.0M+1728.5S | 3516.8M | 3 |
| 3 | I+1896.7M+1669.0S | 3311.9M | 5 |
| 4 | I+1812.0M+1637.2S | 3201.8M | 5 |
| 5 | I+1773.2M+1622.2S | 3151.0M | 7 |
| 6 | I+1739.3M+1609.5S | 3106.9M | 7 |
| 7 | I+1715.1M+1600.4S | 3075.5M | 7 |
| 8 | I+1697.0M+1593.6S | 3051.9M | 7 |

Table B.25: $p^{th}$ Order Binary Method Average Cost ($d = 521$)

| $N$ | Field Cost | $M$-cost | $s_1$ |
|---|---|---|---|
| 2 | I+2784.7M+2345.0S | 4740.7M | 3 |
| 3 | I+2566.4M+2262.7S | 4456.6M | 5 |
| 4 | I+2451.3M+2219.5S | 4306.9M | 5 |
| 5 | I+2394.3M+2197.6S | 4232.3M | 7 |
| 6 | I+2348.2M+2180.3S | 4172.5M | 7 |
| 7 | I+2315.3M+2168.0S | 4129.7M | 7 |
| 8 | I+2290.7M+2158.8S | 4097.7M | 7 |

Table B.26: Right-to-Left Parallel Method Average Cost ($d = 224$)

| $N$ | Field Cost | $M$-cost | $s_1$ |
|---|---|---|---|
| 2 | I+1194.7M+1005.0S | 2078.7M | 1 |
| 3 | I+1100.0M+972.0S | 1957.6M | 1 |
| 4 | I+1045.3M+949.0S | 1884.5M | 1 |
| 5 | I+1020.0M+942.0S | 1853.6M | 1 |
| 6 | I+1013.3M+947.0S | 1850.9M | 1 |
| 7 | I+981.3M+925.0S | 1801.3M | 1 |
| 8 | I+970.7M+921.0S | 1787.5M | 1 |

Table B.27: Right-to-Left Parallel Method Average Cost ($d = 256$)

| $N$ | Field Cost | $M$-cost | $s_1$ |
|---|---|---|---|
| 2 | I+1365.3M+1149.0S | 2364.5M | 1 |
| 3 | I+1261.3M+1115.0S | 2233.3M | 1 |
| 4 | I+1194.7M+1085.0S | 2142.7M | 1 |
| 5 | I+1178.7M+1089.0S | 2129.9M | 1 |
| 6 | I+1146.7M+1072.0S | 2084.3M | 1 |
| 7 | I+1134.7M+1070.0S | 2070.7M | 1 |
| 8 | I+1109.3M+1053.0S | 2031.7M | 1 |

Table B.28: Right-to-Left Parallel Method Average Cost ($d = 384$)

| $N$ | Field Cost | $M$-cost | $s_1$ |
|---|---|---|---|
| 2 | I+2048.0M+1725.0S | 3508.0M | 1 |
| 3 | I+1877.3M+1661.0S | 3286.1M | 1 |
| 4 | I+1792.0M+1629.0S | 3175.2M | 1 |
| 5 | I+1745.3M+1614.0S | 3116.5M | 1 |
| 6 | I+1706.7M+1597.0S | 3064.3M | 1 |
| 7 | I+1686.7M+1592.0S | 3040.3M | 1 |
| 8 | I+1664.0M+1581.0S | 3008.8M | 1 |

Table B.29: Right-to-Left Parallel Method Average Cost ($d = 521$)

| $N$ | Field Cost | $M$-cost | $s_1$ |
|---|---|---|---|
| 2 | I+2784.0M+2346.0S | 4740.8M | 1 |
| 3 | I+2552.0M+2259.0S | 4439.2M | 1 |
| 4 | I+2445.3M+2224.0S | 4304.5M | 1 |
| 5 | I+2380.0M+2202.0S | 4221.6M | 1 |
| 6 | I+2320.0M+2172.0S | 4137.6M | 1 |
| 7 | I+2300.0M+2172.0S | 4117.6M | 1 |
| 8 | I+2288.0M+2175.0S | 4108.0M | 1 |

