High-Speed Elliptic Curve and Pairing-Based Cryptography

Thesis · April 2011

1 author:

Patrick Longa
Microsoft
45 PUBLICATIONS **467** CITATIONS

SEE PROFILE

High-Speed Elliptic Curve and Pairing-Based Cryptography

by

Patrick Longa

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2011

AUTHOR'S DECLARATION

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Elliptic Curve Cryptography (ECC), independently proposed by Miller [Mil86] and Koblitz [Kob87] in mid 80's, is finding momentum to consolidate its status as the public-key system of choice in a wide range of applications and to further expand this position to settings traditionally occupied by RSA and DL-based systems. The non-existence of known subexponential attacks on this cryptosystem directly translates to shorter keylengths for a given security level and, consequently, has led to implementations with better bandwidth usage, reduced power and memory requirements, and higher speeds. Moreover, the dramatic entry of pairing-based cryptosystems defined on elliptic curves at the beginning of the new millennium has opened the possibility of a plethora of innovative applications, solving in some cases longstanding problems in cryptography. Nevertheless, public-key cryptography (PKC) is still relatively expensive in comparison with its symmetric-key counterpart and it remains an open challenge to reduce further the computing cost of the most time-consuming PKC primitives to guarantee their adoption for secure communication in commercial and Internet-based applications. The latter is especially true for pairing computations. Thus, it is of paramount importance to research methods which permit the efficient realization of Elliptic Curve and Pairing-based Cryptography on the several new platforms and applications.

This thesis deals with efficient methods and explicit formulas for computing elliptic curve scalar multiplication and pairings over fields of large prime characteristic with the objective of enabling the realization of software implementations at very high speeds.

To achieve this main goal in the case of elliptic curves, we accomplish the following tasks:

identify the elliptic curve settings with the fastest arithmetic; accelerate the precomputation stage in the scalar multiplication; study number representations and scalar multiplication algorithms for speeding up the evaluation stage; identify most efficient field arithmetic algorithms and optimize them; analyze the architecture of the targeted platforms for maximizing the performance of ECC operations; identify most efficient coordinate systems and optimize explicit formulas; and realize implementations on x86-64 processors with an optimal algorithmic selection among all studied cases.

In the case of pairings, the following tasks are accomplished: accelerate tower and curve arithmetic; identify most efficient tower and field arithmetic algorithms and optimize them; identify the curve setting with the fastest arithmetic and optimize it; identify state-of-the-art techniques for the Miller loop and final exponentiation; and realize an implementation on x86-64 processors with optimal algorithmic selection.

The most outstanding contributions that have been achieved with the methodologies above in this thesis can be summarized as follows:

- *Two* novel precomputation schemes are introduced and shown to achieve the lowest costs in the literature for different curve forms and scalar multiplication primitives. The detailed cost formulas of the schemes are derived for most relevant scenarios.
- A new methodology based on the *operation cost per bit* to devise highly optimized and compact multibase algorithms is proposed. Derived multibase chains using bases {2,3} and {2,3,5} are shown to achieve the lowest theoretical costs for scalar multiplication on certain curve forms and for scenarios with and without precomputations. In addition, the zero and nonzero density formulas of the original (width-w) multibase NAF method are derived by using Markov chains. The application of "fractional" windows to the multibase method is described together with the derivation of the corresponding density formulas.
- Incomplete reduction and branchless arithmetic techniques are optimally combined for devising high-performance field arithmetic. Efficient algorithms for "small" modular operations using suitably chosen pseudo-Mersenne primes are carefully analyzed and optimized for incomplete reduction.
- Data dependencies between contiguous field operations are discovered to be a source of performance degradation on x86-64 processors. *Three* techniques for reducing the number of potential pipeline stalls due to these dependencies are proposed: field arithmetic scheduling, merging of point operations and merging of field operations.
- Explicit formulas for *two* relevant cases, namely Weierstrass and Twisted Edwards curves over $\mathbb{F}_p$ and $\mathbb{F}_{p^2}$, are carefully optimized employing incomplete reduction, minimal number of operations and reduced number of data dependencies between

contiguous field operations.

- Best algorithms for the field, point and scalar arithmetic, studied or proposed in this thesis, are brought together to realize *four* high-speed implementations on x86-64 processors at the 128-bit security level. Presented results set *new speed records* for elliptic curve scalar multiplication and introduce up to 34% of cost reduction in comparison with the best previous results in the literature.
- A *generalized lazy reduction technique* that enables the elimination of up to 32% of modular reductions in the pairing computation is proposed. Further, a methodology that keeps intermediate results under Montgomery reduction boundaries maximizing operations without carry checks is introduced. Optimized formulas for the popular tower $\mathbb{F}_p \rightarrow \mathbb{F}_{p^2} \rightarrow \mathbb{F}_{p^6} \rightarrow \mathbb{F}_{p^{12}}$ are explicitly stated and a detailed operation count that permits to determine the theoretical cost improvement attainable with the proposed method is carried out for the case of an optimal ate pairing on a Barreto-Naehrig (BN) curve at the 128-bit security level.
- Best algorithms for the different stages of the pairing computation, including the proposed techniques and optimizations, are brought together to realize a high-speed implementation at the 128-bit security level. Presented results on x86-64 processors set *new speed records* for pairings, introducing up to 34% of cost reduction in comparison with the best published result.

From a general viewpoint, the proposed methods and optimized formulas have a practical impact in the performance of cryptographic protocols based on elliptic curves and pairings in a wide range of applications. In particular, the introduced implementations represent a direct and significant improvement that may be exploited in performance-dominated applications such as high-demand Web servers in which millions of secure transactions need to be generated.

Acknowledgements

This Ph.D. thesis would not have been possible without the support and encouragement of many people. My thanks go first to my supervisor, Dr. Catherine Gebotys, for her invaluable support and guidance during all my Ph.D. studies.

I am also grateful to all my professors at the University of Waterloo, especially to Dr. Anwar Hasan and Dr. David Jao for providing me with very useful feedback and comments on my preliminary technical reports that later became part of Chapter 4, and to Dr. Hiren Patel for his useful feedback on my research about efficient ECC implementation that later became part of Chapter 5.

I would like to thank my committee members, Dr. Gordon Agnew, Dr. Anwar Hasan, Dr. Michael Scott and Dr. Doug Stinson, for taking the time for reading this thesis and providing many useful suggestions that helped me improve this work.

My thanks go to Dr. Michael Scott for his valuable help and feedback when I was developing the ECC implementations presented in Chapter 5 on top of the MIRACL crypto library that he developed; and to Dr. Huseyin Hisil for very valuable discussions on elliptic curves. Works from both authors have been an inspiration and the basis for several developments in this thesis.

I would like to thank Diego F. Aranha, Dr. Catherine Gebotys, Dr. Koray Karabina and Dr. Julio Lopez for our joint work on pairings [AKL+10], which is part of Chapter 6.

Special thanks go to Diego F. Aranha for his friendship, our always interesting discussions on

cryptography and its efficient implementation, and our joint effort to develop the pairing implementation presented in Chapter 6.

My thanks go to Tom St Denis, Diego F. Aranha and Dr. Colin Walter for providing valuable comments for several sections of this thesis.

I would like to thank my colleagues in the Laboratory for Side-Channel Security of Embedded Systems at the University of Waterloo, including Dr. Solmaz Ghaznavi, Farhad Haghighizadeh, Marcio Juliato, David Kenney, Dr. Amir Khatibzadeh and Edgar Mateos, for their friendship and our very interesting discussions that made my studies more enjoyable and productive.

This work would not have been possible without the financial support of the NSERC Alexander Graham Bell Canada Graduate Scholarship – Doctoral (CGS-D) and the University of Waterloo President's Graduate Scholarship. Also, many test results presented in Chapters 5 and 6 were obtained using the facilities of the Shared Hierarchical Academic Research Computing Network (SHARCNET) and Compute/Calcul Canada. My sincere gratitude goes to all of them.

Last but not least, I am profoundly grateful to my mother, Patricia Pierola, and my brothers, Patricia and Franccesco Longa, for their love and unconditional encouragement. Especially, I dedicate this work to my wife, Veronica Zeballos, and my daughter, Adriana Longa, because their infinite love, support, patience and faith in my work were the ones that actually made this thesis possible.

To my wife and daughter,
Veronica and Adriana,
my lights in this World

Table of Contents

Appendices

List of Tables

List of Algorithms

List of Acronyms

a, m, s, r, i	Addition, multiplication, squaring, reduction and inversion over $\mathbb{F}_{p^2}$
A, D, M, S, R, I	Addition, multiplication by curve constant, multiplication, squaring, reduction and inversion over $\mathbb{F}_p$
$\mathcal{A}$, $\mathcal{C}$, $\mathcal{J}$, $\mathcal{J}^m$, $\mathcal{H}$, $\mathcal{LD}$	Affine, Chudnovsky, Jacobian, modified Jacobian, homogeneous and Lopez-Dahab coordinates
AES	Advanced Encryption Standard algorithm
AVX	Advanced Vector Extensions
BDHP	Bilinear Diffie-Hellman Problem
BN	Barreto-Naehrig curve
CBEA	Cell Broadband Engine Architecture
DBL, TPL, QPL, ADD	Point doubling, tripling, quintupling and addition
DBLADD	Point doubling-addition
DBNS	Double-Base Number System
DHP	Diffie-Hellman Problem
DL	Discrete Logarithm
DLP	Discrete Logarithm Problem
DOS	Dahmen-Okeya-Schepers precomputation method
ECC	Elliptic Curve Cryptography
ECDH	Elliptic Curve Diffie-Hellman key exchange
ECDLP	Elliptic Curve Discrete Logarithm Problem
ECDSA	Elliptic Curve Digital Signature Algorithm
Frac-*wmb*NAF	Fractional Width-*w* Multibase Non-Adjacent Form method

Frac-*w*NAF	Fractional Width-*w* Non-Adjacent Form method
GLS	Galbraith-Lin-Scott method
GLV	Gallant-Lambert-Vanstone method
GPR	General Purpose Register
GPU	Graphical Processing Unit
HECC	Hyperelliptic Curve Cryptosystem
IBE	Identity-Based Encryption
$\mathcal{IE}$, $\mathcal{E}$, $\mathcal{E}^e$	Inverted Edwards, Twisted Edwards homogeneous and extended Twisted Edwards homogeneous coordinates
ILP	Instruction-Level Parallelism
IR	Incomplete reduction
ISA	Instruction Set Architecture
$\mathcal{JQ}^e$	Extended Jacobi quartic coordinates
JSF	Joint Sparse Form method
LG	Longa-Gebotys precomputation scheme
LM	Longa-Miri precomputation scheme
NFS	Number Field Sieve
NIST	U.S. National Institute of Standards and Technology
NSA	U.S. National Security Agency
PKC	Public-Key Cryptography
RAW	Read-After-Write or true data dependence
RSA	Rivest-Shamir-Adleman cryptosystem
SCA	Side-Channel Analysis attack
SIMD	Single Instruction, Multiple Data
SSE2	Streaming SIMD Extensions 2
(*w*)*mb*NAF	(Width-*w*) Multibase Non-Adjacent Form method
(*w*)NAF	(Width-*w*) Non-Adjacent Form method

Chapter 1

Introduction

1.1. Motivation

Since its discovery by Diffie and Hellman in 1976 [DH76], public-key cryptography (PKC) has revolutionized the way communications are securely achieved by governments, banks, enterprises and even plain people. Based on clever mathematical constructs, public-key systems appeared to alleviate the difficult problem of key management and distribution, and provide such powerful tools as digital signatures. See, for example, [HMV04, Section 1.2] or [ACD+05, Section 1] for an introduction to PKC.

Nonetheless, RSA, the dominant public-key system during many years, and discrete logarithm (DL)-based cryptosystems are already exhibiting clear limitations to keep an acceptable performance level in the plethora of new applications and platforms in the new millennium that range from constrained, power-limited wireless devices [BCH+00, Lau04] to cluster servers performing millions of secure transactions for e-commerce and e-banking [GGC02, GSF04]. A relatively new, more "compact" player in the public-key crypto arena has been gaining increasing attention in academia and commercial applications: elliptic curve cryptosystems.

Elliptic Curves for Cryptography

The complex and elegant mathematics behind elliptic curves have attracted number theorists and

algebra geometers long time before the remarkable work by Lenstra [Len87] using elliptic curves for factoring led to the independent discovery by Miller [Mil86] and Koblitz [Kob87] of Elliptic Curve Cryptography (ECC) in 1985. Since then, with the exception of some studies that found vulnerabilities in certain special curves [MOV93, Sma99], it has not been possible to find better attacks than Pollard's rho [Pol78], which runs in exponential time, for elliptic curves with large prime order subgroup. As a consequence, elliptic curve cryptosystems require shorter keys to attain a certain security level in comparison with those required by the traditional RSA and DL-based systems. For instance, to achieve a level of security equivalent to the Advanced Encryption Standard algorithm with 256 bits (AES-256), the National Institute of Standards and Technology (NIST) recommends the use of ECC keys of 512 bits, whereas RSA would require keylengths of more than 15000 bits [NIST07]. This significant difference in favour of ECC has led in many scenarios to faster, more power-efficient and/or memory-friendly implementations, which make this cryptosystem especially attractive for constrained devices such as wireless sensor nodes, smartcards, personal digital assistants (PDAs), cellphones, smartphones, and many others. Moreover, the superior speed of ECC over RSA supports the improvement of performance of Web servers in which public-key transactions may be a bottleneck, thus enabling the use of strong cryptography on a wider range of Internet-based applications [GSF04].

A clear example of the importance of ECC in future commercial and governmental applications has been set by the inclusion of ECC primitives in the U.S. National Security Agency (NSA) Suite B Cryptography, which contains a set of recommended algorithms for classified and unclassified U.S. security systems and information [NSA09]. In particular, the Elliptic Curve Digital Signature (ECDSA) algorithm and the Elliptic Curve Diffie-Hellman (ECDH) key exchange over prime fields (see §2.2.3) are recommended in Suite B for providing security up to top secret level. Hence, ECC is arguably getting positioned as the dominant public-key system in many applications, and is expected to occupy that privileged position for several years to come. As direct consequence of this technological shift, the efficient implementation of ECC schemes in software and hardware platforms is gaining key importance to realize strong cryptography.

In that direction, this thesis deals with the fast and efficient computation of elliptic curve scalar multiplication. This critical operation, denoted by kP (where k is a scalar and P a point on an elliptic curve), is the central and most time-consuming operation in ECC. Although several methods to compute kP efficiently have been proposed and extensively studied in past years, it is still a very interesting challenge to improve further the performance of this operation. Elliptic curve scalar multiplication comprises *three* arithmetic layers: field arithmetic, point arithmetic and scalar arithmetic. Cryptographic protocols and schemes work on top of these layers; see Section 2.2.3 and [HMV04, Chapter 4] for an overview. In this thesis, we focus on improving the overall computation at all three arithmetic layers to try to achieve the *highest speed possible* in

software. In this effort we follow the next steps: (i) identify the elliptic curve settings with the fastest arithmetic; (ii) accelerate the precomputation stage of scalar multiplication; (iii) study number representations and scalar multiplication algorithms for speeding up the evaluation stage; (iv) identify most efficient field arithmetic algorithms and optimize them; (v) analyze the architecture of the targeted platforms for maximizing the performance of ECC operations; (vi) identify most efficient coordinate systems and optimize explicit formulas; and (vii) realize implementations on x86-64 processors with an optimal algorithmic selection among all studied cases.

Grouping together the steps above, let us consider in greater detail the most relevant problems and aspects that are considered in this study.

Precomputation Stage: step (ii)

A practical strategy that reduces the number of required operations at the expense of some extra memory is the use of precomputations. In this case, a table of points is built and stored in advance (*precomputation stage*) for later use during the execution of the scalar multiplication itself (*evaluation stage*). The effect of computing these additional points in the overall cost basically depends on the context in which the scalar multiplication occurs. In [HMV04], Hankerson et al. distinguishes two possible scenarios that depend on the prior knowledge of the initial point P, and classifies the different methods for scalar multiplication according to them.

Let us illustrate both scenarios, and their subtleties, in the context of the ECDH key exchange (see Section 2.2.3): when each Bob and Alice computes the initial scalar multiplication using a random scalar in the first phase of the ECDH scheme, both use a publicly known point P for the computation. Because P is available beforehand, it is obvious that methods that extensively exploit precomputations to reduce the cost of the evaluation stage are preferable in this scenario. Examples of efficient methods in this case are comb methods [HMV04, Section 3.3.2]. On the other hand, during the second phase of the ECDH scheme, Bob and Alice exchange the results from the first phase and calculate a new scalar multiplication. This time, however, the results (which are also points on the curve) are not known in advance by their corresponding receptors. Although methods may still exploit precomputations, this time the overall cost includes the costs of both the precomputation and evaluation stages. A well-known method in this case is width-w NAF (wNAF) [Sol00], which is the windowed version of the standard non-adjacent form (NAF).

Scalar Representation in the Evaluation Stage: step (iii)

The cost of the evaluation stage in the computation kP is strongly tied to the representation used for the scalar k. With the exception of Montgomery's method [Mon87], the most popular approach has been the use of the NAF or wNAF representation in combination with some version

of the *double-and-add* algorithm (see Section 2.2.4.3). However, recently there has been an increased interest in using novel arithmetic representations of integers based on double- and multi-base number systems [DJM98, DIM05]. In general, it has been observed that these representations enable a reduction in the number of point operations required for computing *kP*. However, it is still an open question to determine up to what extent and in which scenarios the new multibase representations reduce the computational cost of scalar multiplication. It has been shown that these methods in fact reduce the number of point operations but in exchange they require more complex formulas besides point doubling and addition. Partially, the question above could be answered by trying to find the "optimal" (or close to "optimal") multibase representation of a given scalar for a particular setting, where "optimal" is defined here as relative to the computational cost and not to the minimization of the number of additions.

Efficient Implementation on x86-64 Processors: steps (iv)-(vii)

Over the years, many efforts have focused on efficient implementation of ECC primitives on different platforms [BHL+01, GPW+04, GAS+05, Ber06, CS09]. An incomplete list includes the analysis on 8-bit microcontrollers, 32-bit embedded devices, graphical processing units, processors based on the x86 Instruction Set Architecture (ISA) or the cell broadband engine architecture, among many others. At a high-level, these works provide two main contributions:

- The compilation of state-of-the-art algorithms and their efficient combination trying to achieve the highest performance possible on the targeted platforms.
- The publication of benchmark results that illustrate the potential performance achievable by the particular cryptographic primitive on the targeted platforms.

As a side-effect, when different test results are made available, readers learn from direct comparisons among alternative methods or algorithms.

Processors based on the x86-64 ISA [AMD] have become increasingly popular in the last few years and are now being extensively used for notebook, desktop and server computers. Hence, efficient cryptographic computation on these processors is of paramount importance to realize strong cryptography in a wide variety of applications. Relevant questions are then: what are the methods, formulas and parameters that once combined achieve the highest performance for computing ECC primitives on these processors? and what are the features of these devices that can be exploited to gain (or sometimes, not to lose) performance? It is then obvious that, for best results, the analysis should contemplate architectural features of the processors under analysis.

Elliptic Curve Forms: step (i)

Elliptic curves over prime fields have been traditionally represented in its short Weiertrass form, $y^2 = x^3 + ax + b$, where $a,b \in \mathbb{F}_p$. More specifically, the projective form of this curve equation using Jacobian coordinates has been the preferred elliptic curve shape for many years by most implementers and standardization bodies such as NIST and IEEE [NIST00, NIST09, IEEE00]. However, in the last few years intense research has been working on new and improved curve forms. Although these curves have not been standardized by national/international bodies up to date, they provide attractive advantages such as faster arithmetic and/or higher resilience against certain side-channel analysis (SCA) attacks [Sma01, BJ03b, BL07]. Since in this thesis we are particularly interested in high-speed cryptography, we focus on *two* curve forms that currently exhibit the lowest point operation costs: extended Jacobi quartic form, $y^2 = dx^4 + 2ax^2 + 1$, $a,d \in \mathbb{F}_p$; and Twisted Edwards form, $ax^2 + y^2 = 1 + dx^2y^2$, $a,d \in \mathbb{F}_p$. For each case, we consider in our analysis and implementations the coordinate system(s) and curve parameters that in our experience provide the highest performance (see Section 2.2.5 for further details):

- Mixed homogeneous/extended homogeneous coordinates for the Twisted Edwards curve $ax^2 + y^2 = 1 + dx^2y^2$ with $a = -1$ [HWC+08, His10].
- Inverted Edwards coordinates for the Twisted Edwards curve $ax^2 + y^2 = 1 + dx^2y^2$ with $a = 1$ [BL07b].
- Extended Jacobi quartic coordinates for the extended Jacobi quartic curve $y^2 = dx^4 + 2ax^2 + 1$ with $d = 1$ [HWC+07, HWC+08b].

We also include the short Weierstrass form because of its widespread use in practice:

- Jacobian coordinates for the short Weierstrass form $y^2 = x^3 + ax + b$ with $a = -3$.

Pairing-Based Cryptography

Since Boneh and Franklin [BF01], following pioneering works by several authors [Jou00, SOK00, Ver01], formalized the use of pairings based on elliptic curves with the introduction of Identity-Based Encryption (IBE) in 2001, the interest of cryptographers and implementers in this new research area have grown dramatically. This is mainly due to the potential of pairings for elegantly solving many open problems in cryptography such as Identity-Based Encryption [BF01], short signatures [BLS04], multi-party key agreements [Jou00], among many others. See, for example, [Men09] for an introduction to pairing-based cryptography.

Nevertheless, the pairing computation, which is the central and most time-consuming operation in most pairing-based schemes, is still relatively expensive in comparison with ECC

operations (e.g., an elliptic curve scalar multiplication is about *ten* times faster than a pairing computation at the 128-bit security level on x86-64 processors [BGM+10, GLS09]). Hence, the development of techniques and methods leading to optimization of the pairing computation are of great importance. Given the technological shift to x86-64-based processors, a series of efforts have recently developed faster pairing implementations targeting these platforms [HMS08, NNS10, BGM+10]. However, it remains a challenging effort to try to optimize further this crucial operation for incentivizing the adoption of these elegant cryptosystems in commercial applications.

In this thesis, we focus on improving the overall pairing computation to try to achieve the *highest speed possible* in software. In this effort we follow the next steps: (i) accelerate tower and curve arithmetic; (ii) identify most efficient tower and field arithmetic algorithms and optimize them; (iii) identify elliptic curve setting with the fastest arithmetic and optimize it; (iii) identify state-of-the-art techniques for the Miller loop and final exponentiation; and (iv) realize implementation on x86-64 processors with an optimal algorithmic selection.

1.2. Contributions

In this thesis, we propose efficient methods and optimized explicit formulas for accelerating the computation of elliptic curve scalar multiplication and pairings on ordinary curves over prime fields. In many cases, the improvements are generic and apply to different types of (hardware and software) platforms.

Our main contributions can be summarized as follows:

- At the precomputation stage, we propose *two* innovative low-cost precomputation schemes. The first scheme, intended for standard curves using Jacobian coordinates, is based on a special addition formula due to Meloni [Mel07]. The second scheme, especially effective for some special curves and multiple scalar multiplication methods such as the Joint Sparse Form (JSF) [Sol01], is based on the concept of *conjugate addition in projective coordinates*. We provide the theoretical costs for single and multiple scalar multiplications and perform an extensive comparative analysis for *three* specific systems: Jacobian, extended Jacobi quartic and inverted Edwards coordinates.
- At the evaluation stage, we provide the theoretical cost analysis of the multibase NAF representation and its windowed variant [Lon07], adapt the concept of "fractional" windows [Möl03] to width-w multibase NAF to obtain a more generic method that allows choosing a flexible number of precomputed points, and introduce a method for deriving high-performance multibase algorithms based on the *operation cost per bit* that apply to a wide set of scenarios, ranging from very constrained environments to

applications where memory is not scarce. An extensive comparison with other works is performed on curves using Jacobian, extended Jacobi quartic and inverted Edwards coordinates at different security levels. A relevant comparison with the fastest curves using radix-2 methods is presented and demonstrates that "slower" curves employing *refined multibase chains* become competitive for suitably chosen curve parameters on memory-constrained devices.

- We bring together the most efficient ECC algorithms for performing elliptic curve scalar multiplication on x86-64 processors and optimize them using techniques from computer architecture. We study the optimal combination of incomplete reduction technique and elimination of conditional branches to achieve high-speed field arithmetic over $\mathbb{F}_p$ using a pseudo-Mersenne prime. We also demonstrate the high penalty incurred by data dependencies between instructions in neighbouring field operations. *Three* generic techniques are proposed to minimize the number of pipeline stalls due to true data dependencies and to reduce the number of function calls and memory accesses. Further, explicit formulas are optimized by minimizing the number of "small" field operations, which are not inexpensive on the targeted platforms. Improved explicit formulas exploiting incomplete reduction and exhibiting minimal number of operations and reduced number of data dependencies between contiguous field operations are derived and explicitly stated for Jacobian coordinates and mixed Twisted Edwards homogeneous/extended homogeneous coordinates for *two* cases: with and without using the GLS method [GLS09]. Record-breaking implementations demonstrating the significant performance improvements obtained with the optimizations and techniques under analysis at the 128-bit security level are described. Benchmark results for different x86-64 processors exhibiting up to 34% cost reduction in comparison with the best published results are presented.
- We introduce a *generalized lazy reduction technique* that allows us to eliminate up to 32% of the total number of modular reductions when applied to the towering and curve arithmetic in the pairing computation. Furthermore, we present a methodology to keep intermediate results under Montgomery reduction boundaries so that the number of operations without carry checks is maximized. We illustrate the method with the well-known tower $\mathbb{F}_p \rightarrow \mathbb{F}_{p^2} \rightarrow \mathbb{F}_{p^6} \rightarrow \mathbb{F}_{p^{12}}$, for which case we explicitly state the improved formulas. Curve arithmetic using Jacobian and homogeneous coordinates is optimized using the projective equivalence class and with the application of lazy reduction. A detailed operation count that allows us to determine the theoretical cost improvement attainable with the proposed method is carried out for the case of an optimal ate pairing on a BN curve [BN05] at the 128-bit security level. To illustrate the practical performance boost obtained with the new formulas we realize a record-breaking

implementation of the pairing above, also incorporating state-of-the-art techniques. Benchmark results for different x86-64 processors exhibiting up to 34% cost reduction in comparison with the best published results in the literature are presented.

The details above only highlight the most relevant contributions of this thesis. The reader is referred to Chapters 3, 4, 5 and 6 for additional outcomes.

Partial results that have been developed further in this thesis already appear in the following relevant publications:

[1] "New Composite Operations and Precomputation Scheme for Elliptic Curve Cryptosystems over Prime Fields", with A. Miri. In *Proc. Int. Conference on Practice and Theory in Public Key Cryptography (PKC 2008)*, 2008. This corresponds to part of Chapter 3.

[2] "Novel Precomputation Schemes for Elliptic Curve Cryptosystems", with C. Gebotys. In *Proc. Int. Conference on Applied Cryptography and Network Security (ACNS 2009)*, 2009. This corresponds to part of Chapter 3.

[3] "Fast Multibase Methods and Other Several Optimizations for Elliptic Curve Scalar Multiplication", with C. Gebotys. In *Proc. Int. Conference on Practice and Theory in Public Key Cryptography (PKC 2009)*, 2009. This corresponds to Chapter 4.

[4] "Efficient Techniques for High-Speed Elliptic Curve Cryptography", with C. Gebotys. In *Proc. Workshop on Cryptographic Hardware and Embedded Systems (CHES 2010)*, 2010. This corresponds to Chapter 5.

[5] "Faster Explicit Formulas for Computing Pairings over Ordinary Curves", with D.F. Aranha, K. Karabina, C. Gebotys and J. Lopez. In *Proc. Advances in Cryptology - Eurocrypt 2011 (to appear)*, 2011. This corresponds to Chapter 6.

1.3. Outline

This thesis is organized as follows. In Chapter 2, we present the mathematical background necessary for the understanding of Elliptic Curve and Pairing-based Cryptography, including curve definitions and operation costs that will be accessed throughout the thesis.

In Chapter 3, we introduce the novel precomputation schemes, namely LM and LG schemes, and present their operation costs when applied to different curve forms in various settings.

In Chapter 4, we discuss our contributions for accelerating the evaluation stage using multibase representations. We present the theoretical analysis of the (width-w) multibase NAF method, optimize the windowed variant by applying fractional windows and introduce the new methodology to derive refined algorithms able to find improved multibase chains.

In Chapter 5, we discuss the efficient implementation of elliptic curve scalar multiplication on x86-64 processors, present highly optimized field and point arithmetic and discuss our implementation results on a variety of 64-bit platforms.

In Chapter 6, we discuss the generalization of the lazy reduction technique for the efficient computation of pairings, present the highly optimized formulas and illustrate the performance improvement with a high-speed implementation of an optimal ate pairing on a BN curve.

Finally, in Chapter 7 we summarize the contributions of this thesis and discuss future work.

At the end, we present several appendices. In Appendices A1 and A2 we present the detailed pseudocode of the LM precomputation scheme and derive the costs of the method for the different variants. In Appendix A3, we present the explicit formulas for conjugate addition using Jacobian, extended Jacobi quartic and inverted Edwards coordinates. In Appendix A4 we detail the calculation of points using the LG precomputation scheme for different number of precomputations. In Appendices A5 and A6, we prove the theoretical costs of the LG method for single and multiple scalar multiplication cases. Appendix A7 presents extended cost comparisons between precomputation methods using 256- and 512-bit scalars. In Appendix B1 and B2, we detail the optimized point formulas used in our traditional and GLS-based implementations of scalar multiplication, respectively. Appendix C1 discusses the application of the generalized lazy reduction technique to compressed squarings.

Chapter 2

Background

In this chapter, we introduce the mathematical tools that are considered fundamental for the understanding of Elliptic Curve and Pairing-based Cryptography. For more extensive treatments, the reader is referred to [HMV04, ACD+05]. First, we begin with an exposition of basic abstract algebra and elliptic curves, and then discuss the security foundations of ECC, some of the most popular EC-based cryptographic schemes and the arithmetic layers that constitute the computation of elliptic curve scalar multiplication. Following, we summarize some advanced research topics related to special curves and the Galbraith-Lin-Scott (GLS) method, which are extensively used in Chapters 3-5. We end this chapter with a brief introduction to Pairing-based Cryptography, including a description of the optimal ate pairing used in Chapter 6.

2.1. Preliminaries

In this section, we introduce some fundamental concepts about finite groups, finite fields, cyclic subgroups and the generalized discrete logarithm problem.

Finite Groups

A set G is called a finite group with order q, and denoted by $(G, *)$, if it has a finite number q of elements, has a binary operation $* : G \times G \rightarrow G$ and satisfies the following properties [HMV04]:

- Associativity: $(a * b) * c = a * (b * c)$, for all elements $a, b, c \in \mathrm{G}$.
- Existence of an identity: there exists an element $e \in \mathrm{G}$ such that $a * e = e * a = a$ for all $a \in \mathrm{G}$. Element *e* is called the identity of the group.
- Existence of inverses: for each element $a \in \mathrm{G}$, there exists an element $b \in \mathrm{G}$ such that $a * b = b * a = e$. Element *b* is called the inverse of *a*.

In addition, the group is called *abelian* if it satisfies the commutativity law, that is, $a * b = b * a$, for all elements $a, b \in \mathrm{G}$.

If the binary (group) operation is called addition (+), then the group is additive. In this case, the identity element is usually denoted by 0 (zero) and the additive inverse of an element *a* is denoted by $-a$. If, otherwise, the binary (group) operation is called multiplication $(\bullet)$, then the finite group is multiplicative. In this case, the identity element is usually denoted by 1 and the multiplicative inverse of an element *a* is denoted by a^{-1}.

Finite Fields

A field is a set $\mathbb{F}$ together with two operations, addition (+) and multiplication $(\bullet)$, s.t. $(\mathbb{F}, +)$ and $(\mathbb{F}^*, \bullet)$ are abelian groups and the distributive law $(a + b) \cdot c = a \cdot c + b \cdot c$ holds for all elements $a, b, c \in \mathbb{F}$. There exists a finite field if and only if its order *q* is a prime power with the form $q = p^m$, where *p* is a prime and $m \geq 1$. We denote this field by $\mathbb{F}_q$ and distinguish the following cases:

- If $m = 1$, it is called *prime field* and is denoted by $\mathbb{F}_p$. In this case, $\mathbb{F}_p = \{0, 1, 2, \ldots, p-1\}$, which are all the integers modulo *p*. The group operations are then addition and multiplication *modulo p*.
- If $m \geq 2$, it is called *extension field* and is denoted by $\mathbb{F}_{p^m}$. Using polynomial basis representation, one can define $\mathbb{F}_{p^m}$ as the set of all polynomials in the indeterminate *x* with coefficients in $\mathbb{F}_p$ and degree at most $(m-1)$:

  $$\mathbb{F}_{p^m} = \mathbb{F}_p[x] / f(x) = \{a_{m-1}x^{m-1} + a_{m-2}x^{m-2} + \ldots + a_1 x + a_0 : a_i \in \mathbb{F}_p\}.$$

  The group operations are polynomial addition and multiplication with coefficients reduced modulo *p*. Multiplication is performed modulo an irreducible polynomial $f(x)$. Special cases of extension fields are, for example, $\mathbb{F}_{2^m}$, which are known as *binary extension fields* (or, simply, binary fields), and $\mathbb{F}_{p^2}$, which are known as *quadratic extension fields.*

Two notable cases are extensively used today to build elliptic curve cryptosystems: prime fields $\mathbb{F}_p$ and binary fields $\mathbb{F}_{2^m}$. In this thesis, we focus on the former case. Also, other extension fields $\mathbb{F}_{p^m}$ of large prime characteristic are employed in many applications including

pairing-based cryptography (see Section 2.3) and new ECC systems based on the GLS method (see Section 2.2.6).

Cyclic Subgroups

Let G be a finite group of order n with multiplication $(\bullet)$ as binary operation, and let g be an element of G such that $\langle g \rangle = \{g^i : 0 \le i \le r-1\}$ is the subgroup of G generated by g, where r is the order of the element g, that is, r is the smallest positive integer for which $g^r = 1$. It is known that r always exists and is in fact a divisor of n. Then, G is a cyclic group with generator g if $G = \langle g \rangle$ (i.e., $r = n$ holds). The set $\langle g \rangle$ is also a group itself under the same binary operation and is called the *cyclic subgroup* of G generated by g. More precisely, G contains exactly one cyclic subgroup of order d for each divisor d of n.

In the next section, we explore the way in which all the points belonging to an elliptic curve over a prime field $\mathbb{F}_p$ form an abelian group under addition, and how the cyclic subgroups of this group can be used to implement EC-based cryptosystems.

Generalized Discrete Logarithm Problem (DLP)

Given a multiplicative cyclic group $(G, \bullet)$ of order n with generator g and an element $y \in \langle g \rangle$, the DLP is defined as the problem of determining the unique integer $x \in [0, n-1]$ such that $y = g^x$. In this case, a system based on this problem is considered suitable for cryptography if the discrete logarithm problem is intractable and there are fast algorithms to compute the group operation in G.

Two groups are extensively used in discrete logarithm (DL) systems: the cyclic subgroups of the multiplicative group of a finite field and cyclic subgroups of elliptic curve groups. The former case has been studied since the late 70's. Hence, cryptosystems based on this setting will be regarded as *traditional DL-based systems*.

2.2. Introduction to Elliptic Curves

A *non-singular* elliptic curve E over a finite field K, which is denoted by E/K, is defined by the general Weierstrass equation:

$$E_{W,a_1,a_2,a_3,a_4,a_6} : \quad y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6, \tag{2.1}$$

where: $a_1, a_2, a_3, a_4, a_6 \in K$, the discriminant $\Delta = -d_2^2 d_8 - 8d_4^3 - 27d_6^2 + 9d_2 d_4 d_6 \neq 0$, $d_2 = a_1^2 + 4a_2$, $d_4 = 2a_4 + a_1 a_3$, $d_6 = a_3^2 + 4a_6$ and $d_8 = a_1^2 a_6 + 4a_2 a_6 - a_1 a_3 a_4 + a_2 a_3^2 - a_4^2$. The condition $\Delta \neq 0$ guarantees that there does not exist more than *one* tangent line for a given point on the curve, i.e., the curve is "smooth".

If we define elliptic curve points as the pairs (x, y) solving the curve equation (2.1) and $\mathbb{L}$ is any extension field of *K*, the set of $\mathbb{L}$-rational points on $E_{W,a_1,a_2,a_3,a_4,a_6}$ is:

$$E_W(\mathbb{L}) = \{(x,y) \in \mathbb{L} \times \mathbb{L} : y^2 + a_1xy + a_3y - x^3 - a_2x^2 - a_4x - a_6 = 0\} \cup \{\mathcal{O}\}, \qquad (2.2)$$

where $\mathcal{O}$ represents the point at infinity and is an $\mathbb{L}$-rational point for all extension fields $\mathbb{L}$ of *K*.

Definition 2.1. Two elliptic curves $E_1 = E_{W,a_1,a_2,a_3,a_4,a_6}$ and $E_2 = E_{W,b_1,b_2,b_3,b_4,b_6}$ defined over *K* in Weierstrass form are said to be *isomorphic over K* if there exist $r,s,t \in K$ and $u \in K \setminus \{0\}$ such that the mapping (also called an *admissible change of variables*):

$$(x, y) \mapsto \left(u^2x + r, u^3y + u^2sx + t\right) \qquad (2.3)$$

transforms E_1 into E_2.

Definition 2.2. If $r,s,t \in \bar{K}$ (closure of *K*) and $u \in \bar{K} \setminus \{0\}$ in the setting of Definition 2.1, then curves E_1 into E_2 are *isomorphic over* $\bar{K}$ or *twists* of each other. Moreover, $j(E_1) = j(E_2)$ if and only if E_1 into E_2 are twists, where $j()$ denotes the *j-invariant* of a given curve equation.

Following Definition 2.2, the *j*-invariant can be used to determine if two curves are twists.

The Weierstrass equation has had a privileged role in most standards and cryptographic applications because of the fact that every elliptic curve can be expressed in this form. Moreover, it enables efficient computation when simplified to its isomorphic forms over *K* known as *short Weierstrass curves*, which are obtained through an admissible change of variables.

2.2.1. Short Weierstrass Form

Since in the present work we mainly focus our attention on prime fields $\mathbb{F}_p$ with $p > 3$, we limit following definitions to $\mathbb{F}_p$ only. However, the reader should be aware that the same descriptions extend to any prime field *K* with prime characteristic > 3.

For the case of $\mathbb{F}_p$ with $p > 3$, the general Weierstrass equation (2.1) simplifies to the following form, known as *short Weierstrass form*:

$$E_{W,a,b}: \quad y^2 = x^3 + ax + b, \qquad (2.4)$$

where $a,b \in \mathbb{F}_p$, $\Delta = -16(4a^3 + 27b^2) \neq 0$ and $j(E_{W,a,b}) = 1728a^3/4\Delta$. In the remainder of this work, we refer to eq. (2.4) as simply E_W. Since this curve form has been recommended (and even enforced in some cases) by numerous international standardization bodies, we will also

refer to it as *standard curve*.

The set of elliptic curve points (x, y) solving the curve equation (2.4) plus the point at infinity, which is given by:

$$E_W(\mathbb{F}_p) = \{(x, y) \in \mathbb{F}_p \times \mathbb{F}_p : y^2 - x^3 - ax - b = 0\} \cup \{\mathcal{O}\}, \tag{2.5}$$

form an additive abelian group $(E_W(\mathbb{F}_p), +)$ when the so called *chord-and-tangent rule* is used to define the group operation. In this case, the point at infinity $\mathcal{O}$ acts as the identity element of the group law (see Section 2.2.4.2 for more details).

Cyclic subgroups of the group $(E_W(\mathbb{F}_p), +)$ can be used to build elliptic curve cryptosystems. The hardness of these constructs is based on the so-called Elliptic Curve Discrete Logarithm Problem (ECDLP), described next.

2.2.2. Scalar Multiplication and the Elliptic Curve Discrete Logarithm Problem (ECDLP)

Let $E/\mathbb{F}_p$ be an elliptic curve defined over $\mathbb{F}_p$. If $P \in E(\mathbb{F}_p)$ is a point of order *r*, the cyclic subgroup of $E(\mathbb{F}_p)$ generated by *P* is $\{\mathcal{O}, P, 2P, \ldots (r-1)P\}$. Then, if we define the scalar *k* as an integer in the range $[1, r-1]$, we can represent the main operation in ECC, namely, scalar multiplication (a.k.a. point multiplication), as the following computation:

$$Q = kP, \tag{2.6}$$

where the result *Q* is also a point in the subgroup of $(E(\mathbb{F}_p), +)$ generated by *P*.

Although the scalar multiplication with form (2.6) is the most common operation in elliptic curve cryptosystems, some settings such as digital signatures require a computation with the form $kP + lQ$, where $P, Q \in E(\mathbb{F}_p)$ are points of order *r* and *k*, *l* are integers in the range $[1, r-1]$. This operation is also known as *multiple scalar multiplication*. To make a clear distinction between both primitives, we will refer to operation (2.6) as *single scalar multiplication* whenever necessary.

The hardness of systems based on elliptic curve scalar multiplication is based on the Elliptic Curve Discrete Logarithm Problem (ECDLP), which is an adaptation of the traditional DLP to elliptic curve groups.

Definition 2.3. Given the cyclic group $(E(\mathbb{F}_p), +)$ with generator *P* and a point $Q \in \langle P \rangle$, the ECDLP is defined as the problem of determining the unique integer $k \in [0, r-1]$ such that $Q = kP$, where *r* is the order of points *P* and *Q*.

The ECDLP is assumed to be harder than other recognized problems such as integer factorization and the discrete logarithm problem in the multiplicative group of a finite field, which are the foundations of RSA [RSA78] and the ElGamal [ElG84] cryptosystems, respectively.

To assess more precisely the impact of the attacks available for each problem, we first introduce the following definition about algorithmic running time.

Definition 2.4. If we define the running time of a given algorithm with input n by $L_n[a,c] = O\left(\exp((c+\varepsilon)(\ln n)^a (\ln\ln n)^{1-a})\right)$, where $c > 0$ and $0 \le a \le 1$ are constants and $\lim_{n\to\infty} \varepsilon = 0$, then it is said to be polynomial in $\ln n$ (i.e., $O((\ln n)^{c+\varepsilon})$) if $a = 0$, exponential in n (i.e., $O(n^{c+\varepsilon})$) if $a = 1$, and subexponential if $0 < a < 1$.

Then, the parameter a can be seen as a measure of the efficacy of an attack to solve a particular problem, where higher values indicate inefficiency (as it is approximating to exponential running time) and lower values indicate efficiency (as it is approximating to polynomial running time). As consequence, one would prefer systems for which only exponential attacks are known.

In particular, the need for increasingly larger keys in RSA and traditional DL-based systems is due to the existence of a sub-exponential attack, known as the Number Field Sieve (NFS) [LLM+93, Gor93], which solves the integer factorization and discrete logarithm problems. This attack falls in the category of the well-known *index calculus* attacks, and has an expected running time of $L_n[\frac{1}{3}, 1.923]$. In contrast, the fastest known method to solve ECDLP is Pollard's rho [Pol78], which falls in the category of *square root* attacks and has the exponential running time $O(\sqrt{r})$, where r is the order of the cyclic group with generator P in the setting of Definition 2.3.

Note that there are "weaker" curves such as supersingular curves for which it is feasible to transport the ECDLP to the DLP in the group $\mathbb{F}_{q^k}^*$ using the Weil pairing and then to apply index calculus attacks [MOV93]. However, for the wide range of remaining elliptic curves with large prime order subgroup there are still no better attacks than Pollard's rho.

In conclusion, it is expected that the key sizes required for ECC using a suitably chosen curve and underlying field for a given security level are significantly smaller than those required for traditional cryptosystems based on the integer factorization and DL problems.

Table 2.1 shows the key sizes for EC-based and RSA cryptosystems for equivalent security levels, as recommended by [NIST07]. Security levels are shown at the bottom of the table and refer to the bitlength n of keys in a well-designed symmetric cryptosystem such that a brute force attack would require performing 2^n steps in order to break the system. For instance, an attacker would need to go through all 2^{256} possible keys to break AES-256, where $n = 256$. Estimates for ECC and RSA systems are based on the key size necessary to successfully run the fastest

algorithm that solves each problem (i.e., Pollard's rho and NFS, respectively) in a number of steps that matches the corresponding security level.

Table 2.1. Key sizes for ECC and RSA for equivalent security levels [NIST07].

Cryptosystem	Key size (bits)				
ECC	160	224	256	384	512
RSA	1024	2048	3072	7680	15360
Security level	80	112	128 (AES-128)	192 (AES-192)	256 (AES-256)

As we can observe from Table 2.1, ECC requires much smaller keys. This directly translates to important savings in bandwidth and memory requirements to transmit/store key material. Moreover, with the rapid advances in software/hardware implementation during the last years, that advantage has also been extended to faster execution times.

These advantages directly reflect on cryptographic systems based on elliptic curves that have single and multiple scalar multiplications as their main primitives. Next, we review some of the best known elliptic curve cryptosystems.

2.2.3. Elliptic Curve Cryptographic Schemes

Elliptic Curve Key Generation

First, a public-key system requires a key pair consisting of the private and public keys. This is given in Algorithm 2.1 for the case of ECC.

Algorithm 2.1. Elliptic curve key generation

Input: domain parameters (E, p, P, r)

Output: private key k and public key Q

1: Select a random integer $k \in [1, r-1]$

2: Compute $Q = kP$

3: Return Q

Elliptic Curve Diffie-Hellman Key Exchange (ECDH)

Based on the original key exchange proposed by Diffie and Hellman in [DH76], this scheme makes use of elliptic curve groups to allow that two parties establish a shared secret key over a

public medium. The protocol is illustrated in Algorithm 2.2 for the case of ECC.

Algorithm 2.2. Elliptic curve Diffie-Hellman key exchange (ECDH)

Input: domain parameters (E, p, P, r)

Output: shared secret key $Q = abP$

Alice side:		Bob side:	
1:	Select a random integer $a \in [1, r-1]$	1:	Select a random integer $b \in [1, r-1]$
2:	Compute $Q_a = aP$ and send it to Bob	2:	Compute $Q_b = bP$ and send it to Alice
3:	Upon reception of Q_b, compute $Q = aQ_b$	3:	Upon reception of Q_a, compute $Q = bQ_a$

ElGamal Elliptic Curve Cryptosystem

This cryptosystem is an adaptation to ECC of the encryption/decryption system proposed by ElGamal in [ElG84]. Encryption and decryption schemes are illustrated in Algorithms 2.3 and 2.4, respectively.