Table B.30: Right-to-Left Windowing Method Average Cost ($d = 224$)

| $w$ | Field Cost | $M$-cost | $s_1$ |
|---|---|---|---|
| 2 | I+911.0M+895.0S | 1707.0M | 111 |
| 3 | I+955.0M+907.0S | 1760.6M | 74 |
| 4 | I+1035.0M+923.0S | 1853.4M | 55 |
| 5 | I+1211.0M+971.0S | 2067.8M | 44 |

Table B.31: Right-to-Left Windowing Method Average Cost ($d = 256$)

| $w$ | Field Cost | $M$-cost | $s_1$ |
|---|---|---|---|
| 2 | I+1039.0M+1023.0S | 1937.4M | 127 |
| 3 | I+1087.0M+1039.0S | 1998.2M | 85 |
| 4 | I+1163.0M+1051.0S | 2083.8M | 63 |
| 5 | I+1351.0M+1111.0S | 2319.8M | 51 |

corresponding to the length of the NIST primes $P224$, $P256$, $P384$ and $P521$. The tables for $d$ chosen to correspond with are similar and can be found in Appendix B.

Tables B.34 to B.37 describe the field costs and $M$-costs of Algorithm 34 with $d$ corresponding to the length of the NIST prime $P224$, $P256$, $P384$ and $P521$.

Tables B.38 to B.41 describe the field costs and $M$-costs of the variant of Algorithm 36 with no recombination stage. For each $d$ corresponding to a NIST prime, $b_{max}, t_{max}, m_{max}$ are chosen optimally. For multipliers of length $d = 224$ the optimal

Table B.32: Right-to-Left Windowing Method Average Cost ($d = 384$)

| $w$ | Field Cost | $M$-cost | $s_1$ |
|---|---|---|---|
| 2 | I+1551.0M+1535.0S | 2859.0M | 191 |
| 3 | I+1591.0M+1543.0S | 2905.4M | 127 |
| 4 | I+1675.0M+1563.0S | 3005.4M | 95 |
| 5 | I+1851.0M+1611.0S | 3219.8M | 76 |

Table B.33: Right-to-Left Windowing Method Average Cost ($d = 521$)

| $w$ | Field Cost | $M$-cost | $s_1$ |
|---|---|---|---|
| 2 | I+2103.0M+2087.0S | 3852.6M | 260 |
| 3 | I+2143.0M+2095.0S | 3899.0M | 173 |
| 4 | I+2235.0M+2123.0S | 4013.4M | 130 |
| 5 | I+2411.0M+2171.0S | 4227.8M | 104 |

Table B.34: Left-to-Right (3 Processors) Average Cost ($d = 224$)

| $l$ | Field Cost | $M$-cost | $s_1$ |
|---|---|---|---|
| 4 | I+1070.3M+958.0S | 1916.7M | 3 |
| 5 | I+1057.0M+955.0S | 1901.0M | 4 |
| 6 | I+1062.3M+964.0S | 1913.5M | 5 |
| 7 | I+1042.3M+946.0S | 1879.1M | 6 |
| 8 | I+1043.7M+946.0S | 1880.5M | 7 |
| 9 | I+1051.7M+951.0S | 1892.5M | 8 |
| 10 | I+1078.3M+973.0S | 1936.7M | 9 |

Table B.35: Left-to-Right (3 Processors) Average Cost ($d = 256$)

| $l$ | Field Cost | $M$-cost | $s_1$ |
|---|---|---|---|
| 4 | I+1219.7M+1094.0S | 2174.9M | 3 |
| 5 | I+1215.7M+1102.0S | 2177.3M | 4 |
| 6 | I+1195.7M+1089.0S | 2146.9M | 5 |
| 7 | I+1195.7M+1091.0S | 2148.5M | 6 |
| 8 | I+1182.3M+1078.0S | 2124.7M | 7 |
| 9 | I+1206.3M+1099.0S | 2165.5M | 8 |
| 10 | I+1206.3M+1096.0S | 2163.1M | 9 |