Algorithm 2.3. ElGamal elliptic curve encryption

Input: domain parameters (E, p, P, r), public key Q and plaintext m

Output: ciphertext (C_0, C_1)

1: Represent m as a point $M \in E(\mathbb{F}_p)$
2: Select a random integer $d \in [1, r-1]$
3: Compute $C_0 = dP$
4: Compute $C_1 = M + dQ$
5: Return (C_0, C_1)

Algorithm 2.4. ElGamal elliptic curve decryption

Input: domain parameters (E, p, P, r), private key k and ciphertext (C_0, C_1)

Output: plaintext m

1: Compute $M = C_1 - kC_0$
2: Extract m from M
3: Return m

Elliptic Curve Digital Signature Algorithm (ECDSA)

This is the elliptic curve analogue of the Digital Signature Algorithm (DSA) and is the most

popular EC-based signature scheme. It has been standardized in ANSI X9.62, FIPS 186-2, IEEE 1363-2000 and ISO/IEC 15946-2. Signature generation and verification are illustrated in Algorithms 2.5 and 2.6 . H denotes a hash function that is assumed to be preimage and collision resistant.

Algorithm 2.5. ECDSA signature generation

Input: domain parameters (E, p, P, r), private key k and message m
Output: signature (s_0, s_1)

1: Select a random integer $d \in [1, r-1]$
2: Compute $dP = (x_1, y_1)$ and set $z = x_1$
3: Compute $s_0 \equiv z \pmod{r}$. If $s_0 = 0$, go to step 1
4: Compute $e = H(m)$
5: Compute $s_1 = d^{-1}(e + kz) \bmod r$. If $s_0 = 0$, go to step 1
6: Return (s_0, s_1)

Algorithm 2.6. ECDSA signature verification

Input: domain parameters (E, p, P, r), public key Q, message m and signature (s_0, s_1)
Output: reject or accept the signature

1: If $(s_0, s_1) \notin [1, r-1]$, return (reject the signature)
2: Compute $e = H(m)$
3: Compute $t \equiv s_1^{-1} \bmod r$.
4: Compute $u = et \bmod n$ and $v = s_0 t \bmod n$
5: Compute $T = uP + vQ = (x_1, x_2)$ and set $z = x_1$. If $T = \mathcal{O}$, return (reject the signature)
6: If $s_0 \equiv z \pmod{r}$, return (accept the signature)
7: Else return (reject the signature)

The security of the ECDH key exchange, ElGamal elliptic curve cryptosystem and ECDSA is based on the intractability of the ECDLP in $\langle P \rangle$. In addition, the ECDSA requires that the hash function H be preimage and collision resistant. As can be seen, scalar multiplication (or multiple scalar multiplication) constitutes the central (and most time-consuming) operation of the schemes above. Hence, speeding up this operation has a direct impact in the computing performance of any cryptographic protocol based on elliptic curves.

In the following section, we briefly describe the arithmetic layers that constitute the computation of elliptic curve scalar multiplication. The interested reader is referred to [HMV04, ACD+05] for a more detailed look at the topic.

2.2.4. ECC Scalar Multiplication Arithmetic

The computation of elliptic curve scalar multiplication consists of *three* arithmetic levels or layers: field, point and scalar arithmetic. As previously seen, a cryptographic protocol or scheme works on top of scalar multiplication. However, since this thesis focuses on the efficient computation of this operation, our discussion will center on the aforementioned arithmetic levels.

2.2.4.1. Level 1: Finite Field Arithmetic

The lowest level of scalar multiplication over prime fields consists of finite field operations, which are basically traditional arithmetic operations reduced *modulo* the prime p:

- Addition: given $a,b \in \mathbb{F}_p$, compute $(a+b) \bmod p = r$, where $r = a + b - p$ if $a + b \geq p$ or $r = a + b$ if $a + b < p$.
- Subtraction: given $a,b \in \mathbb{F}_p$, compute $(a-b) \bmod p = r$, where $r = a - b + p$ if $a - b < 0$ or $r = a - b$ if $a - b \geq 0$.
- Multiplication: given $a,b \in \mathbb{F}_p$, compute $(a \cdot b) \bmod p = r$, where r is the remainder of dividing $(a \cdot b)$ by p s.t. $0 \leq r \leq p-1$.
- Squaring: given $a \in \mathbb{F}_p$, compute $a^2 \bmod p = r$, where r is the remainder of dividing a^2 by p s.t. $0 \leq r \leq p-1$.
- Inversion: given a nonzero element $a \in \mathbb{F}_p$, compute $a^{-1} \bmod p = r$, where r is the unique integer in $\mathbb{F}_p$ for which $(a \cdot r) \bmod p = 1$.

Since modular reduction represents an important portion of the cost of computing modular arithmetic, it is relevant to optimize this operation. In the setting of elliptic curve point multiplication, the selection of a prime of special form (e.g., a pseudo-Mersenne prime p s.t. $p \approx 2^m$) enables very efficient modular reduction; see Chapter 5 for an implementation of the field arithmetic using a pseudo-Mersenne prime. If a general form for the prime p is mandatory for security concerns (e.g., in pairing-based cryptosystems), then the use of Montgomery arithmetic [Mon85] is a popular choice given its relatively efficient reduction step. In this case, elements x are represented with the form $a = x \cdot 2^N \bmod p$, where $N = t \cdot w$, $2^N > p$, w is the computer wordlength and t is the number of words. Montgomery reduction produces $a \cdot 2^{-N} \bmod p$ for an input $a < 2^N \cdot p$. Then, Montgomery multiplication of elements $a = x \cdot 2^N \bmod p$ and $b = y \cdot 2^N \bmod p$ can be performed as $c = ab \bmod p = (x \cdot y \cdot 2^{2N}) \cdot 2^{-N} \pmod p = xy \cdot 2^N \bmod p$, which is in Montgomery representation; see Chapter 6 for an implementation of the field arithmetic using Montgomery arithmetic with a prime of "general" form.

The reader is referred to [HMV04, Chapter 2] and [ACD+05, Chapter 10] for more detailed discussions about efficient algorithms to perform integer arithmetic and field operations.

In the remainder of this work, we use the following notation in *italics* to specify the computing time (or computing cost) required to perform field operations in $\mathbb{F}_p$: *A* (field addition or subtraction), *S* (field squaring), *M* (field multiplication) and *I* (field inversion). In some cases, multiplication by a curve parameter is required. The cost of this operation is denoted by *D*.

In theoretical estimates throughout this work, we make the following assumptions: $1S = 0.8M$, which is commonly used in the literature; the costs of computing field addition/ subtraction and division/multiplication by a small constant are roughly equivalent to one another and/or negligible in comparison with the cost of field multiplication and squaring; and curve parameters are suitably chosen such that the cost of multiplying by these constants is negligible. Whenever required for simplification purposes, the assumptions above are applied in our theoretical cost analysis. However, the reader should be aware that these assumptions may vary from one implementation to another.

2.2.4.2. Level 2: Point Arithmetic

This level consists of the binary (group) operation accompanying the defined additive abelian group $(E(\mathbb{F}_p),+)$. The different variants of this group operation are better known as *point operations*.

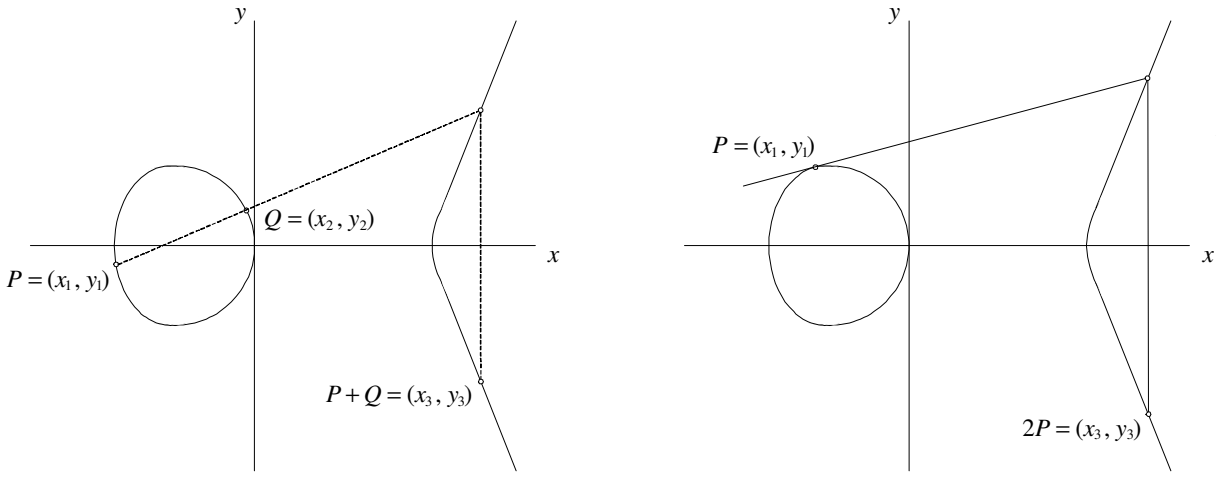
The elementary representation of points is based on the natural representation using (x, y) coordinates, which is called in the context of ECC *affine coordinates* (denoted by $\mathcal{A}$ for the remainder of this work). As previously stated, the group addition is geometrically defined by the *chord-and-tangent rule*: (i) the result of adding two points is the projection over the *x* axis of the point that intersects the line that crosses the two original points being added. This operation is referred to as *point addition* and can be visualized in Figure 2.1(a) over the real numbers; (ii) the result of adding a point to itself can be geometrically defined as the projection over the *x* axis of the point that intersects the tangent of the original point. This operation is referred to as *point doubling* and can be visualized in Figure 2.1(b) over the real numbers.

Following the geometrical definition, it is relatively easy to derive the following formula to add two points. Let E_W be an elliptic curve over $\mathbb{F}_p$ in short Weierstrass form (2.4), where $p > 3$. Given two points $P = (x_1, y_1)$ and $Q = (x_2, y_2) \in E_W(\mathbb{F}_p)$, where $P \neq \pm Q$, the addition $P + Q = (x_3, y_3)$ is obtained as follows:

$$x_3 = \lambda^2 - x_1 - x_2, \quad y_3 = \lambda(x_1 - x_3) - y_1, \tag{2.7}$$

where: $\lambda = \dfrac{y_2 - y_1}{x_2 - x_1}$. This addition formula has a cost of $1I + 2M + 1S$.

Similarly, formula for point doubling in affine coordinates can be easily derived from the previously described geometric description. Let E_W be an elliptic curve over $\mathbb{F}_p$ in short Weierstrass form (2.4), where $p > 3$. Given a point $P = (x_1, y_1) \in E_W(\mathbb{F}_p)$, $2P = (x_3, y_3)$ can be



(a) Addition P+Q (b) Doubling 2P

Figure 2.1. Group law over $\mathbb{R}$.

obtained as follows:

$$x_3 = \lambda^2 - 2x_1, \quad y_3 = \lambda(x_1 - x_3) - y_1, \tag{2.8}$$

where: $\lambda = \dfrac{3x_1^2 + a}{2y_1}$. The cost of the previous formula is $1I + 2M + 2S$.

There are a few exceptions to the previous formulas that can be solved by applying the identity element, namely, the point at infinity $\mathcal{O}$. Recall that the point at infinity can be geometrically defined as the point "lying far out on the y-axis such that any line $x = c$, for some constant c, parallel to the y-axis passes through it" [ACD+05]. Thus, if $P = (x_1, y_1)$ and $Q = (x_1, -y_1)$, then the addition is given by: $P + Q = (x_1, y_1) + (x_1, -y_1) = \mathcal{O}$. $Q = (x_1, -y_1)$ is called the negative of P and is denoted by $-P$. Similarly, $P + \mathcal{O} = \mathcal{O} + P = P$, and $\mathcal{O} = -\mathcal{O}$.

Inversion-Free (Projective) Coordinates

As we have seen in the previous section, point formulas based on affine coordinates require the computation of field inversions. Particularly over prime fields, inversions are highly expensive in comparison with other field operations, and should be avoided as much as possible. Although their relative cost depends on the characteristics of a particular implementation, it has been observed that, especially in the case of efficient forms for the prime p as recommended by [NIST00], $1I > 30M$. For instance, benchmarks presented by [LH00] and [BHL+01] show I/M ratios between 30-40 and 50-100, respectively.

To solve this problem, one can use instead projective coordinates with the form $(X : Y : Z)$,

in which case the third coordinate Z permits to replace inversions for a few other field operations. More precisely, given a prime field $\mathbb{F}_p$ and $c,d \in \mathbb{Z}^+$, there is an equivalence relation $\sim$ among nonzero triplets over $\mathbb{F}_p$, such that [HMV04]:

$$(X_1,Y_1,Z_1) \sim (X_2,Y_2,Z_2) \;\Leftrightarrow\; X_1 = \lambda^c X_2,\; Y_1 = \lambda^d Y_2 \text{ and } Z_1 = \lambda Z_2, \text{ for some } \lambda \in \mathbb{F}_p^*, \qquad (2.9)$$

The equivalence class of a projective point is $(X:Y:Z) = \{(\lambda^c X, \lambda^d Y, \lambda Z) : \lambda \in \mathbb{F}_p^*\}$, where any element (X,Y,Z) can be used as a *representative* of such a point. In particular, $(X/Z^c, Y/Z^d, 1)$ is the only representative in the set for which $Z = 1$. That means that there is a *one-to-one* mapping between affine points and projective points.

If, for instance, one fixes $c = 2$ and $d = 3$ the new representation is known as *Jacobian coordinates* (denoted by $\mathcal{J}$ in the remainder), which is a special case of projective coordinates that has yielded very efficient point formulae [HMV04, Elm06]. Then, in this case the equivalence class of a *(Jacobian) projective point* is given by:

$$(X:Y:Z) = \{(\lambda^2 X, \lambda^3 Y, \lambda Z) : \lambda \in \mathbb{F}_p^*\}. \qquad (2.10)$$

Note that, in the Jacobian representation, each projective point $(X:Y:Z)$ corresponds to the affine point $(X/Z^2, Y/Z^3)$. In this case, the curve equation (2.4) acquires the projective form $Y^2 = X^3 + aXZ^4 + bZ^6$, the negative of a point $P = (X:Y:Z)$ is given by $-P = (X:-Y:Z)$ and the point at infinity corresponds to $\mathcal{O} = (1:1:0)$.

In Table 2.2, we summarize costs of the most efficient point formulas in $\mathcal{J}$ coordinates, including recently proposed *composite operations* such as tripling (3*P*) and quintupling (5*P*) of a point, which are built on top of traditional doubling and addition operations and are relevant for the efficient implementation of multibase scalar multiplication methods (see Chapter 4). Also, we include the highly efficient doubling-addition operation proposed by the author in [Lon07] which computes the recurrent value $2P + Q$ and is more efficient than performing a doubling followed by an addition when using Jacobian coordinates (see also [LM08b]). Besides "traditional" costs in each case, we also show costs of formulas after applying the technique of replacing multiplications by squarings (labeled as "Using *S*-*M* tradings") [LM08] using the algebraic substitutions $a \cdot b = \left[(a+b)^2 - a^2 - b^2\right]/2$ or $2a \cdot b = \left[(a+b)^2 - a^2 - b^2\right]$. In general, this technique is more efficient always that $M - S > 4A$ or $M - S > 2A$ (respect.). The reader is referred to our online database [Lon08] for complete details about state-of-the-art formulas using Jacobian coordinates.

Note that formulas considered in Table 2.2 fix $a = -3$ in the curve equation (2.4) for efficiency purposes. This assumption, which has been shown not to impose significant restrictions to the cryptosystem [BJ03], has been recommended and incorporated in public-key standards [NIST00, IEEE00].

Table 2.2. Costs (in terms of multiplications and squarings) of point operations using Jacobian ($\mathcal{J}$) and mixed Jacobian-affine coordinates.

Point operation	Cost	
	"Traditional"	Using *S*-*M* tradings
Doubling (DBL), $2\mathcal{J} \to \mathcal{J}$	$4M + 4S$	$3M + 5S$
Mixed doubling (mDBL), $2\mathcal{A} \to \mathcal{J}$	$2M + 4S$	$1M + 5S$
Tripling (TPL), $3\mathcal{J} \to \mathcal{J}$	$9M + 5S$	$7M + 7S$
Mixed tripling (mTPL), $3\mathcal{A} \to \mathcal{J}$	$7M + 5S$	$5M + 7S$
Quintupling (QPL), $5\mathcal{J} \to \mathcal{J}$	$13M + 9S$	$10M + 12S$
Mixed quintupling (mQPL), $5\mathcal{A} \to \mathcal{J}$	$12M + 8S$	$8M + 12S$
Mixed addition (mADD), $\mathcal{J} + \mathcal{A} \to \mathcal{J}$	$8M + 3S$	$7M + 4S$
Mixed[2] addition (mmADD), $\mathcal{A} + \mathcal{A} \to \mathcal{J}$	$4M + 2S$	$4M + 2S$
Addition (ADD), $\mathcal{J} + \mathcal{J} \to \mathcal{J}$	$12M + 4S$	$11M + 5S$
Addition with *two* stored values ($\text{ADD}_{[1,1]}$), $\mathcal{J} + \mathcal{J} \to \mathcal{J}$	$11M + 3S$	$10M + 4S$
Addition with *four* stored values ($\text{ADD}_{[2,2]}$), $\mathcal{J} + \mathcal{J} \to \mathcal{J}$	$10M + 2S$	$9M + 3S$
Mixed doubling-addition (mDBLADD), $2\mathcal{J} + \mathcal{A} \to \mathcal{J}$	$13M + 5S$	$11M + 7S$
Doubling-addition (DBLADD), $2\mathcal{J} + \mathcal{J} \to \mathcal{J}$	$17M + 6S$	$14M + 9S$
Doubling-addition ($\text{DBLADD}_{[1,1]}$), $2\mathcal{J} + \mathcal{J} \to \mathcal{J}$	$16M + 5S$	$13M + 8S$

For the remainder, doubling ($2P$), tripling ($3P$), quintupling ($5P$), addition (P+Q) and doubling-addition ($2P$+Q) are denoted by DBL, TPL, QPL, ADD and DBLADD, respectively. If at least one of the inputs is in affine and the output is in $\mathcal{J}$ coordinates, the operations use *mixed* coordinates (see Cohen et al. [CMO98]) and are denoted by mDBL, mTPL, mQPL, mADD and mDBLADD, corresponding to each of the previous point operations. For addition, the case in which both inputs are in affine is denoted by mmADD. Costs are expressed in terms of field multiplications (M) and squarings (S) only. The reader is referred to [Lon08] for the full operation count.

In some cases, it is possible to reduce the cost of certain operations if some values are precomputed in advance. That is the case of addition and doubling-addition with stored values (identified by the subscripts $[M,S]$, where M and S denote the number of precalculated multiplications and squarings, respect.). If, for instance, values Z_i^2 and Z_i^3 are calculated for each precomputed point d_iP in windowed methods the costs of the aforementioned operations can be reduced by $1M + 1S$. Maximum savings can be achieved if *four* values, namely, Z_i^2, Z_i^3, Z_2^2 and Z_2^3, can be precalculated before performing an addition of the form $(X_1 : Y_1 : Z_1) + (X_2 : Y_2 : Z_2)$. In this case, we can save up to $2M + 2S$.

Variants of $\mathcal{J}$ coordinates have also been explored in the literature. In particular, the four-tuple $(X:Y:Z:aZ^4)$ and five-tuple $(X:Y:Z:Z^2:Z^3)$, known as modified Jacobian ($\mathcal{J}^m$) [CMO98] and Chudnovsky ($\mathcal{C}$) [CC86] coordinates, respectively, permit the saving of some operations by passing recurrent values between point operations. However, most benefits achieved with these representations are virtually cancelled by assuming $a=-3$ in the EC equation and with the alternative use of operations with stored values. Other (somewhat less efficient) system, referred to as homogeneous ($\mathcal{H}$) coordinates, is defined by fixing $c=d=1$ in (2.9).

The costs presented in Table 2.2 (specifically, costs labeled as "Using *S-M* tradings") will be used later for assessing the methods proposed for precomputation and multibase scalar multiplication in Chapters 3 and 4, respectively. Also, our high-speed implementations of scalar multiplication in Chapter 5 are based on standard curves using this system. In this case, given the relatively high cost of additions and other "small" operations on x86-64 processors, we make use of "traditional" operations without exploiting *S-M* tradings.

2.2.4.3. Level 3: Scalar Arithmetic

This level of computation refers to the efficient execution of scalar multiplication (2.6) employing the point operations discussed in the previous section. Because the naïve method computing $kP = P + P + \ldots + P$ using $(k-1)$ point additions is highly expensive, it is important the use of efficient number representations for the scalar k to make this operation reasonably efficient.

In that direction there have appeared a myriad of methods for computing scalar multiplication in the last few years. These methods are generically classified according to their applicability to *two* possible scenarios: (i) the initial point P is fixed and known before execution; (ii) the initial point P varies and is not known in advance. If the initial point P is known in advance, as happens in the ElGamal elliptic curve encryption scheme or the first phase of the Diffie-Hellman key exchange (see Section 2.2.3), efficient methods can precalculate multiples of P almost for "free" to reduce costs during the evaluation stage (e.g., comb methods). On the other hand, if the point P is not known in advance, as happens during the ElGamal decryption or the second phase of the Diffie-Hellman key exchange, methods should include the precomputation cost in the overall cost and, hence, precomputed points should be used sensibly.

In this thesis we focus on methods falling in the second category (i.e., point P is not known in advance). In this case, it is standard to use the so-called *double-and-add* algorithm, which is the analogue of the square-and-multiply method used for exponentiation in multiplicative groups. This method uses the binary representation of integers, as can be seen in Algorithm 2.7(a). Moreover, since negating points is inexpensive and can be performed on-the-fly it is convenient to use signed binary representations that potentially allow the reduction of nonzero digits (which

directly translates to a reduction in the number of required additions). By adjusting the double-and-add for this case, we obtain what is known as the *double-and-(add-or-subtract)* method. See Algorithm 2.7(b). Popular signed binary representations are the standard *non-adjacent form* (NAF) and its variants, which are briefly described in the next subsection.

Algorithm 2.7. Left-to-right methods for scalar multiplication

Input: (a) $k = (k_{t-1}, k_{t-2}, \ldots, k_0)_2$ or (b) $k = (k_{t-1}, k_{t-2}, \ldots, k_0)_{\text{NAF}}$; and $P \in E(\mathbb{F}_p)$
Output: kP

(a)

1: $Q = \mathcal{O}$
2: For $i = t-1$ downto 0 do
3: $Q \leftarrow 2Q$
4: If $k_i = 1$ then $Q \leftarrow Q + P$
5: Return Q

(b)

1: $Q = \mathcal{O}$
2: For $i = t-1$ downto 0 do
3: $Q \leftarrow 2Q$
4: If $k_i = 1$ then $Q \leftarrow Q + P$
5: If $k_i = -1$ then $Q \leftarrow Q - P$
6: Return Q

Note that Algorithm 2.7 presents left-to-right versions of the methods discussed above. There are also right-to-left variants which can be advantageous when protection against side-channel analysis (SCA) attacks is required. The same observation applies to other methods such as the Montgomery Ladder [Mon87].