Table B.36: Left-to-Right (3 Processors) Average Cost ($d = 384$)

| $l$ | Field Cost | $M$-cost | $s_1$ |
|---|---|---|---|
| 6 | I+1755.7M+1614.0S | 3126.9M | 5 |
| 7 | I+1747.7M+1613.0S | 3118.1M | 6 |
| 8 | I+1737.0M+1606.0S | 3101.8M | 7 |
| 9 | I+1747.7M+1617.0S | 3121.3M | 8 |
| 10 | I+1761.0M+1629.0S | 3144.2M | 9 |
| 11 | I+1742.3M+1609.0S | 3109.5M | 10 |
| 12 | I+1742.3M+1606.0S | 3107.1M | 11 |

Table B.37: Left-to-Right (3 Processors) Average Cost ($d = 521$)

| $l$ | Field Cost | $M$-cost | $s_1$ |
|---|---|---|---|
| 6 | I+2369.0M+2189.0S | 4200.2M | 5 |
| 7 | I+2361.0M+2193.0S | 4195.4M | 6 |
| 8 | I+2361.0M+2200.0S | 4201.0M | 7 |
| 9 | I+2327.7M+2172.0S | 4145.3M | 8 |
| 10 | I+2358.3M+2203.0S | 4200.7M | 9 |
| 11 | I+2349.0M+2194.0S | 4184.2M | 10 |
| 12 | I+2350.3M+2194.0S | 4185.5M | 11 |

Table B.38: Double-Base $n$-Chain Average Cost ($d = 224$)

| $N$ | Field Cost | $M$-cost | $s_1$ |
|---|---|---|---|
| 2 | I+1282.0M+889.0S | 2073.2M | 1 |
| 3 | I+1218.0M+865.0S | 1990.0M | 1 |
| 4 | I+1186.0M+853.0S | 1948.4M | 1 |
| 5 | I+1162.0M+844.0S | 1917.2M | 1 |
| 6 | I+1154.0M+841.0S | 1906.8M | 1 |
| 7 | I+1138.0M+835.0S | 1886.0M | 1 |
| 8 | I+1138.0M+835.0S | 1886.0M | 1 |

Table B.39: Double-Base $n$-Chain Average Cost ($d = 256$)

| $N$ | Field Cost | $M$-cost | $s_1$ |
|---|---|---|---|
| 2 | I+1466.0M+1010.0S | 2354.0M | 1 |
| 3 | I+1394.0M+983.0S | 2260.4M | 1 |
| 4 | I+1354.0M+968.0S | 2208.4M | 1 |
| 5 | I+1338.0M+962.0S | 2187.6M | 1 |
| 6 | I+1322.0M+956.0S | 2166.8M | 1 |
| 7 | I+1306.0M+950.0S | 2146.0M | 1 |
| 8 | I+1298.0M+947.0S | 2135.6M | 1 |

value of $(b_{max}, t_{max})$ is $(121, 65)$, for $d = 256$, $(134, 77)$, for $d = 384$, $(216, 106)$ and for $d = 521$, $(269, 159)$ just as in Section 3.2.6, see Table 3.10.

Tables B.42 to B.12 describe the result of Chapter 4 for parameters corresponding to the NIST primes $P224$, $P256$ and $P384$ in terms of their field costs and their $M$-costs.