In the remainder of this work, for a scalar multiplication kP, we assume that $P \in E(\mathbb{F}_p)$ is of order r and $E(\mathbb{F}_p)$ is of order $\#E(\mathbb{F}_p) = h \cdot r$, where r is prime and $h << r$. Since it is known that $\#E(\mathbb{F}_p) \approx p$ following Hasse's theorem (see Theorem 3.7 in [HMV04]), we have that $r \approx p$. Then, if k is a scalar randomly chosen in the range $[1, r-1]$, the average length of k in binary representation is $n = \log_2 p$ and the corresponding operation will be referred as *n-bit scalar multiplication*. In this case, double-and-add and double-and-(add-or-subtract) algorithms will require in average $(n-1)$ main loop iterations. We refer as nonzero density or Hamming weight to the number of nonzero digits in a given scalar representation. In particular, for scalar multiplication, the nonzero density of the representation of k directly translates to the number of required point additions to compute kP.

Non-Adjacent Form (NAF) and Width-*w* Non-Adjacent Form (*w*NAF)

Among different signed radix-2 representations using digits from the set $D = \{0, \pm 1\}$, NAF is a canonical representation with the fewest number of nonzero digits for any scalar k [Rei60]. The NAF representation of k contains at most *one* nonzero digit among any two successive digits. The expected nonzero density of this representation is $\delta_{\text{NAF}} = 1/3$. Hence, the average cost of an *n*-

bit scalar multiplication using NAF is approximately $(n-1)\text{DBL}+(n/3)\text{ADD}$, where DBL and ADD represent the cost of doubling and addition, respectively.

If there is memory available, one can exploit the use of precomputations by means of a method known as *w*NAF [Sol00], which uses precomputed values to "insert" windows of width *w*. The latter permits the consecutive execution of several doublings to reduce the density of the expansion. The *w*NAF representation of *k* contains at most *one* nonzero digit among any *w* successive digits, and uses the digit set $D=\{0,\pm1,\pm3,\pm5,\ldots,\pm(2^{w-1}-1)\}$, where $w>2\in\mathbb{Z}^+$. The average density of nonzero digits for a window of width *w* is $\delta_{w\text{NAF}}=1/(w+1)$, and the number of required precomputed points is $(2^{w-2}-1)$ (hereafter we refer as precomputed points to *non-trivial points* not including $\{\mathcal{O},P\}$). Hence, the cost using this method is approximately $(n-1)\text{DBL}+(n/(w+1))\text{ADD}$ plus the cost of the precomputation stage.

Fractional Width-*w* NAF (Frac-*w*NAF)

The *w*NAF representation requires the precomputation of $(2^{w-2}-1)$ non-trivial points, i.e., 1, 3, 7, 15 points, and so on. However, a specific implementation could have memory restrictions that do not adjust to these values. Moreover, because the applicable scenario involves an initial point *P* not known in advance, the precomputed table must be built every time a scalar multiplication is performed. Hence, it is often the case that a table with a number of points different to that fixed by standard windows achieves the minimal cost.

Möller [Möl03] proposed to solve this problem by recoding the binary representation of an integer with windows of flexible size using a digit set of the form $D=\{0,\pm1,\pm3,\pm5,\ldots,\pm m\}$, where $m\geq1$ is an odd integer. In this way, one can flexibly choose any number of precomputed points. This method is denoted by Frac-*w*NAF and its expected nonzero density is given by $\delta_{\text{Frac-}w\text{NAF}}=\left(\lfloor\log_2 m\rfloor+(m+1)/(2^{\lfloor\log_2 m\rfloor})+1\right)^{-1}$ [Möl05].

Note that if $m=1$, Frac-*w*NAF is actually reduced to the NAF method with a nonzero density of about 1/3. Similarly, $\delta_{\text{Frac-}w\text{NAF}}$ attains the same values as $\delta_{w\text{NAF}}$ for the standard window values of *w*NAF. For instance, Frac-*w*NAF with $m=7$ reduces to *w*NAF with $w=4$ (in this case, $\delta=0.2$).

2.2.5. Special Curve Forms

During the last few years, there has been a growing interest in studying curve forms different to the standardized Weierstrass form (2.4). These special curves have gained increasing attention because in some cases they offer higher resilience against side-channel analysis attacks and/or enable faster implementations. In this work, we focus on *two* special curve forms that have been shown to achieve very high performance: extended Jacobi quartics and Twisted Edwards curves.

Next, we briefly describe both curve shapes in their most generalized form.

Extended Jacobi Quartic Curve

It is defined by the non-singular curve equation:

$$E_{JQ}: \; y^2 = dx^4 + 2ax^2 + 1, \tag{2.11}$$

where $a,d \in \mathbb{F}_p$ and $d(a^2 - d) \neq 0$. Results by Billet and Joye [BJ03b] show that every elliptic curve of even order can be written in extended Jacobi quartic form. The projective curve in weighted projective coordinates is given by $Y^2 = dX^4 + 2aX^2Z^2 + Z^4$, where a projective point $(X:Y:Z)$ corresponds to the affine point $(X/Z, Y/Z^2)$. In this case, the negative of a point $P = (X:Y:Z)$ is represented by $-P = (-X:Y:Z)$ and the identity element is given by $(0:1:1)$. The most efficient formulae for this case have been developed by Hisil et al. [HWC+07, HWC+08b] using an extended coordinate system of the form $(X:Y:Z:X^2:Z^2)$ that will be referred in the remainder of this work as $\mathcal{JQ}^e$.

Note that, recently, Hisil et al. [HWC+09] proposed the use of a mixed system that efficiently combines homogeneous coordinates with an extended homogeneous coordinate system with the form $(X:Y:Z:T)$, where $T = X^2/Z$. However, formulas for composite operations known to date are faster in weighted projective coordinates $\mathcal{JQ}^e$.

In Table 2.3, we summarize the costs of formulas using extended Jacobi quartic coordinates [HWC+07, HWC+08b]. Note that it is also possible to trade multiplications for squarings in some cases (labeled as "Using *S-M* tradings"). And similarly to the case of operations with stored values

Table 2.3. Costs of point operations for an extended Jacobi quartic curve with $d = 1$ using extended Jacobi quartic ($\mathcal{JQ}^e$) coordinates.

Point operation	Coord.	Cost	
		"Traditional"	Using *S-M* tradings
DBL	$2(\mathcal{JQ}^e) \to \mathcal{JQ}^e$	$3M + 4S + 1D$	$2M + 5S + 1D$
mDBL	$2(\mathcal{A}) \to \mathcal{JQ}^e$	$1M + 6S + 1D$	$7S + 1D$
TPL	$3(\mathcal{JQ}^e) \to \mathcal{JQ}^e$	$8M + 4S + 1D$	$8M + 4S + 1D$
mTPL	$3(\mathcal{A}) \to \mathcal{JQ}^e$	$5M + 6S + 2D$	$5M + 6S + 2D$
QPL	$5(\mathcal{JQ}^e) \to \mathcal{JQ}^e$	$14M + 4S + 1D$	$14M + 4S + 1D$
mQPL	$5(\mathcal{A}) \to \mathcal{JQ}^e$	$11M + 6S + 2D$	$11M + 6S + 2D$
mADD	$\mathcal{JQ}^e + \mathcal{A} \to \mathcal{JQ}^e$	$7M + 2S + 1D$	$6M + 3S + 1D$
mmADD	$\mathcal{A} + \mathcal{A} \to \mathcal{JQ}^e$	$4M + 3S + 1D$	$4M + 3S + 1D$
ADD	$\mathcal{JQ}^e + \mathcal{JQ}^e \to \mathcal{JQ}^e$	$8M + 3S + 1D$	$7M + 4S + 1D$
$ADD_{[0,1]}$	$\mathcal{JQ}^e + \mathcal{JQ}^e \to \mathcal{JQ}^e$	$8M + 2S + 1D$	$7M + 3S + 1D$

using Jacobian coord. (see Section 2.2.4.2), the original cost of addition can be reduced further. For instance, the addition with cost of $7M + 4S$ can be reduced to $7M + 3S$ by noting that $(X_i + Z_i)^2$ can be precomputed for each precomputed point (see [HWC+07] for complete details).

Given the relatively "well-balanced" performance among all point operations listed in Table 2.3, we use these costs (specifically, the costs labeled as "Using *S*-*M* tradings", assuming that $1D \approx 0M$) for evaluating the multibase methods in Chapter 4. We also use this system for illustrating the use of the LG precomputation scheme in Chapter 3.

Twisted Edwards Curve

This form is a generalization of Edwards curves [Edw07] and is defined by the non-singular curve equation:

$$E_{TE}: \; ax^2 + y^2 = 1 + dx^2y^2, \tag{2.12}$$

where $a, d \in \mathbb{F}_p$ and $ad(a-d) \neq 0$. An efficient projective system for performing arithmetic on these curves is known as *inverted Edwards coordinates* (referred to as $\mathcal{IE}$ coordinates for the reminder) [BL07b]. In this system, the equation takes the form $(X^2 + Y^2)Z^2 = X^2Y^2 + dZ^4$, assuming that $a = 1$, where $XYZ \neq 0$, each projective point $(X : Y : Z)$ corresponds to $(Z/X, Z/Y)$ in affine and the negative of a point $P = (X : Y : Z)$ is given by $-P = (-X : Y : Z)$.

Recently, there have been remarkable improvements in the case of Twisted Edwards curves using homogeneous coordinates (denoted by $\mathcal{E}$). For this case, the curve acquires the projective form $aX^2Z^2 + Y^2Z^2 = dX^2Y^2 + Z^4$ for which each triplet $(X : Y : Z)$ corresponds to the affine point $(X/Z, Y/Z)$, $Z \neq 0$. Hisil et al. [HWC+08] introduced extended homogeneous coordinates (denoted by $\mathcal{E}^e$), where each point $(X : Y : Z : T)$ corresponds to $(X/Z, Y/Z)$ in affine and $T = XY/Z$. The negative of $(X : Y : Z : T)$ is given by $(-X : Y : Z : -T)$, and $(X : Y : Z : T) = \{(\lambda X, \lambda Y, \lambda Z, \lambda T) : \lambda \in \mathbb{F}_p^*\}$. In [HWC+08], Hisil et al. also suggest the map $(x, y) \mapsto (x/\sqrt{-a}, y)$ to convert the Twisted Edwards curve to $-x^2 + y^2 = 1 + d'x^2y^2$, where $d' = -d/a$, allowing further reductions in the cost of point operations. For the point multiplication, they ultimately propose to compute a doubling followed by an addition as $2\mathcal{E} \to \mathcal{E}^e$ and $\mathcal{E}^e + \mathcal{E}^e \to \mathcal{E}$ or $\mathcal{E}^e + \mathcal{A} \to \mathcal{E}$ (which can be unified into a doubling-addition operation with the form $(2\mathcal{E})^e + \mathcal{E}^e \to \mathcal{E}$ or $(2\mathcal{E})^e + \mathcal{A} \to \mathcal{E}$) and to compute the remaining doublings as $2\mathcal{E} \to \mathcal{E}$. This combined system is called mixed homogeneous/extended homogeneous coordinates and is denoted by $\mathcal{E}/\mathcal{E}^e$.

In Table 2.4, we summarize the costs of formulas using $\mathcal{IE}$ [BL07b] and $\mathcal{E}/\mathcal{E}^e$ [HWC+08] coordinates. Again, it is possible to trade multiplications for squarings in some cases (labeled as "Using *S*-*M* tradings").

Table 2.4. Costs of point operations for a Twisted Edwards curve using inverted Edwards ($\mathcal{IE}$) and mixed homogeneous/extended homogeneous ($\mathcal{E}/\mathcal{E}^e$) coordinates.

Point operation	$\mathcal{IE}$ $(a=1)$			$\mathcal{E}/\mathcal{E}^e$ $(a=-1)$		
	Coord.	"Traditional"	Using *S-M* tradings	Coord.	"Traditional"	Using *S-M* tradings
DBL	$2(\mathcal{IE})\rightarrow\mathcal{IE}$	$4M+3S+1D$	$3M+4S+1D$	$2(\mathcal{E})\rightarrow\mathcal{E}$	$4M+3S$	$3M+4S$
mDBL	$2(\mathcal{A})\rightarrow\mathcal{IE}$	$4M+2S$	$3M+3S$	$2(\mathcal{A})\rightarrow\mathcal{E}$	-	-
TPL	$3(\mathcal{IE})\rightarrow\mathcal{IE}$	$9M+4S+1D$	$9M+4S+1D$	$3(\mathcal{E})\rightarrow\mathcal{E}$	-	-
mTPL	$3(\mathcal{A})\rightarrow\mathcal{IE}$	$7M+3S$	$7M+3S$	$3(\mathcal{A})\rightarrow\mathcal{E}$	-	-
mADD	$\mathcal{IE}+\mathcal{A}\rightarrow\mathcal{IE}$	$8M+1S+1D$	$8M+1S+1D$	$\mathcal{E}^e+\mathcal{A}\rightarrow\mathcal{E}^e$	$7M$	$7M$
mmADD	$\mathcal{A}+\mathcal{A}\rightarrow\mathcal{IE}$	$7M$	$7M$	$\mathcal{A}+\mathcal{A}\rightarrow\mathcal{E}^e$	-	-
ADD	$\mathcal{IE}+\mathcal{IE}\rightarrow\mathcal{IE}$	$9M+1S+1D$	$9M+1S+1D$	$\mathcal{E}^e+\mathcal{E}^e\rightarrow\mathcal{E}^e$	$8M$	$8M$
mDBLADD	$2(\mathcal{IE})+\mathcal{A}\rightarrow\mathcal{IE}$	-	-	$(2\mathcal{E})^e+\mathcal{A}\rightarrow\mathcal{E}$	$11M+3S$	$10M+4S$
DBLADD	$2(\mathcal{IE})+\mathcal{IE}\rightarrow\mathcal{IE}$	-	-	$(2\mathcal{E})^e+\mathcal{E}^e\rightarrow\mathcal{E}$	$12M+3S$	$11M+4S$

Given the availability of a tripling formula of relatively good performance in $\mathcal{IE}$ coordinates, we use this system (costs labeled as "Using *S-M* tradings", assuming that $1D \approx 0M$) for evaluating the multibase methods discussed in Chapter 4. We also use $\mathcal{IE}$ coordinates for illustrating the use of the LG precomputation scheme in Chapter 3. On the other hand, $\mathcal{E}/\mathcal{E}^e$ coordinates currently offer the highest performance for scalar multiplication using traditional radix-2 methods, even surpassing the performance of mixed Jacobi quartic homogeneous/ extended homogeneous coordinates (see [HWC+09]). Hence, our high-speed implementations of scalar multiplication in Chapter 5 are based on Twisted Edwards curves using this system. In this case, given the relatively high cost of additions and other "small" operations on x86-64 processors, we make use of "traditional" operations without exploiting *S-M* tradings.

2.2.6. The Galbraith-Lin-Scott (GLS) Method

Recently, Galbraith et al. [GLS09] proposed to perform ECC computations on the quadratic twist E' of an elliptic curve E over $\mathbb{F}_{p^2}$ with an efficiently computable homomorphism $\psi(x,y)\rightarrow(\alpha x,\beta y)$ such that $\psi(P)=\lambda P$ and $\lambda^2+1\equiv 0(\bmod r)$, where $P\in E'(\mathbb{F}_{p^2})[r]$. Then, following [GLV01], kP can be computed as a multiple point multiplication with form $k_0P+k_1(\lambda P)$, where k_0 and k_1 have approximately half the bitlength of k. Integers k_0 and k_1 can be calculated by solving a closest vector in a lattice or (in the case of a random scalar k) by simply choosing the integers directly [GLS09].

It has also been observed that the GLS method can be adapted to different curve forms. In Chapter 5, we evaluate various techniques and optimizations in combination with this method to realize high-speed elliptic curve implementations on software. For this purpose, we choose curves in Weierstrass and Twisted Edwards form. The details of these curve forms using the GLS method, mainly taken from the literature, are summarized next. For complete details, please refer to [GLS09, GLS08].

The Case with Weierstrass Form

Corollary 2.1. Let curve E_W over $\mathbb{F}_p$ be defined by (2.4) with $\#E_W(\mathbb{F}_p) = p+1-t$ points, where t is called the trace of $E_W/\mathbb{F}_p$, $|t| \le 2p$, and μ be a quadratic non-residue in $\mathbb{F}_{p^2}$. If $ab \neq 0$, E_W is isomorphic to the curve:

$$E'_W/\mathbb{F}_{p^2}: \; y^2 = x^3 + a'x + b', \tag{2.13}$$

with $a' = \mu^2 a$ and $b' = \mu^3 b \in \mathbb{F}_{p^2}$, and $\#E'_W(\mathbb{F}_{p^2}) = (p-1)^2 + t^2$. Curve E'_W is the *quadratic twist of* E_W *over* $\mathbb{F}_{p^2}$. The twisting isomorphism is given by $\phi: E_W \to E'_W$, $\phi(x,y) = (ux, \sqrt{u}^3 y)$, which is defined over $\mathbb{F}_{p^4}$. The group homomorphism is given by:

$$\psi(x,y) = \left(\frac{\mu}{\mu^p} \cdot \bar{x}, \sqrt{\mu^3/\mu^{3p}} \cdot \bar{y}\right), \tag{2.14}$$

where $\bar{x}$ and $\bar{y}$ denote the Galois conjugates of x and y, respectively.

The Case with Twisted Edwards Form

Corollary 2.2. Let curve E_{TE} over $\mathbb{F}_p$ be defined by (2.12) with $\#E_{TE}(\mathbb{F}_p) = p+1-t$ points $4 \mid (p+1-t)$, $|t| \le 2p$, and μ be a quadratic non-residue in $\mathbb{F}_{p^2}$. Then E_{TE} is isomorphic to the curve:

$$E'_{TE}/\mathbb{F}_{p^2}: \; a'x^2 + y^2 = 1 + d'x^2y^2, \tag{2.15}$$

with $a' = \mu a$ and $d' = \mu d \in \mathbb{F}_{p^2}$, and $\#E'_{TE}(\mathbb{F}_{p^2}) = (p-1)^2 + t^2$. Curve E'_{TE} is the *quadratic twist of* E_{TE} *over* $\mathbb{F}_{p^2}$. The twisting isomorphism is given by $\phi: E_{TE} \to E'_{TE}$, $\phi(x,y) = (x/\sqrt{u}, y)$, and the group homomorphism is:

$$\psi(x,y) = (\sqrt{\mu^p/\mu} \cdot \bar{x}, \bar{y}) \; . \tag{2.16}$$

Following [GLS09], for our implementations on Weierstrass and Twisted Edwards curves in Chapter 5 we fix $p = 2^{127} - 1 \equiv 3 \pmod 4$ and $\mu = 2 + i \in \mathbb{F}_{p^2}$ where $i = \sqrt{-1} \in \mathbb{F}_p$. The chosen prime is assumed to provide approximately 128 bits of security.

2.2.6.1. Arithmetic over Quadratic Extension Fields

Since for our case $\mathbb{F}_{p^2} = \mathbb{F}_p(i)$ with $i = \sqrt{-1} \in \mathbb{F}_p$, elements in $\mathbb{F}_{p^2}$ can be represented by $x = a + bi$, where $a, b \in \mathbb{F}_p$. The conjugate of x is given by $\bar{x} = a - bi$.

Then, field arithmetic consists of usual polynomial addition and multiplication in i with coefficients reduced modulo p. Moreover, as suggested in [GLS09], $\mathbb{F}_{p^2}$ multiplication can be sped up by using Karatsuba method [Kar95] such that $(a+bi)\cdot(c+di) = (ac-bd)+(bc+ad)i$ is computed as $(ac-bd)+((a+b)(c+d)-ac-bd)i$, which requires 3 $\mathbb{F}_p$ multiplications and 5 $\mathbb{F}_p$ additions/subtractions instead of 4 $\mathbb{F}_p$ multiplications and 2 $\mathbb{F}_p$ additions/subtractions. Similarly, a squaring with the form $(a+bi)^2$ can be computed as $(a+b)(a-b)+2abi$ with 2 $\mathbb{F}_p$ multiplications and 3 $\mathbb{F}_p$ additions/subtractions, which is more efficient than computing $(a+bi)^2 = (a^2-b^2)+2abi$ always that $2S > M + A$ or $(a+bi)^2 = (a^2-b^2)+[(a+b)^2-a^2-b^2]i$ always that $3S + A > 2M$.

2.2.6.2. Security of the GLS Method

An attack by Gaudry [Gau09] has been shown to solve the ECDLP on general abelian varieties of small dimension. Specifically, this attack can solve the ECDLP in $E(\mathbb{F}_{q^m})$ in $\tilde{O}(q^{2-2/m})$, which is faster than Pollard's rho algorithm if $m > 2$. Hence, it does not have any implications on the practical implementations in $E(\mathbb{F}_{q^2})$ discussed in this work.

Definition 2.5. Let E be an elliptic curve defined in $\mathbb{F}_q$, a point $P \in E(\mathbb{F}_q)$ of order r, a point $xP \in \langle P \rangle$ for a random integer $x \in [0, r-1]$ and a *reusable* point $aP \in \langle P \rangle$ for an integer $a \in [0, r-1]$. The *Static Diffie-Hellman Problem* (denoted by Static DHP) is defined as the problem of determining axP.

Recently, Granger [Gra10] introduced a new attack that was shown to solve the Static DHP in heuristic time $\tilde{O}(q^{1-1/(m+1)})$ for any elliptic curve in $E(\mathbb{F}_{q^m})$ if an attacker has access to a Static DHP oracle. Hence, this result is immediately more efficient than Gaudry's attack and, most importantly, faster than Pollard's rho attack if $m = 2$. Accordingly, it is suggested to avoid the use of the GLS method in settings where the Static DHP applies (e.g., when the same Diffie-Hellman secret is reused for various Diffie-Hellman key agreements). Alternatively, one may increase the key size accordingly to make this attack and Pollard's rho algorithm roughly equivalent for solving the ECDLP in $E(\mathbb{F}_{q^2})$.

We remark that it is known that the Static DHP can be solved for any arbitrary curve in $E(\mathbb{F}_q)$ with $O(q^{1/3})$ Static DHP oracle queries and $O(q^{1/3})$ group operations [BG04], which is faster than the best generic attack achieving complexity $O(q^{1/2})$, namely Pollard's rho.

2.3. Introduction to Pairings

An *admissible bilinear pairing* is an efficiently computable function $e: \mathrm{G}_1 \times \mathrm{G}_2 \to \mathrm{G}_\mathrm{T}$, where G_1 and G_2 are cyclic subgroups of elliptic curve groups, G_T is a cyclic subgroup of the multiplicative group of a finite field, G_1, G_2 and G_T have order *r*, and the following conditions hold:

- Bilinearity: for all $P, Q \in \mathrm{G}_1$ and all $R, S \in \mathrm{G}_2$, $e(P+Q,R) = e(P,R) \cdot e(Q,R)$ and $e(P,R+S) = e(P,R) \cdot e(P,S)$.
- Non-degeneracy: $e(P,R) \neq 1$ for some $P \in \mathrm{G}_1$ and $R \in \mathrm{G}_2$. Or, equivalently, $e(P,R) = 1$ for all $R \in \mathrm{G}_2$ if and only if $P = \mathcal{O}$; and $e(P,R) = 1$ for all $P \in \mathrm{G}_1$ if and only if $R = \mathcal{O}$.

Also, it immediately follows that $e(aP,bR) = e(P,R)^{ab} = e(bP,aR)$ for any two integers *a* and *b*.

Bilinear pairings provide elegant solutions to some longstanding problems in cryptography such as Identity-Based Encryption (IBE) [BF01, SOK00], three-party one-round Diffie-Hellman key exchange [Jou00], short signatures [BLS04], among others, and has been the focus of intense research since its introduction by Boneh, Franklin and others at the beginning of the new millennium. For illustration purposes we show in Algorithm 2.8 the three-party one-round key agreement by Joux using a bilinear pairing on $(\mathrm{G}_1, \mathrm{G}_\mathrm{T})$. The reader is referred to [Men09] for a discussion of other fundamental pairing-based protocols.

Algorithm 2.8. Pairing-based tree-party one-round key exchange

Input: domain parameters $(\mathrm{G}_1, \mathrm{G}_\mathrm{T}, E, p, P, r)$
Output: shared secret key $K = e(P,P)^{abc}$

Alice side:	Bob side:
1: Select a random integer $a \in [1, r-1]$	1: Select a random integer $b \in [1, r-1]$
2: Send $Q_a = aP$ to Bob and Charlie	2: Send $Q_b = bP$ to Alice and Charlie
3: Upon reception of Q_b and Q_c, compute $K = e(bP,cP)^a$	3: Upon reception of Q_a and Q_c, compute $K = e(aP,cP)^b$

Charlie side:

1: Select a random integer $c \in [1, r-1]$
2: Send $Q_c = cP$ to Alice and Bob
3: Upon reception of Q_a and Q_b, compute $K = e(aP,bP)^c$

The security of Algorithm 2.8 relies on the impossibility of computing $e(P,P)^{abc}$ given P, aP, bP and cP by an eavesdropper. This is an instance of the so-called Bilinear Diffie-Hellman Problem, whose intractability is the security basis of many pairing-based protocols. As will be evident later, the hardness of this problem implies the hardness of the Diffie-Hellman Problem.

Definition 2.6. The Bilinear Diffie-Hellman Problem (denoted by BDHP) is the problem of computing $e(P,R)^{xy}$ given P, xP, yP and R.

Definition 2.7. The (Computational) Diffie-Hellman Problem (denoted by DHP) is the problem of computing xyP given P, xP, yP.

Note that if the DHP can be solved in G_1, the value xyP is available and $e(P,R)^{xy}$ can be easily computed as $e(xyP,R)$. A similar conclusion is achieved for G_T. Since the DHP can be easily solved if the DLP can be solved ($DLP_{assumption} \geq_P DHP_{assumption}$, that is, DHP is not harder than the DLP), it can be concluded that $DLP_{assumption} \geq_P DHP_{assumption} \geq_P BDHP_{assumption}$. Since nothing else is known about the difficulty of solving the BDHP, it is assumed to be as difficult as the DHP and that the security of pairing-based cryptographic schemes ultimately relies on the hardness of the DLP in G_1, G_2 and G_T.

Miller introduced in [Mil86b] an algorithm to evaluate rational functions on algebraic curves, enabling the efficient computation of pairings at linear complexity with respect to the input size (see also [Mil04]). Since then many optimizations have been proposed to improve the so called *Miller's algorithm* by, for instance, reducing the loop length [HSV06, LLP09, Ver10] or constructing pairing-friendly elliptic curves [BN05, BW05, SB06].

When $G_1 = G_2$ the pairing is called symmetric and is defined over supersingular curves. In this case, η_T pairing is arguably the most efficient algorithm [BGO+07]. If, otherwise, $G_1 \neq G_2$, the pairing is called asymmetric and is defined over ordinary elliptic curves. In this case, the optimized variants of the Tate pairing [BKL+02] (e.g., ate [HSV06], R-ate [LLP09], optimal ate [Ver10] pairing) achieve the highest performance.

In this work, we focus on the efficient implementation of asymmetric pairings with ordinary curves (see Chapter 6). Accordingly, we will assume the following groups for the construction of pairings: $G_1, G_2 =$ cyclic subgroups of $E(\mathbb{F}_p)$; $G_T =$ cyclic subgroup of $\mathbb{F}_{p^k}^*$.

For the case of ordinary curves, Barreto and Naehrig [BN05] proposed a large and easy-to-generate family of elliptic curves (called BN curves) with embedding degree $k = 12$, which is optimal for implementing pairings at the 128-bit security level. For our analysis and tests we choose the optimal ate pairing algorithm [Ver10]. We stress, however, that according to our tests other variants of the Tate pairing achieve similar performance on the targeted platforms (i.e., x86-64-based processors).

2.3.1. Optimal Ate Pairing on BN Curves

A Barreto-Naehrig (BN) curve has the form:

$$E_{BN}: \; y^2 = x^3 + b, \tag{2.17}$$

defined over $\mathbb{F}_p$ with $b \neq 0$ and embedding degree $k = 12$, where $p = 36u^4 + 36u^3 + 24u^2 + 6u + 1$, prime order $n = 36u^4 + 36u^3 + 18u^2 + 6u + 1$ and $u \in \mathbb{Z}$.

Let the map $\pi_p : E_{BN} \to E_{BN}$ be the p-power endomorphism $\pi_p(x, y) = (x^p, y^p)$, $E_{BN}[n]$ the n-torsion subgroup of E_{BN}, E'_{BN} the sextic twist $E'_{BN} / \mathbb{F}_{p^2}: \; y^2 = x^3 + b/\xi$ with ξ neither a cube nor a square in $\mathbb{F}_{p^2}$, $\mathrm{G}_1 = E_{BN}[n] \cap \mathrm{Ker}(\pi_p - [1]) = E_{BN}(\mathbb{F}_p)[n]$, G_2 the preimage $E'_{BN}(\mathbb{F}_{p^2})[n]$ of $E_{BN}[n] \cap \mathrm{Ker}(\pi_p - [p]) \subseteq E_{BN}(\mathbb{F}_{p^{12}})[n]$ and $\mathrm{G}_\mathrm{T} = \mu_n \subset \mathbb{F}^*_{p^{12}}$ the group of n-th roots of unity. The optimal ate pairing on equation (2.17) is defined as [NNS10]:

$$\begin{aligned} a_{opt} &: \mathrm{G}_2 \times \mathrm{G}_1 \to \mathrm{G}_\mathrm{T} \\ &(Q, P) \to \left(f_{r,Q}(P) \cdot l_{[r]Q, \pi_p(Q)}(P) \cdot l_{[r]Q + \pi_p(Q), -\pi_p^2(Q)}(P) \right)^{\frac{p^{12}-1}{n}}, \end{aligned} \tag{2.18}$$

where $r = 6u + 2 \in \mathbb{Z}$, $f_{r,Q}(P)$ is a normalized function with divisor $(f_{r,Q}) = r(Q) - ([r]Q) - (r-1)(\mathcal{O})$ and l_{Q_1, Q_2} is the line arising in the addition of points Q_1 and Q_2 evaluated at point P. Precisely, Miller's algorithm computes the function $f_{r,Q}$ using a double-and-add approach that involves the computation of point doublings, point additions and line evaluations. In Algorithm 2.9, the so-called *Miller loop* corresponds to lines 2-4. The pairing computation is completed by the

Algorithm 2.9. Optimal ate pairing on BN curves (including the case $u < 0$)

Input: $P \in \mathrm{G}_1, Q \in \mathrm{G}_2, r = |6u + 2| = \sum_{i=0}^{\log_2 r} r_i 2^i$

Output: $a_{opt}(Q, P)$

1: $T \leftarrow Q, \; f \leftarrow 1$
2: For $i = \lfloor \log_2 r \rfloor - 1$ downto 0 do
3: $\quad f \leftarrow f^2 \cdot l_{T,T}(P), T \leftarrow 2T$
4: $\quad$ if $r_i = 1$ then $f \leftarrow f \cdot l_{T,Q}(P), T \leftarrow T + Q$
5: $Q_1 \leftarrow \pi_p(P), \; Q_2 \leftarrow \pi_p^2(Q)$
6: If $u < 0$ then $T \leftarrow -T, \; f \leftarrow f^{-1}$
7: $f \leftarrow f \cdot l_{T,Q_1}(P), \; T \leftarrow T + Q_1$
8: $f \leftarrow f \cdot l_{T,-Q_2}(P), \; T \leftarrow T - Q_2$
9: $f \leftarrow f^{(p^6-1)(p^2+1)(p^4-p^2+1)/n}$
10: Return f

final exponentiation, which corresponds to line 9 in the same algorithm. Note that the power $(p^{12}-1)/n$ is factored in the exponents (p^6-1), (p^2+1) and $(p^4-p^2+1)/n$.

Chapter 3

New Precomputation Schemes

This chapter revisits the problem of calculating precomputations efficiently when the base point(s) is not known in advance. There are two standard table forms used by most elliptic curve scalar multiplication methods in the literature: d_iP and $c_iP + d_iQ$, where $c_i, d_i \in D = \{0, \pm 1, \pm 3, \pm 5, ..., \pm m\}$ with m odd. In the first case, it is required the on-line calculation of the non-trivial points d_iP, where $d_i \in D^+ \setminus \{0,1\} = \{3,5,...,m\}$ with m odd. In the second case, it is required (in the extreme case) the on-line calculation of the non-trivial points $c_iP \pm d_iQ$, where $c_i, d_i \in D^+ = \{0,1,3,5,...,m\}$, $c_i > 1$ if $d_i = 0$, $d_i > 1$ if $c_i = 0$, and m odd. The negative of these points can be computed on-the-fly at negligible cost. In the remainder, we will refer to these tables built with non-trivial points as simply d_iP and $c_iP \pm d_iQ$, respectively. Well-known methods to compute scalar multiplication using the former table are *w*NAF and Frac-*w*NAF in the case of single scalar multiplication, and the interleaving NAF method in the case of multiple scalar multiplication [HMV04]. Methods that employ a table with the form $c_iP \pm d_iQ$ are commonly intended for multiple scalar multiplication, such as the Joint Sparse Form (JSF) [Sol01] and its variants [KZZ04, OKN10].

In this chapter, we propose *two* novel methods for precomputing points and carry out an exhaustive analysis at different memory and security requirement levels:

- The first scheme, referred to as *Longa-Miri (LM) Scheme*, is based on the special addition with identical Z coordinate [Mel07] and is intended for tables with the form

d_iP using Jacobian coordinates on standard curves.

- The second scheme, referred to as *Longa-Gebotys (LG) Scheme*, is based on the concept of *conjugate addition in projective coordinates* and offers superior flexibility since it can be applied to any curve form and adapted to tables with forms d_iP and $c_iP \pm d_iQ$.

The different schemes are adapted and analyzed (whenever relevant) in *three* possible scenarios (see Section 3.1.1): case 1, without using inversions; case 2, using only one inversion; and case 3, using multiple inversions. The analysis of the proposed schemes includes *three* curves of interest: Weiertrass curves using Jacobian coordinates $\mathcal{J}$, extended Jacobi quartics using extended Jacobi quartic coordinates $\mathcal{JQ}^e$, and Twisted Edwards curves using inverted Edwards coordinates $\mathcal{IE}$.

This chapter is organized as follows. §3.1 discusses the most relevant previous work. §3.2 introduces the LM precomputation scheme for standard curves using Jacobian coordinates, targeting the single scalar multiplication case. §3.3 introduces the LG precomputation scheme and discusses its applicability to different curves forms for both single and multiple scalar multiplication. §3.4 presents the performance analysis of the proposed schemes, including detailed comparisons with previous methods. §3.5 discusses other applications for conjugate additions. And, finally, some conclusions are drawn in §3.6.

3.1. Previous Work

In this section we summarize most relevant efforts in the literature to reduce the time complexity of the precomputation stage of scalar multiplication. We also recall the special addition by Meloni [Mel07], which is used here to build a novel precomputation scheme. We remark that, in the present work, we focus on methods that are efficient when the initial point P in the computation kP is not known in advance.

3.1.1. Precomputation for Single Scalar Multiplication

The most commonly used precomputation table has the form d_iP, where $d_i \in D^+ \setminus \{0,1\} = \{3,5,...,m\}$, for some odd integer m. This table form can be found in most algorithms to compute scalar multiplication such as the wNAF and Frac-wNAF methods (see Section 2.2.4.3).

The traditional approach is to compute the points by following the sequence $P \rightarrow 3P \rightarrow 5P \rightarrow \ldots \rightarrow mP$ with the application of an addition with $2P$ at each step. Depending on the coordinate system(s) applied for the calculation, we can distinguish *three* different cases:

- *Case 1:* points are precomputed and left in some projective system. This scenario has the

potential advantage of having very low cost because no additional coordinate system conversion is required. However, because points are left in certain projective system, additions during the evaluation stage have general form and one cannot make use of efficient mixed addition or mixed doubling-addition operations.

- *Case 2:* points are computed in some projective system and then converted to affine coordinates. The latter step is usually performed with the Montgomery' simultaneous inversion method in order to reduce the number of inversions (see Alg. 2.26 of [HMV04]). In this scenario, precomputation cost is higher because of the conversion to affine step. However, the use of mixed additions (or mixed doubling-additions) allows reducing costs during the evaluation stage.
- *Case 3:* points are computed and left in affine coordinates. This case is probably the most expensive approach of all three cases in terms of speed, mainly because inversion is especially expensive over prime fields. One potential advantage of this approach is that memory requirement is kept to a minimal.

Cases 1 and 2 were studied by Cohen et al. [CMO98] when proposed the use of mixed coordinates to implement scalar multiplication on Weiertrass curves. In particular, Cohen et al. proposed two alternatives using different coordinate systems: $(C^1, C^2, C^3) = (\mathcal{J}^m, \mathcal{J}, \mathcal{C})$ and $(C^1, C^2, C^3) = (\mathcal{J}^m, \mathcal{J}, \mathcal{A})$, where C^1 represents the system to perform doublings, C^2 represents the system for every doubling before an addition, and C^3 represents the system to perform additions (in the evaluation and precomputation stages). In particular, the first approach, which computes precomputations in $\mathcal{C}$ coordinates (corresponding to case 1), was shown to be more efficient than the second approach using $\mathcal{A}$ coordinates combined with the Montgomery' simultaneous inversion method (corresponding to case 2) always that $1I > 30M$ approximately.

Nevertheless, the conclusions drawn in [CMO98] are somewhat outdated because $\mathcal{J}^m$ coordinates (proposed for the evaluation stage in both cases) do not provide any advantage if $a = -3$, as discussed in Section 2.2.4.2. Also, Cohen et al.'s approach to case 2 involves the use of Montgomery's method over groups of points. However, a more popular alternative in recent years has been to apply the method to *all* points in the table so that the number of inversions is limited to only *one*. In this scenario, possible approaches are to compute precomputed points in $\mathcal{J}$, $\mathcal{C}$ or $\mathcal{H}$ coordinates and then use Montgomery's method over all the partial points.

Very recently, Dahmen et al. [DOS07] proposed a highly efficient method (called the DOS method) and showed that it is more cost-effective than all other previous schemes using one inversion (case 2). Also, when compared to the approach using only $\mathcal{A}$ coordinates (case 3), the DOS method exhibits superior performance for a wide range of *I/M* ratios. The DOS method's cost is $1I + (10L-1)M + (4L+4)S$, where $L = (m-1)/2$ is the number of non-trivial points in the table, and it has a memory requirement of $(2L+4)$ registers (in this thesis, we assume that

each "register" can store a field element). One disadvantage of the DOS method is that there is no straightforward version to compute points as in case 1.

3.1.2. Special Addition with Identical *Z* Coordinate

The following formula was proposed by Meloni in [Mel07]. Let $P=(X_1:Y_1:Z)$ and $Q=(X_2:Y_2:Z)$ be two points with the same Z coordinate in $\mathcal{J}$ on an elliptic curve E_W defined over $\mathbb{F}_p$. The addition $P+Q=(X_3:Y_3:Z_3)$ can be obtained as follows:

$$\begin{aligned} X_3 &= (Y_2-Y_1)^2-(X_2-X_1)^3-2X_1(X_2-X_1)^2, \\ Y_3 &= (Y_2-Y_1)\left[X_1(X_2-X_1)^2-X_3\right]-Y_1(X_2-X_1)^3, \\ Z_3 &= Z(X_2-X_1). \end{aligned} \tag{3.1}$$

Remarkably, Meloni also noticed that one can extract from (3.1) a new representative of $P=(X_1:Y_1:Z)$ given by $\left(X_1(X_2-X_1)^2, Y_1(X_2-X_1)^3, Z(X_2-X_1)\right)$, which has identical Z coordinate to $P+Q=(X_3:Y_3:Z_3)$. So one can continue applying the same formula recursively.

The new addition only costs $5M+2S$, which represents a significant reduction in comparison with $8M+3S$ (or $7M+4S$), corresponding to the mixed Jacobian-affine addition (see Table 2.2). Unfortunately, it is not possible to directly replace traditional additions with this special operation since, obviously, it is expected that additions are computed over operands with different Z coordinates during standard scalar multiplication. Hence, Meloni [Mel07] applied his formula to the context of scalar multiplication with star addition chains, where the particular sequence of operations allows the replacement of each traditional addition by formula (3.1) (referred to as $\text{ADD}_{\text{Co-}Z}$ for the remainder, borrowing notation from [GMJ10]).

Nevertheless, the author noticed in [Lon07] that the new addition can in fact be useful to devise new formulas for composite operations such as doubling-addition that are applicable to traditional scalar multiplication methods (see [Lon07] and also [LM08b]).

In Section 3.2, we again exploit the $\text{ADD}_{\text{Co-}Z}$ operation to build low-cost precomputation tables. The new approach is called *LM Scheme*, offers very low cost and can be easily adapted to cases 1 and 2, exhibiting higher performance and flexibility than the DOS method.

3.1.3. Precomputation for Special Curves and Multiple Scalar Multiplication

To the best of our knowledge, most research in the literature has only explored the efficiency of precomputation schemes on standard curves of Weierstrass form (2.4). Although the traditional

sequence $P \rightarrow 3P \rightarrow 5P \rightarrow \ldots \rightarrow mP$ can be easily adapted to special curves, it is still lacking a thorough performance analysis. In this work, we derive for first time the costs involved in computing the precomputed table on certain special curves using the traditional sequence and applying the most efficient point operations at our disposal. Moreover, we propose a new scheme based on the concept of "conjugate" additions (see Section 3.3). The new method is called *LG Scheme* and is shown to achieve the lowest costs on extended Jacobi quartics using $\mathcal{JQ}^e$ coordinates and Twisted Edwards curves using $\mathcal{IE}$ coordinates.

For the case of multiple scalar multiplication, JSF-based methods need the calculation of a table of the form $c_i P \pm d_i Q$, where $c_i, d_i \in \{0,1,3,5,...,m\}$ for some odd integer value m [KZZ04]. In [OTV05], Okeya et al. observed that an inversion can be saved when computing $P \pm Q$ in affine coordinates (which can be seen as an addition/conjugate addition in $\mathcal{A}$). However, the derived scheme was basically intended for implementations using the affine representation only and, hence, inefficient when compared to cases using some projective system over prime fields. Recently, Järvinen et al. [JFS07] extended Okeya et al.'s idea of exploiting redundancies in affine formulae to precompute points. To get rid of the multiple inversions, they took advantage of Montgomery' simultaneous inversion method and derived an efficient scheme for a table with the form $dP \pm lQ \pm kR$, where $d,l,k \in \{0,1\}$. Hence, in its actual format their methodology only applies to that specific table form and is expected to be efficient on standard curves only since it is still based on affine formulae.

Because the concept of "conjugate" addition, as discussed in this work, takes advantage of redundancies in the computation of $P \pm Q$ in *projective coordinates*, it naturally applies to precomputation tables that appear in multiple scalar multiplication algorithms and enables efficient computation over prime fields. In Section 3.3.3, we analyze the savings achieved with this approach and show its advantages in terms of computing cost. Moreover, we analyze for first time the performance of precomputation methods on certain special curves in this setting. Specifically, we study the case with Jacobi quartics using $\mathcal{JQ}^e$ coordinates and Twisted Edwards curves using $\mathcal{IE}$ coordinates.

NOTE: Okeya et al.'s idea is similar to the proposed concept of "conjugate" addition. However, their observation was restricted to affine coordinates whereas we discovered the idea of saving operations in the computation $P \pm Q$ when observing redundancies in projective coordinate formulae. In general, projective coordinates are largely preferred over prime fields (especially on special curves), so savings in these settings are more valuable.

For the remainder of this chapter, we assume that curve parameters for the curves under analysis can be chosen such that the cost of multiplying a curve constant can be considered negligible in comparison with a regular multiplication. Also, in most cases additions and

subtractions are neglected in our cost analysis. These assumptions greatly simplify our analysis without affecting the conclusions.

3.2. Precomputation Scheme Based on the Addition with Identical *Z* Coordinate: LM Scheme

The proposed scheme, computes the precomputed table as follows:

$$d_i P = 2P + \ldots + 2P + 2P + P \,, \tag{3.2}$$

performing additions from right to left. We will show that all the additions in (3.2) can be computed with the $\mathrm{ADD}_{\text{Co-}Z}$ operation proposed by Meloni [Mel07], reducing costs in comparison with previous approaches.

The direct scheme applying (3.2) and calculating the points in $\mathcal{J}$ coordinates is referred to as LM Scheme, case 1. Furthermore, although the author proposed in [Lon07, Section 3.4.1] a version of the method using only one inversion (case 2), in this work we observe that some values computed during the aforementioned additions can be efficiently exploited to minimize costs during conversion to $\mathcal{A}$ coordinates. In this regard, we present *two* new and optimized schemes which are referred to as LM Scheme, cases 2a and 2b.

3.2.1. Method Description

Our method can be summarized in the following two steps.

Step 1: Computation of precomputed points in Jacobian coordinates

Point *P* is assumed to be initially in $\mathcal{A}$ coordinates. By applying the mixed coordinates approach proposed in [CMO98], we can compute the point $2P$ required in (3.2) in $\mathcal{J}$ as follows:

$$X_2 = \alpha^2 - 2\beta \,,\;\; Y_2 = \alpha(\beta - X_2) - y_1^4 \,,\;\; Z_2 = y_1 \,, \tag{3.3}$$

where $\alpha = (3x_1^2 + a)/2$, $\beta = \left[(x_1 + y_1^2)^2 - x_1^2 - y_1^4\right]/2$, and the input and result are $P = (x_1, y_1)$ and $2P = (X_2 : Y_2 : Z_2) \in E(\mathbb{F}_p)$, respectively. Formula (3.3) can be easily derived from the doubling formula (5.2), Section 5.4, by setting $Z_1 = 1$, and has a cost of only $1M + 5S + 12A$. Note that, if $1M - 1S < 4A$, then computing β as $x_1 \cdot y_1^2$ is more efficient with a total cost of $2M + 4S + 8A$.