Table B.40: Double-Base $n$-Chain Average Cost ($d = 384$)

| $N$ | Field Cost | $M$-cost | $s_1$ |
|---|---|---|---|
| 2 | I+2167.0M+1525.0S | 3467.0M | 1 |
| 3 | I+2055.0M+1483.0S | 3321.4M | 1 |
| 4 | I+1999.0M+1462.0S | 3248.6M | 1 |
| 5 | I+1967.0M+1450.0S | 3207.0M | 1 |
| 6 | I+1943.0M+1441.0S | 3175.8M | 1 |
| 7 | I+1927.0M+1435.0S | 3155.0M | 1 |
| 8 | I+1919.0M+1432.0S | 3144.6M | 1 |

Table B.41: Double-Base $n$-Chain Average Cost ($d = 521$)

| $N$ | Field Cost | $M$-cost | $s_1$ |
|---|---|---|---|
| 2 | I+2967.0M+2042.0S | 4680.6M | 1 |
| 3 | I+2815.0M+1985.0S | 4483.0M | 1 |
| 4 | I+2743.0M+1958.0S | 4389.4M | 1 |
| 5 | I+2695.0M+1940.0S | 4327.0M | 1 |
| 6 | I+2663.0M+1928.0S | 4285.4M | 1 |
| 7 | I+2647.0M+1922.0S | 4264.6M | 1 |
| 8 | I+2631.0M+1916.0S | 4243.8M | 1 |

Table B.42: Parallel Scalar Multiplication ($P224$)

| Processors | Algorithm | Variables | Field Cost | $M$-cost | $s_1$ |
|---|---|---|---|---|---|
| 2 | $p^{th}$ (28) | – | I+1200.7M+1008.5S | 2087.5M | 3 |
| | R2L Par. (29) | – | I+1194.7M+1005.0S | 2078.7M | 1 |
| | 2P R2L (31) | – | I+905.0M+897.0S | 1702.6M | 74 |
| | 2P Win R2L (32) | $w = 4$ | I+1035.0M+923.0S | 1853.4M | 55 |
| | | $w = 5$ | I+1211.0M+971.0S | 2067.8M | 44 |
| | 2P L2R (35) | $v = 3, w = 4$ | 2I+1074.0M+956.0S | 1998.8M | 18 |
| | | $v = 4, w = 5$ | 2I+1034.5M+938.8S | 1945.6M | 14 |
| | | $v = 5, w = 6$ | 2I+1039.7M+934.9S | 1947.6M | 13 |
| | DB2Chain (36) | – | I+1282.0M+889.0S | 2073.2M | 1 |
| | MontLad (37) | – | I+1569.0M+670.0S | 2185.0M | 446 |
| 3 | $p^{th}$ (28) | – | I+1114.4M+975.7S | 1975.0M | 5 |
| | R2L Par. (29) | – | I+1100.0M+972.0S | 1957.6M | 1 |
| | 3P L2R (34) | $l = 4$ | I+1070.3M+958.0S | 1916.7M | 3 |
| | | $l = 5$ | I+1057.0M+955.0S | 1901.0M | 4 |
| | | $l = 6$ | I+1062.3M+964.0S | 1913.5M | 5 |
| | | $l = 7$ | I+1042.3M+946.0S | 1879.1M | 6 |
| | DB3Chain (36) | – | I+1218.0M+865.0S | 1990.0M | 1 |
| 4 | $p^{th}$ (28) | – | I+1065.3M+957.2S | 1911.1M | 5 |
| | R2L Par. (29) | – | I+1045.3M+949.0S | 1884.5M | 1 |
| | DB4Chain (36) | – | I+1186.0M+853.0S | 1948.4M | 1 |
| 5 | $p^{th}$ (28) | – | I+1047.9M+950.2S | 1888.0M | 7 |
| | R2L Par. (29) | – | I+1020.0M+942.0S | 1853.6M | 1 |
| | DB5Chain (36) | – | I+1162.0M+844.0S | 1917.2M | 1 |
| 6 | $p^{th}$ (28) | – | I+1028.2M+942.8S | 1862.5M | 7 |
| | R2L Par. (29) | – | I+1013.3M+947.0S | 1850.9M | 1 |
| | DB6Chain (36) | – | I+1154.0M+841.0S | 1906.8M | 1 |
| 7 | $p^{th}$ (28) | – | I+1014.2M+937.6S | 1844.2M | 7 |
| | R2L Par. (29) | – | I+981.3M+925.0S | 1801.3M | 1 |
| | DB7Chain (36) | – | I+1138.0M+835.0S | 1886.0M | 1 |
| 8 | $p^{th}$ (28) | – | I+1003.7M+933.6S | 1830.6M | 7 |
| | R2L Par. (29) | – | I+970.7M+921.0S | 1787.5M | 1 |
| | R2L Hed. (30) | – | I+905.0M+897.0S | 1702.6M | 1 |
| | DB8Chain (36) | – | I+1138.0M+835.0S | 1886.0M | 1 |

Table B.43: Parallel Scalar Multiplication ($P256$)

| Processors | Algorithm | Variables | Field Cost | $M$-cost | $s_1$ |
|---|---|---|---|---|---|
| 2 | $p^{th}$ (28) | – | I+1371.3M+1152.5S | 2373.3M | 3 |
| | R2L Par. (29) | – | I+1365.3M+1149.0S | 2364.5M | 1 |
| | 2P R2L (31) | – | I+1033.0M+1025.0S | 1933.0M | 84 |
| | 2P Win R2L (32) | $w = 4$ | I+1163.0M+1051.0S | 2083.8M | 63 |
| | | $w = 5$ | I+1351.0M+1111.0S | 2319.8M | 51 |
| | 2P L2R (35) | $v = 3, w = 4$ | 2I+1226.0M+1092.0S | 2259.6M | 21 |
| | | $v = 4, w = 5$ | 2I+1177.9M+1072.0S | 2195.4M | 15 |
| | | $v = 5, w = 6$ | 2I+1178.3M+1066.4S | 2191.5M | 14 |
| | DB2Chain (36) | – | I+1466.0M+1010.0S | 2354.0M | 1 |
| | MontLad (37) | – | I+1793.0M+766.0S | 2485.8M | 510 |
| 3 | $p^{th}$ (28) | – | I+1270.9M+1114.3S | 2242.4M | 5 |
| | R2L Par. (29) | – | I+1261.3M+1115.0S | 2233.3M | 1 |
| | 3P L2R (34) | $l = 4$ | I+1219.7M+1094.0S | 2174.9M | 3 |
| | | $l = 6$ | I+1195.7M+1089.0S | 2146.9M | 5 |
| | | $l = 8$ | I+1182.3M+1078.0S | 2124.7M | 7 |
| | DB3Chain (36) | – | I+1394.0M+983.0S | 2260.4M | 1 |
| 4 | $p^{th}$ (28) | – | I+1214.7M+1093.2S | 2169.3M | 5 |
| | R2L Par. (29) | – | I+1194.7M+1085.0S | 2142.7M | 1 |
| | DB4Chain (36) | – | I+1354.0M+968.0S | 2208.4M | 1 |
| 5 | $p^{th}$ (28) | – | I+1192.9M+1084.6S | 2140.6M | 7 |
| | R2L Par. (29) | – | I+1178.7M+1089.0S | 2129.9M | 1 |
| | DB5Chain (36) | – | I+1338.0M+962.0S | 2187.6M | 1 |
| 6 | $p^{th}$ (28) | – | I+1170.4M+1076.2S | 2111.4M | 7 |
| | R2L Par. (29) | – | I+1146.7M+1072.0S | 2084.3M | 1 |
| | DB6Chain (36) | – | I+1322.0M+956.0S | 2166.8M | 1 |
| 7 | $p^{th}$ (28) | – | I+1154.4M+1070.1S | 2090.5M | 7 |
| | R2L Par. (29) | – | I+1134.7M+1070.0S | 2070.7M | 1 |
| | DB7Chain (36) | – | I+1306.0M+950.0S | 2146.0M | 1 |
| 8 | $p^{th}$ (28) | – | I+1142.3M+1065.6S | 2074.8M | 7 |
| | R2L Par. (29) | – | I+1109.3M+1053.0S | 2031.7M | 1 |
| | R2L Hed. (30) | – | I+1033.0M+1025.0S | 1933.0M | 1 |
| | DB8Chain (36) | – | I+1298.0M+947.0S | 2135.6M | 1 |