Then, by fixing $\lambda = y_1$ in (2.10) we can set a point $P^{(1)}$ equivalent to *P* given by:

$$P^{(1)} = \left(X_1^{(1)}, Y_1^{(1)}, Z_1^{(1)}\right) = \left(x_1 y_1^2, y_1^4, y_1\right) \equiv P = (x_1, y_1, 1) \,,$$

whose computation does not involve additional costs since its coordinates have already been computed in (3.3). Following additions to compute points $d_i P$ are performed using the special addition $\mathrm{ADD}_{\text{Co-}Z}$ as follows:

1st Compute $3P = 2P + P^{(1)} = (X_2, Y_2, Z_2) + \left(X_1^{(1)}, Y_1^{(1)}, Z_1^{(1)}\right) = (X_3, Y_3, Z_3)$:

$$X_3 = \left(Y_1^{(1)} - Y_2\right)^2 - \left(X_1^{(1)} - X_2\right)^3 - 2X_2\left(X_1^{(1)} - X_2\right)^2,$$

$$Y_3 = \left(Y_1^{(1)} - Y_2\right)\left(X_2\left(X_1^{(1)} - X_2\right)^2 - X_3\right) - Y_2\left(X_1^{(1)} - X_2\right)^3,$$

$$Z_3 = Z_2\left(X_1^{(1)} - X_2\right).$$

2nd Fix $2P^{(1)} = \left(X_2^{(1)}, Y_2^{(1)}, Z_2^{(1)}\right) = \left(X_2\left(X_1^{(1)} - X_2\right)^2, Y_2\left(X_1^{(1)} - X_2\right)^3, Z_2\left(X_1^{(1)} - X_2\right)\right) \equiv (X_2, Y_2, Z_2)$, and compute $5P = 2P^{(1)} + 3P = \left(X_2^{(1)}, Y_2^{(1)}, Z_2^{(1)}\right) + (X_3, Y_3, Z_3) = (X_4, Y_4, Z_4)$:

$$X_4 = \left(Y_3 - Y_2^{(1)}\right)^2 - \left(X_3 - X_2^{(1)}\right)^3 - 2X_2^{(1)}\left(X_3 - X_2^{(1)}\right)^2,$$

$$Y_4 = \left(Y_3 - Y_2^{(1)}\right)\left(X_2^{(1)}\left(X_3 - X_2^{(1)}\right)^2 - X_4\right) - Y_2^{(1)}\left(X_3 - X_2^{(1)}\right)^3,$$

$$Z_4 = Z_2^{(1)}\left(X_3 - X_2^{(1)}\right), \quad A_4 = \left(X_3 - X_2^{(1)}\right), \quad B_4 = \left(X_3 - X_2^{(1)}\right)^2, \quad C_4 = \left(X_3 - X_2^{(1)}\right)^3.$$

$\vdots$

$\left(\frac{m-1}{2}\right)^{\text{th}}$ Fix $2P^{((m-3)/2)} = \left(X_2^{((m-3)/2)}, Y_2^{((m-3)/2)}, Z_2^{((m-3)/2)}\right) = \left(X_2^{((m-5)/2)}\left(X_{(m-1)/2} - X_2^{((m-5)/2)}\right)^2, Y_2^{((m-5)/2)}\left(X_{(m-1)/2} - X_2^{((m-5)/2)}\right)^3, Z_2^{((m-5)/2)}\left(X_{(m-1)/2} - X_2^{((m-5)/2)}\right)\right) \equiv \left(X_2^{((m-5)/2)}, Y_2^{((m-5)/2)}, Z_2^{((m-5)/2)}\right)$, and compute $mP = 2P^{((m-3)/2)} + (m-2)P = \left(X_2^{((m-3)/2)}, Y_2^{((m-3)/2)}, Z_2^{((m-3)/2)}\right) + \left(X_{(m+1)/2}, Y_{(m+1)/2}, Z_{(m+1)/2}\right) = \left(X_{(m+3)/2}, Y_{(m+3)/2}, Z_{(m+3)/2}\right)$:

$$X_{(m+3)/2} = \left(Y_{(m+1)/2} - Y_2^{((m-3)/2)}\right)^2 - \left(X_{(m+1)/2} - X_2^{((m-3)/2)}\right)^3 - 2X_2^{((m-3)/2)}\left(X_{(m+1)/2} - X_2^{((m-3)/2)}\right)^2,$$

$$Y_{(m+3)/2} = \left(Y_{(m+1)/2} - Y_2^{((m-3)/2)}\right)\left(X_2^{((m-3)/2)}\left(X_{(m+1)/2} - X_2^{((m-3)/2)}\right)^2 - X_{(m+3)/2}\right) - \ldots$$
$$\ldots Y_2^{((m-3)/2)}\left(X_{(m+1)/2} - X_2^{((m-3)/2)}\right)^3,$$

$$Z_{(m+3)/2} = Z_2^{((m-3)/2)}\left(X_{(m+1)/2} - X_2^{((m-3)/2)}\right), \quad A_{(m+3)/2} = \left(X_{(m+1)/2} - X_2^{((m-3)/2)}\right),$$

$$B_{(m+3)/2} = \left(X_{(m+1)/2} - X_2^{((m-3)/2)}\right)^2, \quad C_{(m+3)/2} = \left(X_{(m+1)/2} - X_2^{((m-3)/2)}\right)^3.$$

Intermediate values A_i and (B_i, C_i), for i = 4 to $(m+3)/2$, are stored for LM Scheme, cases 2a and 2b, respectively, and used in *Step 2* to save some computations when converting points to $\mathcal{A}$ coordinates. Note that the LM Scheme, case 1, does not require neither storing values

(A_i, B_i, C_i) nor executing *Step 2*.

Step 2: Conversion to affine coordinates (cases 2a and 2b only)

This step involves the conversion from $\mathcal{J}$ to $\mathcal{A}$ of points $(X_i : Y_i : Z_i)$ computed in *Step 1*, for i = 3 to $(m+3)/2$, $m > 3$, enabling the use of the efficient mixed addition operation during the evaluation stage of scalar multiplication.

Conversion from $\mathcal{J}$ to $\mathcal{A}$ is achieved by applying $(X_i/Z_i^2, Y_i/Z_i^3, 1)$ (see Section 2.2.4.2). Then, to avoid the computation of several expensive inversions we use a modified version of the Montgomery's method of simultaneous inversion to limit the requirement to only *one* inversion for all the points in the precomputed table $d_i P$.

In LM Scheme, case 2a, we first compute the inverse $r = Z_{(m+3)/2}^{-1}$, and then recover every point using $(X_i/Z_i^2, Y_i/Z_i^3, 1)$ as follows:

mP: $\quad x_{(m+3)/2} = r^2 \cdot X_{(m+3)/2}$, $\quad y_{(m+3)/2} = r^3 \cdot Y_{(m+3)/2}$,

$(m–2)P$: $\quad r = r \cdot A_{(m+3)/2}$, $\quad x_{(m+1)/2} = r^2 \cdot X_{(m+1)/2}$, $\quad y_{(m+1)/2} = r^3 \cdot Y_{(m+1)/2}$,

$\vdots$ $\qquad\qquad$ $\vdots$

$3P$: $\quad r = r \cdot A_4$, $\quad x_3 = r^2 \cdot X_3$, $\quad y_3 = r^3 \cdot Y_3$.

It is important to observe that $Z_j = Z_3 \times \prod_{i=4}^{j} A_i$, for $j = 4$ to $(m+3)/2$, according to *Step 1* and, hence, for $i = (m-2), (m-4), \ldots, 3$, the value $Z_{(i+3)/2}^{-1}$ for each point iP is recovered at every multiplication $r \cdot A_{(i+5)/2}$.

For LM Scheme, case 2b, we first compute $r_1 = \left(Z_{(m+3)/2}^{-1}\right)^2$ and $r_2 = \left(Z_{(m+3)/2}^{-1}\right)^3$, and then recover every point using $(X_i/Z_i^2, Y_i/Z_i^3, 1)$ as follows:

mP: $\quad x_{(m+3)/2} = r_1 \cdot X_{(m+3)/2}$, $\quad y_{(m+3)/2} = r_2 \cdot Y_{(m+3)/2}$,

$(m–2)P$: $\quad r_1 = r_1 \cdot B_{(m+3)/2}$, $\quad r_2 = r_2 \cdot C_{(m+3)/2}$, $\quad x_{(m+1)/2} = r_1 \cdot X_{(m+1)/2}$, $\quad y_{(m+1)/2} = r_2 \cdot Y_{(m+1)/2}$,

$\vdots$ $\qquad\qquad$ $\vdots$

$3P$: $\quad r_1 = r_1 \cdot B_4$, $\quad r_2 = r_2 \cdot C_4$, $\quad x_3 = r_1 \cdot X_3$, $\quad y_3 = r_2 \cdot Y_3$.

In this case: $Z_j^2 = Z_3^2 \times \prod_{i=4}^{j} B_i$ and $Z_j^3 = Z_3^3 \times \prod_{i=4}^{j} C_i$, for $j = 4$ to $(m+3)/2$, according to *Step 1* and, hence, for $i = (m-2), (m-4), \ldots, 3$, the pair $(Z_{(i+3)/2}^{-2}, Z_{(i+3)/2}^{-3})$ for each point iP is recovered with $r_1 \cdot B_{(i+5)/2}$ and $r_2 \cdot C_{(i+5)/2}$.

The reader is referred to Appendix A1 for the detailed pseudocode of the LM Scheme.

3.2.2. Cost Analysis

The cost of the LM Scheme, case 1, is given by:

$$\text{Cost}_{\text{LM Scheme, case 1}} = (6L+1)M + (3L+5)S, \tag{3.4}$$

where $L=(m-1)/2$ is the number of non-trivial points in the table $d_i P$. The cost (3.4) assumes the use of the addition (or doubling-addition) with stored values during the evaluation stage that requires precalculating values Z_i^2 and Z_i^3 (see Table 2.2). Otherwise, the cost can be reduced to only $(5L+1)M + (2L+5)S$. In terms of memory usage (for temporary calculations and point storage), LM Scheme, case 1, requires $(5L+6)$ registers if using the addition or doubling-addition with stored values or $(3L+6)$ registers if using operations without stored values.

The LM Scheme, case 2a, has the following cost:

$$\text{Cost}_{\text{LM Scheme, case 2a}} = 1I + (9L)M + (3L+5)S, \tag{3.5}$$

In terms of memory usage, LM Scheme, case 2a, requires $(3L+3)$ registers overall. In the case of LM Scheme, case 2b, the cost is as follows:

$$\text{Cost}_{\text{LM Scheme, case 2b}} = 1I + (9L)M + (2L+6)S, \tag{3.6}$$

For this scheme, we require $(4L+1)$ registers when $L > 1$. For $L = 1$, the requirement is fixed at 6 registers. It will be shown later that memory requirements of cases 2a and 2b do not exceed the memory allocated for scalar multiplication for small or intermediate values of L, whereas case 1 does not exceed memory constraints in any case. For the detailed estimation of costs and memory requirements of the LM Scheme, cases 1, 2a and 2b, please refer to Appendix A2.

For the record, the original scheme in [Lon07] has a cost of $1I + (11L+2)M + (3L+5)S$. As can be seen in (3.5) and (3.6), the new LM Scheme variants represent an important improvement in terms of computing cost. In particular, case 2b achieves the lowest cost in scenarios using one inversion at the expense of some extra memory.

Next, we analyze the memory requirements for scalar multiplication and determine if our method adjusts to such constraints.

In the case of using general (doubling-additions) additions or general (doubling-additions) additions with stored values for the evaluation stage (i.e., case 1), scalar multiplication requires in total ($3L+R$) or ($5L+R$) registers, respectively, where R is the number of registers needed by the most memory-consuming point operation in a given implementation. In scalar multiplications using solely radix 2, addition and doubling-addition are usually such operations. Depending on the implementation details, these operations can require up to 8 registers [Lon08]. Consequently, the LM Scheme, case 1, adjusts to the above requirements as it always holds that $3L + 6 \leq 3L + R$

and $5L+6 \le 5L+R$ for the two aforementioned cases.

In the case of using mixed additions (or mixed doubling-addition) during evaluation (i.e., case 2), the total requirement of scalar multiplication is given by (2*L*+*R*) registers. Thus, LM Scheme, case 2b, adjusts to the previous requirements for small precomputed tables. If mixed addition or doubling-addition is the most memory-consuming operation then $4L+1 \le 2L+7$ for $L \le 3$. A similar analysis for case 2a allows us to verify that this scheme adjusts to memory constraints for $L \le 4$, which demonstrates that it is efficient for practical implementations based on fractional windows if n = 160 bits. Although cases 2a and 2b require more memory resources for higher values of L necessary in 256- and 512-bit scalar multiplications, we show in Section 3.4.1 that these schemes still achieve the lowest costs for most scenarios for equivalent memory constraints.

In Section 3.4, we analyze in great detail the performance of the proposed method in comparison with the best previous efforts on standard curves.

3.3. Precomputation Scheme based on Conjugate Additions: LG Scheme

The proposed scheme is based on the following simple observation: if $P+Q$ has been computed for two distinct points P, Q, the subtraction of those points only requires a few additional field operations. In the remainder, we will refer to this operation, namely $P-Q$ $(=P+(-Q))$, as "conjugate" addition and denote it by ADD′. It will turn out that this operation allows the efficient computation of precomputed tables.

Next, we describe the strategy of the conjugate addition in projective coordinates, and then discuss its application to computing tables of the form d_iP and $c_iP \pm d_iQ$.

3.3.1. The Strategy: Conjugate Addition using Projective Coordinates

First, $P-Q = P+(-Q)$. As the negative of a point only involves the change of at most one of the coordinate values in the projective representation (see Sections 2.2.4.2 and 2.2.5), it is then expected than computing $P+Q$ and $P-Q$ share most of the intermediate computations.

Let us illustrate the latter with the point addition formula using $\mathcal{J}$ coordinates. Let $P=(X_1:Y_1:Z_1)$ and $Q=(X_2:Y_2:Z_2)$ be two points on an elliptic curve $E_W/\mathbb{F}_p$. If the addition $P+Q=(X_3:Y_3:Z_3)$ is performed using the optimized addition formula:

$$X_3=\alpha^2-\beta^3-2Z_2^2X_1\beta^2,\ \ Y_3=\alpha(Z_2^2X_1\beta^2-X_3)-Z_2^3Y_1\beta^3,\ \ Z_3=\theta\beta, \tag{3.7}$$

where $\alpha=Z_1^3Y_2-Z_2^3Y_1$, $\beta=Z_1^2X_2-Z_2^2X_1$ and $\theta=[(Z_1+Z_2)^2-Z_1^2-Z_2^2]/2$, then $P-Q$ can be

computed as $P+(-Q)=(X_1:Y_1:Z_1)+(X_2:-Y_2:Z_2)=(X_4:Y_4:Z_4)$ reusing the partial values $(\beta^3+2Z_2^2X_1\beta^2)$, $Z_2^2X_1\beta^2$, $Z_2^3Y_1\beta^3$, Z_3, $Z_1^3Y_2$ and $Z_2^3Y_1$. Thus, the conjugate addition can be computed with the following:

$$X_4=\gamma^2-\beta^3-2Z_2^2X_1\beta^2,\ \ Y_4=\gamma(X_4-Z_2^2X_1\beta^2)-Z_2^3Y_1\beta^3,\ \ Z_4=Z_3, \tag{3.8}$$

where $\gamma=Z_1^3Y_2+Z_2^3Y_1$. Note that (3.8) only involves the extra cost of $1M+1S$, which is significantly less than the cost of a general addition (3.7) (i.e., $11M+5S$). If we also consider other usually neglected operations, the cost drops from $11M+5S+9A+1(\times 2)+1(\div 2)$ to only $1M+1S+4A$. In total, the addition/conjugate addition pair costs $12M+6S+13A+1(\times 2)+1(\div 2)$.

It may seem that performing this conjugate operation would involve several extra registers to store partial values temporarily. However, memory requirements can be minimized by performing $P+Q$ and $P-Q$ concurrently. For instance, a possible execution sequence for computing $P\pm Q$ using formulas (3.7) and (3.8) would be as the one shown in Table 3.1.

The execution of the addition/conjugate addition pair detailed in Table 3.1 requires 8 registers only (including temporary registers and registers storing input/output coordinates), which is the same memory requirement of the addition formula alone. Thus, executing the conjugate addition does not increase the memory consumption in this case. Similar results are expected for other coordinate systems.

Table 3.1. Pseudocode of an "interlaced" execution of an addition/conjugate addition pair in $\mathcal{J}$.

INPUT: $P=(X_1:Y_1:Z_1)$ and $Q=(X_2:Y_2:Z_2)$; $T_1\leftarrow X_1$, $T_2\leftarrow Y_1$, $T_3\leftarrow Z_1$, $T_4\leftarrow X_2$, $T_5\leftarrow Y_2$, $T_6\leftarrow Z_2$
OUTPUT: $P+Q=(X_3:Y_3:Z_3)=(T_1:T_2:T_3)$ and $P-Q=(X_4:Y_4:Z_4)=(T_4:T_5:T_3)$

1. $T_7=T_3^2$ $\{Z_1^2\}$	12. $T_8=T_1\times T_8$ $\{Z_2^2X_1\}$	23. $T_1=T_6^2$ $\{\alpha^2\}$
2. $T_4=T_4\times T_7$ $\{Z_1^2X_2\}$	13. $T_7=T_4-T_8$ $\{\beta\}$	24. $T_1=T_1-T_4$ $\{X_3\}$
3. $T_8=T_3\times T_7$ $\{Z_1^3\}$	14. $T_3=T_3/2$ $\{\theta\}$	25. $T_7=T_2\times T_7$ $\{Z_2^3Y_1\beta^3\}$
4. $T_5=T_5\times T_8$ $\{Z_1^3Y_2\}$	15. $T_3=T_3\times T_7$ $\{Z_3=Z_4\}$	26. $T_2=T_8-T_1$ $\{Z_2^2X_1\beta^2-X_3\}$
5. $T_8=T_6^2$ $\{Z_2^2\}$	16. $T_6=T_7^2$ $\{\beta^2\}$	27. $T_2=T_2\times T_6$ $\{\alpha(Z_2^2X_1\beta^2-X_3)\}$
6. $T_7=T_7+T_8$ $\{Z_1^2+Z_2^2\}$	17. $T_7=T_6\times T_7$ $\{\beta^3\}$	28. $T_2=T_2-T_7$ $\{Y_3\}$
7. $T_3=T_3+T_6$ $\{Z_1+Z_2\}$	18. $T_8=T_6\times T_8$ $\{Z_2^2X_1\beta^2\}$	29. $T_6=T_5^2$ $\{\gamma^2\}$
8. $T_3=T_3^2$ $\{\omega=(Z_1+Z_2)^2\}$	19. $T_4=2T_8$ $\{2Z_2^2X_1\beta^2\}$	30. $T_4=T_6-T_4$ $\{X_4\}$
9. $T_3=T_3-T_7$ $\{\omega=Z_1^2-Z_2^2\}$	20. $T_4=T_4+T_7$ $\{\beta^3+2Z_2^2X_1\beta^2\}$	31. $T_8=T_4-T_8$ $\{X_4-Z_2^2X_1\beta^2\}$
10. $T_6=T_6\times T_8$ $\{Z_2^3\}$	21. $T_6=T_5-T_2$ $\{\alpha\}$	32. $T_8=T_5\times T_8$ $\{\gamma(X_4-Z_2^2X_1\beta^2)\}$
11. $T_2=T_2\times T_6$ $\{Z_2^3Y_1\}$	22. $T_5=T_5+T_2$ $\{\gamma\}$	33. $T_5=T_8-T_7$ $\{Y_4\}$

We have derived conjugate addition formulas in projective coordinates ($\mathcal{J}$, $\mathcal{JQ}^e$ and $\mathcal{IE}$ coordinates) and in affine for the three curves under analysis. The costs of the different variants of addition/conjugate addition pairs are summarized in Table 3.2. Note that, in some cases, the traditional operations have been slightly modified so that the cost of the overall formula is minimized. Refer to Appendix A3 for complete details.

Table 3.2. Costs of addition/conjugate addition formulas using projective ($\mathcal{J}$, $\mathcal{IE}$ and $\mathcal{JQ}^e$) and affine coordinates.

Point Operation	Cost		
	Standard curve ($a=-3$), $\mathcal{J}$	Twisted Edwards ($a=1$), $\mathcal{IE}$	Ext. Jacobi quartic ($d=1$), $\mathcal{JQ}^e$
ADD-ADD′, $\mathcal{P} \pm \mathcal{P} \rightarrow \mathcal{P}$	$12M + 6S$	$14M + 1S$	$9M + 5S$
ADD-ADD′$_{[M,S]}$, $\mathcal{P} \pm \mathcal{P} \rightarrow \mathcal{P}$	$11M + 5S$	-	$9M + 4S$
ADD-ADD′$_{[2,2]}$, $\mathcal{P} \pm \mathcal{P} \rightarrow \mathcal{P}$	$10M + 4S$	-	-
mADD-mADD′, $\mathcal{P} \pm \mathcal{A} \rightarrow \mathcal{P}$	$8M + 5S$	$13M + 1S$	$8M + 4S$
mmADD-mmADD′, $\mathcal{A} \pm \mathcal{A} \rightarrow \mathcal{P}$	$5M + 3S$	$11M$	$6M + 4S$
ADD-ADD′, $\mathcal{A} \pm \mathcal{A} \rightarrow \mathcal{A}$	$1I + 4M + 2S$ (1)	$1I + 13M + 1S$	$1I + 10M + 4S$

(1) Formula in affine coordinates from [OTV05]. $\mathcal{P}$: projective coordinates ($\mathcal{J}$, $\mathcal{JQ}^e$ or $\mathcal{IE}$).

In the following section, we introduce novel precomputation schemes for tables with the forms d_iP and $c_iP \pm d_iQ$ that take advantage of the new conjugate formulas. We again consider all *three* precomputation scenarios, i.e., cases 1, 2 and 3.

3.3.2. Precomputation Scheme for Table of the Form d_iP

We propose a recursive scheme that first reaches a "strategic" point and then applies efficiently the conjugate addition technique described in the previous section. In the following, we define as "strategic" to those points that can be efficiently computed and from which it is possible to calculate the maximum possible number of precomputed points at the lowest cost. The steps of our scheme are detailed in the following.

Step 1: Computation of precomputed points

The main body of our scheme is detailed in Algorithm 3.1. In this step, points can be computed in projective coordinates using operations from Tables 2.2, 2.3 or 2.4 (case 1), or directly in $\mathcal{A}$ coordinates (case 3). If projective points are to be converted to $\mathcal{A}$ (case 2) then *Step 2* should be

executed right after.

Basically, Algorithm 3.1 first reaches certain "strategic" point and then computes all the points that are close to it by efficiently computing additions and conjugate additions. The "strategic" points $\mathcal{S}$ proposed in our scheme have the form $P_{i+1} = 2P_i$, for $i \in \mathbb{Z} \geq 0$ and $P_0 = 3P$ (that is, $\mathcal{S} = \{6P, 12P, 24P, \ldots, rP, \ldots, r_{\max}P\}$), which are computed using a combination of one tripling (performed at the beginning; step 2 of Algorithm 3.1) and a sequence of doublings (step 11). Note that there is a minimum number of close points that makes the computation of a "strategic" point worthwhile. If that minimum is not fulfilled (evaluation in step 5) then the algorithm calculates the remaining points from the previous "strategic" point (loop in steps 6-8). The value of such a minimum depends on the particular costs of point operations. For $\mathcal{J}$, $\mathcal{JQ}^e$ and $\mathcal{IE}$ coordinates, we have determined that the lowest cost is achieved if the next "strategic" point is computed always that the value m is greater or equal to such a "strategic" point (i.e., if $m \geq 2r$), in which case steps 10-19 are executed.

Let us illustrate the proposed scheme with the following example.

Example 3.1. If $m = 13$, Algorithm 3.1 computes the first points as $P \rightarrow 3P \rightarrow 6P$, where $6P$ is the first "strategic" point. From this, $5P$ and $7P$ (*close* points) are calculated by adding $6P + (-P)$ and $6P + P$. Note that the latter operations can be calculated with a low cost addition/conjugate addition pair. Then, Algorithm 3.1 calculates the following "strategic" point (since $m > 12$) by doubling $6P \rightarrow 12P = r_{\max}P$, and finally computes close points $9P$, $11P$ and $13P$ by performing $12P + (-3P)$, $12P + (-P)$ and $12P + P$, respectively. Again the last two operations can also be computed with an addition/conjugate addition pair.