Table B.44: Parallel Scalar Multiplication ($P384$)

| Processors | Algorithm | Variables | Field Cost | $M$-cost | $s_1$ |
|---|---|---|---|---|---|
| 2 | $p^{th}$ (28) | – | I+2054.0M+1728.5S | 3516.8M | 3 |
| | R2L Par. (29) | – | I+2048.0M+1725.0S | 3508.0M | 1 |
| | 2P R2L (31) | – | I+1545.0M+1537.0S | 2854.6M | 127 |
| | 2P Win R2L (32) | $w = 4$ | I+1675.0M+1563.0S | 3005.4M | 95 |
| | | $w = 5$ | I+1851.0M+1611.0S | 3219.8M | 76 |
| | 2P L2R (35) | $v = 3, w = 4$ | 2I+1834.0M+1636.0S | 3302.8M | 29 |
| | | $v = 4, w = 5$ | 2I+1751.3M+1604.4S | 3194.9M | 20 |
| | | $v = 5, w = 6$ | 2I+1733.0M+1592.7S | 3167.1M | 17 |
| | DB2Chain (36) | – | I+2167.0M+1525.0S | 3467.0M | 1 |
| | MontLad (37) | – | I+1793.0M+766.0S | 3689.0M | 510 |
| 3 | $p^{th}$ (28) | – | I+1896.7M+1669.0S | 3311.9M | 5 |
| | R2L Par. (29) | – | I+1877.3M+1661.0S | 3286.1M | 1 |
| | 3P L2R (34) | $l = 6$ | I+1755.7M+1614.0S | 3126.9M | 5 |
| | | $l = 7$ | I+1747.7M+1613.0S | 3118.1M | 6 |
| | | $l = 8$ | I+1737.0M+1606.0S | 3101.8M | 7 |
| | DB3Chain (36) | – | I+2055.0M+1483.0S | 3321.4M | 1 |
| 4 | $p^{th}$ (28) | – | I+1812.0M+1637.2S | 3201.8M | 5 |
| | R2L Par. (29) | – | I+1792.0M+1629.0S | 3175.2M | 1 |
| | DB4Chain (36) | – | I+1999.0M+1462.0S | 3248.6M | 1 |
| 5 | $p^{th}$ (28) | – | I+1773.2M+1622.2S | 3151.0M | 7 |
| | R2L Par. (29) | – | I+1745.3M+1614.0S | 3116.5M | 1 |
| | DB5Chain (36) | – | I+1967.0M+1450.0S | 3207.0M | 1 |
| 6 | $p^{th}$ (28) | – | I+1739.3M+1609.5S | 3106.9M | 7 |
| | R2L Par. (29) | – | I+1706.7M+1597.0S | 3064.3M | 1 |
| | DB6Chain (36) | – | I+1943.0M+1441.0S | 3175.8M | 1 |
| 7 | $p^{th}$ (28) | – | I+1715.1M+1600.4S | 3075.5M | 7 |
| | R2L Par. (29) | – | I+1686.7M+1592.0S | 3040.3M | 1 |
| | DB7Chain (36) | – | I+1927.0M+1435.0S | 3155.0M | 1 |
| 8 | $p^{th}$ (28) | – | I+1697.0M+1593.6S | 3051.9M | 7 |
| | R2L Par. (29) | – | I+1664.0M+1581.0S | 3008.8M | 1 |
| | R2L Hed. (30) | – | I+1545.0M+1537.0S | 2854.6M | 1 |
| | DB8Chain (36) | – | I+1919.0M+1432.0S | 3144.6M | 1 |