In Appendix A4, we have sketched the derivation of points for tables with different values m. Note that the described method does not include the case $m = 5$. For a table with $m = 5$, $\mathcal{JQ}^e$ and $\mathcal{J}$ coordinates, it is more efficient to compute points by performing $P \rightarrow 2P \rightarrow 4P$, and then obtaining $3P$ and $5P$ with an addition/conjugate addition pair (i.e., $4P + (-P)$ and $4P + P$). For the case $\mathcal{IE}$, we suggest to compute the table following the sequence $P \rightarrow 2P \rightarrow 3P \rightarrow 5P$.

Step 2: Conversion to affine coordinates (case 2 only)

If mixed addition (or mixed DBLADD) is significantly more efficient than general addition (or general DBLADD) in a given setting, then it could be convenient to express the precomputed table in $\mathcal{A}$ coordinates.

It is known that conversion to $\mathcal{A}$ can be achieved by calculating $(X_i/Z_i^2, Y_i/Z_i^3)$, $(X_i/Z_i, Y_i/Z_i^2)$ and $(Z_i/X_i, Z_i/Y_i)$ for points in $\mathcal{J}$, $\mathcal{JQ}^e$ and $\mathcal{IE}$ coordinates, respectively. For each setting, calculation of denominators (denoted by u_i) can be efficiently carried out by using

Algorithm 3.1. Computation of precomputed points using the LG Scheme

Input: a point P in affine coordinates, and
an odd value $m \neq 5$ to build a table of the form $d_i P$, where $d_i \in \{3,5,7,\ldots,m\}$
Output: the precomputed table $T = \{T_1 = 3P,\ldots,T_{(m-1)/2} = mP\}$ in $\mathcal{P}$ or $\mathcal{A}$ coordinates

1: $r = 3,\ l = 1,\ i = 2,\ n = v = 0$
2: $T_0 = P,\ T_1 = rP$
3: $R = T_1$
4: While $n < (m-3)/2$
5: If $m < 2r$
6: While $n < (m-3)/2$
7: $T_s = R + T_l$
8: $n = n + 1,\ l = l + 1,\ s = s + 1$
9: Else
10: $t = 2^v$
11: $R = 2R,\ v = v + 1,\ r = 2r,\ j = t - 1,\ first = 1$
12: While $j \geq 0$ do
13: $T_i = R - T_j,\ n = n + 1$
14: If $first = 1$, then $l = j + 1,\ s = r - i,\ first = 0$
15: $i = i + 1$
16: If $m \geq r + 2j + 1$, then
17: $T_{(r+2j)/2} = R + T_j,\ n = n + 1$
18: If $T_j = T_0$ then $i = i + 1$
19: $j = j - 1$
20: Return $T = \{T_1,\ldots,T_{(m-1)/2}\}$

the Montgomery's method of simultaneous inversion. In this way, the number of expensive inversions can be limited to only *one*.

First, we compute the inverse $U = (u_1 u_2 \ldots u_t)^{-1}$, where u_i are all distinct denominators of the conversion expressions above (without considering exponents) from all the non-trivial points in the table $\{3P, 5P, \ldots, mP\}$. For $\mathcal{J}$ and $\mathcal{JQ}^e$, the number of such denominators reduces to only $t = (m-1)/2 - c$, where c is the number of points computed via conjugate addition, since points computed with addition/conjugate addition pairs share the same Z coordinate (see Appendix A3). For $\mathcal{IE}$, $t = m - 1$ as each point has *two* distinct denominators, namely X_i and Y_i.

Then, individual denominators u_i are recovered from U and scaled with the corresponding exponent (if any), and the results are finally multiplied to their corresponding numerator following the conversion expressions.

Thus, the use of conjugate additions reduces the cost of the Montgomery's method for $\mathcal{J}$ and $\mathcal{JQ}^e$. Following the details above, it can be verified that one saves $(4M+1S)$ and $(3M+1S)$ per point computed with a conjugate addition using $\mathcal{J}$ and $\mathcal{JQ}^e$ coordinates, respectively.

3.3.2.1. Cost Analysis

The "generic" costs of the proposed scheme, cases 1-3 and case 2, are given by:

$$\text{Cost}_{\text{LG Scheme, cases 1/3}} = 1\text{TPL} + (\omega-2)\text{DBL} + (2\varepsilon - L + 1)\text{ADD} + (L-\varepsilon-1)\text{ADD-ADD}', \quad (3.9)$$

$$\text{Cost}_{\text{LG Scheme, case 2}} = 1\text{TPL} + (\omega-2)\text{DBL} + (2\varepsilon - L + 1)\text{ADD} + (L-\varepsilon-1)\text{ADD-ADD}' + \text{Cost}_{\mathcal{P}\to\mathcal{A}}, \quad (3.10)$$

respectively, where $m > 5$, $L = (m-1)/2$, $r_{\max} = 3\times 2^{\omega-2}$ is the value of the highest "strategic" point, $\varepsilon = \lfloor (6L + 2r_{\max} - 3)/(6r_{\max} - 3) \rfloor \left(L + 1 - 2r_{\max}/3\right) + (r_{\max}/3) - 1$ is the total number of regular additions and $\text{Cost}_{\mathcal{P}\to\mathcal{A}}$ denotes the cost of converting points from projective to affine coordinates and is defined by the following formulas for $\mathcal{J}$, $\mathcal{JQ}^e$ and $\mathcal{IE}$ coordinates:

$$\text{Cost}_{\mathcal{J}\to\mathcal{A}} = 1I + (6L - 4c - 3)M + (L-c)S, \quad (3.11)$$

$$\text{Cost}_{\mathcal{JQ}^e\to\mathcal{A}} = 1I + (5L - 3c - 3)M + (2L-c)S, \quad (3.12)$$

$$\text{Cost}_{\mathcal{IE}\to\mathcal{A}} = 1I + (6L + \lceil (L-2)/L \rceil - 1)M, \quad (3.13)$$

respectively, where $c = L - \varepsilon - 1$ represents the number of conjugate additions. Formulas (3.9) and (3.10) can be refined further for cases 1 and 2 with the use of mixed coordinates (case 2 additionally includes the cost of conversion to affine, i.e., $\text{Cost}_{\mathcal{P}\to\mathcal{A}}$):

$$\text{Cost}_{\text{LG Scheme, case 1(2)}} = 1\text{mTPL} + (\omega-2)\text{DBL} + (\omega-2)\text{mADD-mADD}' + \ldots$$
$$\ldots\left(L-\varepsilon-\omega+1\right)\text{ADD-ADD}' + (2\varepsilon - L + 1)\text{ADD}\ \ (+\text{Cost}_{\mathcal{P}\to\mathcal{A}}). \quad (3.14)$$

Please, refer to Appendix A5 for the proof. We remark that cost formula (3.14) is generalized to any projective system. Hence, depending on the curve form selected, some additional speed-ups are available. Let us discuss some of these optimizations in the context of $\mathcal{J}$ coordinates. First, when performing additions with a "strategic" point Q, the values Z_Q^2 and Z_Q^3 are calculated in the first mixed addition, say $Q+P = (X_Q : Y_Q : Z_Q) + (x_P, y_P)$. Then, following general additions of the form $Q+R = (X_Q : Y_Q : Z_Q) + (X_R : Y_R : Z_R)$ can be executed using $\text{ADD}_{[1,1]}$ in case 1 and save $(1M+1S)$ per operation. This can be optimized further by using $\text{ADD}_{[2,2]}$ instead and save $(2M+2S)$ per general addition if one assumes that the evaluation stage

employs additions with stored values and all values Z_i^2 and Z_i^3 need to be precomputed in case 2. Also, one squaring can be saved every time a doubling $2P_j$ is performed to get a "strategic" point since the value Z_j^2 can be obtained from the initial tripling or the mixed addition preceding this doubling. Moreover, as observed before addition and conjugate addition formulas share the same Z coordinate. Hence, in case 2 we only require $(1M + 1S)$ to get Z_i^2 and Z_i^3 for two points computed with an addition/conjugate addition pair. Similar savings apply to conversion to affine for case 1, where one saves $(4M + 1S)$ per conjugate addition as discussed in the previous section. By applying these optimizations to (3.14) with (3.11), we obtain the following cost formulas for the LG Scheme, cases 1 and 2, using Jacobian coordinates:

$$\text{Cost}_{\text{LG Scheme},\mathcal{J},\text{case 1}} = (9\varepsilon + L + \omega + 2)M + (3\varepsilon + L + 5\omega - 4)S, \tag{3.15}$$

$$\text{Cost}_{\text{LG Scheme},\mathcal{J},\text{case 2}} = 1I + (13\varepsilon + 3L + 5)M + (4\varepsilon + L + 4\omega - 1)S. \tag{3.16}$$

Note that it is still possible to optimize further cost (3.16) for case 2 if every addition with $3P$ is computed with $\text{ADD}_{[1,1]}$ by reusing values Z_{3P}^2 and Z_{3P}^3 computed in the tripling operation. This saves an extra $(1M + 1S)$ per addition with $3P$.

The following optimizations to cost formula (3.14) using $\mathcal{JQ}^e$ coordinates are analogous to the ones described for $\mathcal{J}$ coordinates. First, one squaring can be saved every time a doubling $2P_j$ is performed to get a "strategic" point by noting that $(X_j + Z_j)^2$ can be obtained from the initial tripling or the mixed addition preceding this doubling. Also, when performing additions with a "strategic" point Q, the value $(X_Q + Z_Q)^2$ is calculated in the first mixed addition. Then, each extra addition with the same point Q can be executed using $\text{ADD}_{[0,1]}$ in case 1 and save $1S$ per operation. This can be optimized further by using $\text{ADD}_{[0,2]}$ instead and save $2S$ per general addition if one assumes that the evaluation stage employs additions with stored values and all values $(X_i + Z_i)^2$ need to be precomputed in case 2. Thus, the optimized costs of the LG Scheme, case 1 and case 2, using extended Jacobi quartic coordinates are given by:

$$\text{Cost}_{\text{LG Scheme},\mathcal{JQ}^e,\text{case 1}} = (5\varepsilon + 2L + \omega + 1)M + (\varepsilon + 2L + 5\omega - 5)S, \tag{3.17}$$

$$\text{Cost}_{\text{LG Scheme},\mathcal{JQ}^e,\text{case 2}} = 1I + (8\varepsilon + 4L + \omega + 1)M + (3\varepsilon + 2L + 4\omega - 1)S. \tag{3.18}$$

Again, it is still possible to optimize further cost (3.18) for case 2 if every addition with $3P$ is computed with $\text{ADD}_{[0,1]}$ by reusing the value $(X_{3P} + Z_{3P})^2$ computed in the tripling operation. This saves an extra squaring per addition with $3P$.

In Table 3.3 we list the cost of the LG Scheme for various values L using the derived formulas (3.15), (3.16), (3.17), (3.18). Costs for $\mathcal{IE}$ coordinates can be obtained by simply applying operations from Tables 2.4 and 3.2 to cost formulas (3.14) and (3.13). As operations in

affine are relatively expensive in extended Jacobi quartic and Twisted Edwards curves (see Table 3.2), we only show the performance of case 3 in the setting of standard curves estimated with formula (3.9). In Sections 3.4.1 and 3.4.2, we carry out an exhaustive evaluation of this method's performance.

Table 3.3. Costs of the LG precomputation scheme: case 1 in projective coordinates using $\mathcal{J}$, $\mathcal{JQ}^e$ and $\mathcal{IE}$; case 2 using one inversion; and case 3 in $\mathcal{A}$.

L	Case 1			Case 2			Case 3
	$\mathcal{J}$	$\mathcal{JQ}^e$	$\mathcal{IE}$	$\mathcal{J}$	$\mathcal{JQ}^e$	$\mathcal{IE}$	Standard curve
3	$17M + 17S$	$15M + 17S$	$22M + 8S$	$1I + 27M + 18S$	$1I + 24M + 20S$	$1I + 40M + 8S$	$3I + 13M + 8S$
7	$40M + 32S$	$34M + 32S$	$51M + 14S$	$1I + 64M + 33S$	$1I + 57M + 37S$	$1I + 93M + 14S$	$6I + 23M + 14S$
15	$85M + 57S$	$71M + 57S$	$108M + 22S$	$1I + 139M + 60S$	$1I + 122M + 68S$	$1I + 198M + 22S$	$11I + 41M + 24S$

3.3.3. Precomputation Scheme for Table of the Form $c_iP \pm d_iQ$

This scenario mainly applies to methods for computing multiple scalar multiplication such as those based on JSF [Sol01, OKN10, SEI10]. In this case, the application of our strategy of conjugate additions is straightforward since precomputed points have the form $c_iP \pm d_iQ$ and each point pair $c_iP + d_iQ$ and $c_iP - d_iQ$ with $c_i, d_i \neq 0$ can be computed with an addition/ conjugate addition pair. Points c_iP and d_iQ are computed using the chain $P \rightarrow P+2P = 3P \rightarrow 3P+2P = 5P \rightarrow \ldots \rightarrow (m-2)P+2P = mP$. Interestingly enough, we note that, for the case of Jacobian coordinates with $m \geq 5$, this chain can be performed using the LM Scheme and, thus, reduce the costs further.

In the following, we analyze the cost involved when precomputing points for the window-based JSF [OKN10, SEI10]. Extension of the method to similar table forms easily follows.

3.3.3.1. Cost Analysis

First, a precomputed table $c_iP \pm d_iQ$, where $c_i, d_i \in D^+ = \{0,1,3,5,\ldots,m\}$, $c_i > 1$ if $d_i = 0$, $d_i > 1$ if $c_i = 0$ and m odd, consists of $L = (m^2 + 4m - 1)/2$ non-trivial points. For example, assuming that both P and Q are unknown before execution, if $m = 3$ one needs to precompute *ten* points: $3P$, $3Q$, $P \pm Q$, $3P \pm Q$, $P \pm 3Q$ and $3P \pm 3Q$. Recall that the negative of these points can be computed on-the-fly at negligible cost and, hence, are not included in the table.

Then, the "generic" cost of the LG Scheme for this table form, cases 1, 2 and 3, is given by:

$$\text{Cost}_{\text{LG Scheme, cases 1/3(2)}} = (m-1)\text{ADD} + \frac{(m+1)^2}{4}(\text{ADD}-\text{ADD}') + 2\left\lceil \frac{m-1}{m} \right\rceil \text{DBL} \ (+\text{Cost}_{\mathcal{P}\to\mathcal{A}}), \tag{3.19}$$

where $L = (m^2 + 4m - 1)/2 > 1$ and again $\text{Cost}_{\mathcal{P}\to\mathcal{A}}$ (that only applies to case 2) denotes the cost of converting points from projective to affine coordinates and is defined by cost formulas (3.11), (3.12) and (3.13) for $\mathcal{J}$, $\mathcal{JQ}^e$ and $\mathcal{IE}$, respectively. For these formulas, $c = (m+1)^2/4$. Cost (3.19) assumes that points $c_i P \pm d_i Q$ for which c_i or $d_i = 0$ are computed using the chain $P \to P+2P = 3P \to 3P+2P = 5P \to \ldots \to (m-2)P+2P = mP$. As mentioned before, one can apply the LM Scheme to this computation when using $\mathcal{J}$ coordinates. The cost of this combined LG/LM Scheme is given by $(m \geq 5)$:

$$\text{Cost}_{\text{LG Scheme, }\mathcal{J}\text{,cases 1(2)}} = 2\text{DBL} + (m-1)\text{ADD}_{\text{Co-}Z} + \frac{(m+1)^2}{4}(\text{ADD}-\text{ADD}') \ (+\text{Cost2}_{\mathcal{J}\to\mathcal{A}}), \tag{3.20}$$

where $\text{Cost2}_{\mathcal{J}\to\mathcal{A}} = [2m(m+4)-1]M + [(m+1)^2/4 + 2]S$ applies to case 2 only and represents the cost of converting points from Jacobian to affine coordinates using a modified Montgomery' simultaneous inversion method that has been adapted to case 2b of LM Scheme and the use of conjugate additions. Please, refer to Appendix A6 for the proof and extended details.

We remark that further optimizations are possible, such as the use of mixed coordinates or efficient tripling formulas. Similarly, certain coordinate systems such as $\mathcal{J}$ and $\mathcal{JQ}^e$ allow again the use of efficient addition formulas with stored values, following the same optimizations described in Section 3.3.2.1.

In Table 3.4, we show the cost performance of the proposed scheme for the curve forms under analysis and considering the discussed optimizations. As operations in affine are relatively expensive in $\mathcal{JQ}^e$ and $\mathcal{IE}$ coordinates, we only show the performance of case 3 in the setting of standard curves. We carry out the evaluation of this method's performance in Section 3.4.3.

Table 3.4. Cost of the LG precomputation scheme for tables of the form $c_i P \pm d_i Q$: case 1 in projective coordinates; case 2 using one inversion; and case 3 in affine coordinates.

L	Case 1			Case 2			Case 3
	$\mathcal{J}$	$\mathcal{JQ}^e$	$\mathcal{IE}$	$\mathcal{J}$	$\mathcal{JQ}^e$	$\mathcal{IE}$	Standard curve
2	$6M + 4S$	$6M + 8S$	$11M$	$1I + 10M + 4S$	$1I + 10M + 7S$	$1I + 22M$	$1I + 4M + 2S$
10	$42M + 32S$	$41M + 35S$	$65M + 9S$	$1I + 80M + 35S$	$1I + 76M + 43S$	$1I + 125M + 9S$	$6I + 30M + 16S$
22	$107M + 65S$	$100M + 74S$	$159M + 18S$	$1I + 175M + 68S$	$1I + 180M + 91S$	$1I + 291M + 18S$	$15I + 48M + 26S$

3.4. Performance Comparison

3.4.1. Evaluation of LM and LG Schemes on Standard Curves

There are different schemes to compute precomputed points on standard curves in the literature (see Section 3.1.1). The simplest approaches suggest performing computations in $\mathcal{A}$ or $\mathcal{C}$ coordinates using the chain $P \rightarrow 3P \rightarrow 5P \rightarrow \ldots \rightarrow mP$. The latter requires one doubling and $L=(m-1)/2$ additions, which can be expressed as follows in terms of field operations:

$$\text{Cost}_{\mathcal{A}} = (L+1)I + (2L+2)M + (L+2)S, \tag{3.21}$$

$$\text{Cost}_{\mathcal{C}} = (10L-1)M + (4L+5)S. \tag{3.22}$$

Note that (3.22) shows a better performance than the estimated cost given by [DOS07] since we are considering that the initial doubling $2P$ is computed as $2\mathcal{A} \rightarrow \mathcal{C}$ with a cost of $2M + 5S$, the first addition $P + 2P$ computed with a mixed addition as $\mathcal{A} + \mathcal{C} \rightarrow \mathcal{C}$ $(7M + 4S)$, and the following $(L-1)$ additions as $\mathcal{C} + \mathcal{C} \rightarrow \mathcal{C}$ $(10M + 4S)$. The new operation costs are obtained by applying the technique of replacing multiplications by squarings [LM08]. The memory requirements of the $\mathcal{A}$- and $\mathcal{C}$-based methods are $(2L+R)$ and $(5L+R)$ registers, respectively, where R is again the memory requirement of the most memory-demanding point operation used for scalar multiplication.

Let us first compare the performance of the proposed methods with approaches using several inversions (case 3). In this case, we show in Table 3.5 the performance comparison of the LG Scheme, case 3, with the traditional $\mathcal{A}$-based approach whose cost is given by (3.21). Also, the *I/M* ratios for which the traditional, LG and LM methods achieve the lowest cost are shown at the

Table 3.5. Costs of different schemes using multiple inversions (case 3) and *I/M* ranges for which each scheme achieves the lowest cost on a standard curve form ($1M = 0.8S$).

# Points (L)	2	3 ($w = 4$)	6	7 ($w = 5$)	14	15 ($w = 6$)
LG Scheme (case 3)	$3I + 12.8M$	$3I + 19.4M$	$6I + 31.4M$	$6I + 34.2M$	$11I + 57.4M$	$11I + 60.2M$
Traditional (3.21)	$3I + 9.2M$	$4I + 12M$	$7I + 20.4M$	$8I + 23.2M$	$15I + 42.8M$	$16I + 45.6M$
I/M range (LM, case 2b)	$I > 8.4M$	$I > 8.6M$	$I > 8M$	$I > 9M$	$I > 9.6M$	$I > 10.4M$
I/M range (LG, case 3)	-	$7.4M < I < 8.6M$	-	$5.5M < I < 9M$	$3.7M < I < 9.6M$	$2.9M < I < 10.4M$
I/M range (traditional)	$I < 8.4M$	$I < 7.4M$	$I < 8M$	$I < 5.5M$	$I < 3.7M$	$I < 2.9M$

bottom of the table. Note that we are including in the comparison LM Scheme, case 2b, to determine the efficiency gained by using an approach based on only one inversion (case 2).

An important result from Table 3.5 is that the LM Scheme, case 2b, outperforms approaches using several inversions for a wide range of *I/M* ratios. In general, this method is superior always that inversion is more than 8-10 times the cost of multiplication, which holds on the majority of implementations over prime fields. On the other hand, the LG Scheme, case 3, is only suitable for low/intermediate values *I/M*.

Now, let us evaluate methods for case 1, and consider the $\mathcal{C}$-based approach, whose cost is given by (3.22), for our comparisons. In this case, we should also consider the cost of scalar multiplication as the evaluation stage in $\mathcal{C}$ coordinates has a cost different to our methods.

When precomputations are in $\mathcal{C}$, Cohen et al. [CMO98] proposed the use of $\mathcal{J} + \mathcal{C} \rightarrow \mathcal{J}^m$ to perform additions ($10M + 6S$), $2\mathcal{J}^m \rightarrow \mathcal{J}$ ($2M + 5S$) to every doubling preceding an addition, and $2\mathcal{J}^m \rightarrow \mathcal{J}^m$ ($3M + 5S$) to the rest of doublings. Again, we have reduced the cost of these operations by applying the technique discussed in [LM08] to trade multiplications for squarings. Using this scheme the scalar multiplication cost including precomputations (3.22) is as follows:

$$\left[n \cdot \delta_{\text{Frac-}w\text{NAF}}\left(12M + 11S\right) + n\left(1 - \delta_{\text{Frac-}w\text{NAF}}\right)\left(3M + 5S\right)\right] + \left[\left(10L - 1\right)M + \left(4L + 5\right)S\right]. \qquad (3.23)$$

In the case of LG and LM Schemes, case 1, we consider the use of addition with stored values. Thus, the approximated cost of scalar multiplication is given by:

$$\left[n \cdot \text{DBL} + \left(\frac{(n-1)\,\delta_{\text{Frac-}w\text{NAF}}}{(L+1)}\right)\text{mADD} + \left(\frac{L(n-1)\,\delta_{\text{Frac-}w\text{NAF}}}{(L+1)}\right)\text{ADD}_{[1,1]}\right] + \text{Cost}_{\text{scheme, case 1}}, \qquad (3.24)$$

where $1\text{DBL} = 3M + 5S$, $1\text{mADD} = 7M + 4S$, $1\text{ADD}_{[1,1]} = 10M + 4S$ (as in Table 2.2) and $\text{Cost}_{\text{scheme, case 1}}$ is given by (3.4) or (3.15) for LM and LG Schemes, respectively.

Tables 3.6, 3.7 and 3.8 show the costs of performing an *n*-bit scalar multiplication using the different methods above (case 1) for n = 160, 256 and 512 bits, respectively. We show results for

Table 3.6. Performance comparison of LG and LM Schemes with the $\mathcal{C}$-based method (case 1) in 160-bit scalar multiplication on a standard curve form ($1M = 0.8S$).

# Points (L)	2	3 ($w = 4$)	4	5	6	7 ($w = 5$)
LM Scheme, case 1	1573*M*	1546*M*	1540*M*	1534*M*	1529*M*	1524*M*
LG Scheme, case 1	1577*M*	1547*M*	1545*M*	1544*M*	1537*M*	1526*M*
$\mathcal{C}$-based [CMO98]	1640*M*	1604*M*	1596*M*	1591*M*	1586*M*	1583*M*

Table 3.7. Performance comparison of LG and LM Schemes with the $\mathcal{C}$-based method (case 1) in 256-bit scalar multiplication on a standard curve form ($1M = 0.8S$).

# Points (L)	2	3 ($w = 4$)	4	5	6	7 ($w = 5$)	8
LM Scheme, case 1	2505M	2457M	2443M	2428M	2414M	2401M	2400M
LG Scheme, case 1	2509M	2458M	2448M	2438M	2422M	2403M	2407M
$\mathcal{C}$-based [CMO98]	2607M	2541M	2521M	2503M	2489M	2476M	2477M

# Points (L)	9	10	11	15 ($w = 6$)
LM Scheme, case 1	2399M	2398M	2397M	2397M
LG Scheme, case 1	2410M	2414M	2418M	2397M
$\mathcal{C}$-based [CMO98]	2479M	2481M	2484M	2498M

Table 3.8. Performance comparison of LG and LM Schemes with the $\mathcal{C}$-based method (case 1) in 512-bit scalar multiplication on a standard curve form ($1M = 0.8S$).

# Points (L)	2	3 ($w = 4$)	4	5	6	7 ($w = 5$)	8
LM Scheme, case 1	4991M	4887M	4849M	4811M	4774M	4740M	4730M
LG Scheme, case 1	4995M	4887M	4854M	4821M	4783M	4742M	4736M
$\mathcal{C}$-based [CMO98]	5184M	5040M	4986M	4938M	4895M	4857M	4846M

# Points (L)	9	10	11	12	13	14	15 ($w = 6$)
LM Scheme, case 1	4719M	4709M	4700M	4690M	4681M	4673M	4665M
LG Scheme, case 1	4731M	4725M	4721M	4710M	4694M	4679M	4665M
$\mathcal{C}$-based [CMO98]	4836M	4827M	4819M	4812M	4805M	4800M	4794M

all the possible and practical values L. Also, note that all the methods considered exhibit the same memory requirement, namely, $5L + R$.

As we can see above, the LM method, case 1, achieves the highest performance in all the cases for any number of precomputed points, surpassing the $\mathcal{C}$-based approach by up to 4.1%.

Also, it is important to note that LG Scheme's performance is comparable (or equivalent) to that of LM Scheme in several cases. The latter especially holds for standard window values w (L = 3, 7, 15).

Let us now compare methods using one inversion only (case 2). Previous methods in this scenario perform computations in $\mathcal{H}$, $\mathcal{J}$ or $\mathcal{C}$ coordinates and then convert the points to $\mathcal{A}$ by using Montgomery' simultaneous inversion method to limit the number of inversions to one. Costs of these methods are extracted from [DOS07] (assuming that $1S = 0.8M$):

$$\text{Cost}_{\mathcal{H}\to\mathcal{A}} = 1I + (16L-3)M + (3L+5)S = 1I + (18.4L+1)M \,, \tag{3.25}$$

$$\text{Cost}_{\mathcal{J}\to\mathcal{A}} = 1I + (16L-5)M + (5L+5)S = 1I + (20L-1)M \,, \tag{3.26}$$

$$\text{Cost}_{\mathcal{C}\to\mathcal{A}} = 1I + (16L-4)M + (5L-5)S = 1I + (20L)M \,. \tag{3.27}$$

Recently, Dahmen et al. [DOS07] proposed a new scheme, known as DOS, whose computations are efficiently performed using formulae in affine solely. This scheme has a low memory requirement given by $(2L+4)$ registers and computing cost:

$$\text{Cost}_{\text{DOS}} = 1I + (10L-1)M + (4L+4)S = 1I + (13.2L+2.2)M \,, \tag{3.28}$$

that shows its superiority when compared to methods (3.25), (3.26), (3.27) requiring only one inversion. However, the proposed LM Scheme achieves even lower computing costs given by $\text{Cost}_{\text{LM, case 2a}} = 1I + (11.4L+4)M$ and $\text{Cost}_{\text{LM, case 2b}} = 1I + (10.6L+4.8)M$ (assuming that $1S = 0.8M$ in formulas (3.5) and (3.6)). Therefore, LM Scheme (specifically, case 2b) achieves the lowest cost in the literature when the number of inversions is limited to one. LM Scheme, case 2a, also achieves high performance with the advantage of requiring less memory.

The previous comparison applies to scenarios where memory is not limited. For applications with strict memory constraints, it would be more realistic to compare methods for a certain number of available registers. In Table 3.9, the cost of each method is restricted by the maximum number of registers available for the evaluation stage. For each method, we show the total cost of performing a 160-bit scalar multiplication and the optimal number of precomputed points L when considering that a maximum of $(2L_{ES} + R)$ registers are available for the evaluation stage (i.e., $L \le L_{ES}$). For our analysis, we set R = 7. Also, to compare the performance of schemes using no inversions (case 1) with methods using one inversion (case 2), we include costs of the most efficient scheme found for case 1 (i.e., LM Scheme, case 1; see Tables 3.6, 3.7 and 3.8) and show at the bottom of each table the I/M range for which LM Scheme, case 1, would achieve the lowest cost. For comparisons for n = 256, 512, please refer to Appendix A7.

Table 3.9. Performance comparison of LG and LM Schemes with the DOS method in 160-bit scalar multiplication for different memory constraints on a standard curve ($1M = 0.8S$).

# Registers (L_{ES})	11 (2)		13 (3)		15 (4)		17 (5)		19 (6)	
Method	L	Cost	L	Cost	L	Cost	L	Cost	L	Cost
LM, case 2b	2	**$1I + 1506M$**	3	**$1I + 1481M$**	3	$1I + 1481M$	4	**$1I + 1476M$**	4	**$1I + 1476M$**
LM, case 2a	2	$1I + 1507M$	3	$1I + 1483M$	4	**$1I + 1479M$**	4	$1I + 1479M$	5	**$1I + 1476M$**
LG, case 2	2	$1I + 1511M$	3	$1I + 1486M$	4	$1I + 1489M$	5	$1I + 1494M$	6	$1I + 1489M$
DOS [DOS07]	2	$1I + 1509M$	3	$1I + 1486M$	4	$1I + 1484M$	5	$1I + 1483M$	5	$1I + 1483M$
LM, case 1	1	$1596M$	1	$1596M$	1	$1596M$	2	$1573M$	2	$1573M$
I/M range (LM, case1)	$I > 90M$		$I > 115M$		$I > 117M$		$I > 97M$		$I > 97M$	

# Registers (L_{ES})	23 (8)		27 (10)		29 (11)		39 (16)		≥ 41 (17)	
Method	L	Cost	L	Cost	L	Cost	L	Cost	L	Cost
LM, case 2b	5	**$1I + 1473M$**	6	**$1I + 1470M$**	7	**$1I + 1469M$**	7	**$1I + 1469M$**	7	**$1I + 1469M$**
LM, case 2a	6	$1I + 1474M$	6	$1I + 1474M$	6	$1I + 1474M$	6	$1I + 1474M$	6	$1I + 1474M$
LG, case 2	7	$1I + 1481M$	7	$1I + 1481M$	7	$1I + 1481M$	7	$1I + 1481M$	7	$1I + 1481M$
DOS [DOS07]	5	$1I + 1483M$	5	$1I + 1483M$	5	$1I + 1483M$	5	$1I + 1483M$	5	$1I + 1483M$
LM, case 1	3	$1546M$	4	$1540M$	4	$1540M$	6	$1529M$	7	$1524M$
I/M range (LM, case1)	$I > 73M$		$I > 70M$		$I > 71M$		$I > 60M$		$I > 55M$	

From results in Tables 3.9, A.1, and A.2 that target case 2, it can be seen that LM Scheme achieves the lowest cost for most cases for different security levels (lowest cost per register allowance is shown in **bold**). For n = 160 bits, the LM Scheme, case 2b, offers the lowest costs excepting for $L_{ES} = 4$, in which case LM Scheme, case 2a, is slightly cheaper. For n = 256 bits, LM Scheme, cases 2a and 2b, again achieves the lowest cost for all cases, excepting for $L_{ES} = 5$, for which the DOS method offers a slight advantage. In the case of n = 512 bits, the DOS method finds its best performance by achieving the lowest cost for $L_{ES} = 5,\ 6$ and 8. Also, the LG Scheme, case 2, results more advantageous for $L_{ES} = 7$ and 8. Nevertheless, for most cases the LM Scheme still achieves the highest performance. Also, in settings where memory is not constrained the highest speed-up is achieved with LM Scheme, case 2b, for any value n.

Finally, when comparing methods for case 1 and case 2, it can be observed that LM Scheme, case 1, can be advantageous for n = 160 bits if the ratio I/M is at least 50-60 and there are a high number of registers available. For n = 256 bits, that margin reduces to ratios greater than 90-100. And for n = 512 bits, the LM Scheme, case 1, would be the most efficient method for extremely

high ratios, which seem unrealistic in practical scenarios.

3.4.2. Evaluation of LG Scheme for Extended Jacobi Quartic and Inverted Edwards Coordinates

In this section, we analyze and compare the performance of the proposed LG Scheme (Section 3.3) with extended Jacobi quartics and inverted Edwards coordinates. As we could not find any literature related to precomputation schemes on these settings, we have derived the cost formulas of precomputing points using the traditional chain $P \rightarrow 3P \rightarrow 5P \rightarrow \ldots \rightarrow mP$. For the case without inversions (case 1), the cost of precomputation is given by $(1S = 0.8M)$:

$$\text{Cost}_{\mathcal{IE},\,\text{case}\,1} = (9L+2)M + (1L+3)S = (9.8L+4.4)M\,, \tag{3.29}$$

$$\text{Cost}_{\mathcal{JQ}^e,\,\text{case}\,1} = (7L-1)M + (3L+8)S = (9.4L+5.4)M\,, \tag{3.30}$$

for $\mathcal{IE}$ and $\mathcal{JQ}^e$ coordinates, respectively. These costs have been derived by adding the costs of performing one mixed doubling, one mixed addition and $(L-1)$ general additions. For $\mathcal{JQ}^e$ we consider the use of $\text{ADD}_{[0,1]}$ to reduce costs during the evaluation stage. For case 2, the costs are given by $(1S = 0.8M)$:

$$\text{Cost}_{\mathcal{IE},\,\text{case}\,2} = 1I + (15.8L + \lceil (L-2)/L \rceil + 3.4)M\,, \tag{3.31}$$

$$\text{Cost}_{\mathcal{JQ}^e,\,\text{case}\,2} = 1I + (12L-4)M + (5L+7)S = 1I + (16L+1.6)M\,, \tag{3.32}$$

which have been derived by adding the cost of eq. (3.13) and (3.12) with $c = 0$ (for Montgomery' simultaneous inversion method) to eq. (3.29) and (3.30), respectively.

In Table 3.10, we compare the costs of these schemes with the LG Scheme for different standard windows w. Costs for LG Scheme are calculated with formulas (3.14), (3.17), (3.18). As can be seen, the LG Scheme outperforms the methods using traditional chains in all covered cases for both $\mathcal{IE}$ and $\mathcal{JQ}^e$ coordinates. Note also that the advantage increases with the window size. For instance, if $1I = 30M$, $w = 6$, $\mathcal{JQ}^e$, the cost reduction is as high as 20% and 24% in cases 1 and 2, respectively.

Let us now compare the performance of cases 1 and 2 of LG Scheme with the objective of determining the best method for each possible scenario. In this analysis we should also consider the scalar multiplication cost since different point operation costs apply to different cases. We consider the fractional width-w NAF method for our analysis. For case 1, the approximated cost of

Table 3.10. Performance comparison of LG Scheme with methods using a traditional chain for cases 1 and 2 on $\mathcal{JQ}^e$ and $\mathcal{IE}$ coordinates ($1M = 0.8S$).

Case	Method	Curve form	$L = 1$ ($w = 3$)	$L = 3$ ($w = 4$)	$L = 7$ ($w = 5$)	$L = 15$ ($w = 6$)
Case 1	LG Scheme	$\mathcal{JQ}^e$	$10.6M$	$28.6M$	$59.6M$	$116.6M$
	Scheme (3.30)	$\mathcal{JQ}^e$	$14.8M$	$33.6M$	$71.2M$	$146.4M$
	LG Scheme	$\mathcal{IE}$	$9.4M$	$28.4M$	$62.2M$	$125.6M$
	Scheme (3.29)	$\mathcal{IE}$	$14.2M$	$33.8M$	$73.0M$	$151.4M$
Case 2	LG Scheme	$\mathcal{JQ}^e$	-	$1I + 40.0M$	$1I + 86.6M$	$1I + 176.4M$
	Scheme (3.32)	$\mathcal{JQ}^e$	-	$1I + 49.6M$	$1I + 113.6M$	$1I + 241.6M$
	LG Scheme	$\mathcal{IE}$	-	$1I + 46.4M$	$1I + 104.2M$	$1I + 215.6M$
	Scheme (3.31)	$\mathcal{IE}$	-	$1I + 51.8M$	$1I + 115.0M$	$1I + 241.4M$

scalar multiplication is given by eq. (3.24), and for case 2, the cost is given by:

$$\left[n \cdot \mathrm{DBL} + \delta_{\text{Frac-}w\text{NAF}}(n-1) \cdot \mathrm{mADD}\right] + \mathrm{Cost}_{\text{scheme, case 2}}, \tag{3.33}$$

Tables 3.11 and 3.12 show the performance of scalar multiplication including the costs of the LG Scheme, cases 1 and 2. At the bottom of the table, we display the *I/M* range for which case 1 is the most efficient approach.

As can be observed from Tables 3.11 and 3.12, on $\mathcal{IE}$ and $\mathcal{JQ}^e$ coordinates LG Scheme, case 1, achieves the best performance for most common *I/M* ratios if n = 160 bits. This result differs from that for standard curves where the use of one inversion during precomputation is only efficient for high *I/M* ratios (see Table 3.9). For higher security levels (n = 512 bits), the difference between case 1 and case 2 reduces. Ultimately, the most effective approach would be determined by the particular *I/M* ratio of a given implementation. However, as the window size grows, case 1 would be again largely preferred. Therefore, for applications where memory is not scarce, LG Scheme, case 1, achieves the lowest cost in both $\mathcal{JQ}^e$ and $\mathcal{IE}$ coordinates.

3.4.3. Evaluation of LG Scheme for a Table of the Form $c_iP \pm d_iQ$

In this section, we analyze and compare the performance of LG Scheme when targeting multiple scalar multiplication methods such as JSF (Section 3.3.3). In particular, we first compare our approach with the computation using traditional additions and then we evaluate performance of cases 1 and 2 for the window-based JSF.

Table 3.11. Cost of 160-bit scalar multiplication using Frac-*w*NAF and the LG Scheme (cases 1 and 2); and *I/M* range for which case 1 achieves the lowest cost on $\mathcal{JQ}^e$ and $\mathcal{IE}$ ($1M = 0.8S$).

Method	Curve	# of Points (L)			
		2	3 ($w = 4$)	6	≥ 7 ($w = 5$)
LG Scheme, case 1	$\mathcal{JQ}^e$	$1305M$	$1280M$	$1272M$	$1265M$
LG Scheme, case 2		$1I + 1286M$	$1I + 1267M$	$1I + 1273M$	$1I + 1269M$
I/M range (case 1)		$I > 19M$	$I > 13M$	$I > 0M$	$I > 0M$
LG Scheme, case 1	$\mathcal{IE}$	$1351M$	$1324M$	$1316M$	$1311M$
LG Scheme, case 2		$1I + 1338M$	$1I + 1318M$	$1I + 1329M$	$1I + 1329M$
I/M range (case 1)		$I > 13M$	$I > 6M$	$I > 0M$	$I > 0M$

Table 3.12. Cost of 512-bit scalar multiplication using Frac-*w*NAF and the LG Scheme (cases 1 and 2); and *I/M* range for which case 1 achieves the lowest cost on $\mathcal{JQ}^e$ and $\mathcal{IE}$ ($1M = 0.8S$).

Method	Curve	# of Points (L)			
		2	3 ($w = 4$)	6	7 ($w = 5$)
LG Scheme, case 1	$\mathcal{JQ}^e$	$4126M$	$4036M$	$3951M$	$3922M$
LG Scheme, case 2		$1I + 4055M$	$1I + 3970M$	$1I + 3900M$	$1I + 3874M$
I/M range (case 1)		$I > 71M$	$I > 66M$	$I > 51M$	$I > 48M$
LG Scheme, case 1	$\mathcal{IE}$	$4273M$	$4179M$	$4090M$	$4061M$
LG Scheme, case 2		$1I + 4209M$	$1I + 4120M$	$1I + 4050M$	$1I + 4028M$
I/M range (case 1)		$I > 64M$	$I > 59M$	$I > 40M$	$I > 33M$

Method	Curve	# of Points (L)	
		14	≥ 15 ($w = 6$)
LG Scheme, case 1	$\mathcal{JQ}^e$	$3879M$	$3870M$
LG Scheme, case 2		$1I + 3867M$	$1I + 3862M$
I/M range (case 1)		$I > 12M$	$I > 8M$
LG Scheme, case 1	$\mathcal{IE}$	$4018M$	$4011M$
LG Scheme, case 2		$1I + 4033M$	$1I + 4032M$
I/M range (case 1)		$I > 0M$	$I > 0M$

The "generic" cost of precomputation using ordinary additions is given by:

$$\text{Cost}_{\text{scheme, cases 1(2)}} = \left(\frac{m(m+4)-1}{2}\right)\text{ADD} + 2\left\lceil\frac{m-1}{m}\right\rceil\text{DBL}\ \ (+\text{Cost}_{\mathcal{P}\to\mathcal{A}}), \tag{3.34}$$

where $\text{Cost}_{\mathcal{P}\to\mathcal{A}}$ applies to case 2 only and represents the cost of conversion from projective to affine coordinates given by eq. (3.11), (3.12), (3.13) with $c = 0$ for $\mathcal{J}$, $\mathcal{JQ}^e$, $\mathcal{IE}$ coordinates, resp. For $\mathcal{J}$ and $\mathcal{JQ}^e$, cost (3.34) can again be optimized further by using mixed coordinates, tripling formulas and additions with stored values. In Table 3.13, we compare the performance of this scheme with the LG Scheme. The costs for the latter method are taken from Table 3.4.

Table 3.13. Performance comparison of LG Scheme and a scheme using traditional additions for computing tables of the form $c_iP \pm d_iQ$, cases 1 and 2 ($1M = 0.8S$).

Method	Curve	# of Points (L)		
		$L = 2$ ($m = 1$)	$L = 10$ ($m = 3$)	$L = 22$ ($m = 5$)
LG Scheme, case 1	$\mathcal{J}$	$9M$	$68M$	$159M$
Scheme (3.34), case 1		$11M$	$102M$	$225M$
LG Scheme, case 2		$1I + 13M$	$1I + 108M$	$1I + 229M$
Scheme (3.34), case 2		$1I + 22M$	$1I + 154M$	$1I + 373M$
LG Scheme, case 1	$\mathcal{JQ}^e$	$12M$	$69M$	$159M$
Scheme (3.34), case 1		$16M$	$88M$	$204M$
LG Scheme, case 2		$1I + 16M$	$1I + 110M$	$1I + 253M$
Scheme (3.34), case 2		$1I + 25M$	$1I + 145M$	$1I + 331M$
LG Scheme, case 1	$\mathcal{IE}$	$11M$	$72M$	$173M$
Scheme (3.34), case 1		$14M$	$88M$	$212M$
LG Scheme, case 2		$1I + 22M$	$1I + 88M$	$1I + 305M$
Scheme (3.34), case 2		$1I + 25M$	$1I + 147M$	$1I + 343M$

As can be seen, the LG Scheme outperforms the method using traditional additions in all cases covered. For instance, if $1I = 30M$, $L = 22$, $\mathcal{JQ}^e$, the cost reduction is as high as 22% for both case 1 and 2. Remarkably, the higher improvements are obtained with $\mathcal{J}$ coordinates due to the combined use of LG and LM Schemes (see Section 3.3.3), especially in case 2, where larger savings are obtained through both methods when converting points to affine coordinates. For instance, if $1I = 30M$, $L = 22$, $\mathcal{J}$, the cost reduction is as high as 38% in case 2.

Assuming that points P and Q are unknown before execution and given in affine, a multiple scalar multiplication with the form $kP + lQ$ using windowed JSF costs approx. [$n \cdot \text{DBL} +$

$\delta_{\mathrm{JSF}}(L/(L+2))(n-1)\mathrm{ADD} + \delta_{\mathrm{JSF}}(2/(L+2))(n-1)\mathrm{mADD}] + \mathrm{Cost}_{\mathrm{scheme,\,case\,1}}$ and $[n \cdot \mathrm{DBL} + \delta_{\mathrm{JSF}}(n-1)\mathrm{mADD}] + \mathrm{Cost}_{\mathrm{scheme,\,case\,2}}$ for cases 1 and 2, respectively, where $\delta_{\mathrm{JSF}} = 0.5$ if $m = 1$, $\delta_{\mathrm{JSF}} = 0.3575$ if $m = 3$, $\delta_{\mathrm{JSF}} = 0.31$ if $m = 5$ [SEI10], and $\mathrm{Cost}_{\mathrm{scheme,\,case}\,x}$ represents the cost of precomputation given by formula (3.19). For $\mathcal{J}$ and $\mathcal{JQ}^e$, we use again $\mathrm{ADD}_{[M,S]}$ instead of ADD. The estimates using these cost formulas are displayed in Tables 3.14 and 3.15.

Table 3.14. Cost of 160-bit multiple scalar multiplication using window-based JSF and LG Scheme (cases 1 and 2); and *I/M* ranges for which case 1 achieves the lowest cost; $1M = 0.8S$.

Method	Curve	# of Points (L)		
		$L = 2$ ($m = 1$)	$L = 10$ ($m = 3$)	$L = 22$ ($m = 5$)
LG Scheme, case 1	$\mathcal{J}$	$2059M$	$1909M$	$1917M$
LG Scheme, case 2		$1I + 1944M$	$1I + 1808M$	$1I + 1851M$
I/M range (case 1)		$I > 115M$	$I > 101M$	$I > 66M$
LG Scheme, case 1	$\mathcal{JQ}^e$	$1680M$	$1554M$	$1578M$
LG Scheme, case 2		$1I + 1643M$	$1I + 1548M$	$1I + 1627M$
I/M range (case 1)		$I > 37M$	$I > 6M$	$I > 0M$
LG Scheme, case 1	$\mathcal{IE}$	$1742M$	$1612M$	$1644M$
LG Scheme, case 2		$1I + 1714M$	$1I + 1624M$	$1I + 1731M$
I/M range (case 1)		$I > 28M$	$I > 0M$	$I > 0M$

Table 3.15. Cost of 512-bit multiple scalar multiplication using window-based JSF and LG Scheme (cases 1 and 2); and *I/M* ranges for which case 1 achieves the lowest cost; $1M = 0.8S$.

Method	Curve	# of Points (L)		
		$L = 2$ ($m = 1$)	$L = 10$ ($m = 3$)	$L = 22$ ($m = 5$)
LG Scheme, case 1	$\mathcal{J}$	$6583M$	$5972M$	$5794M$
LG Scheme, case 2		$1I + 6203M$	$1I + 5555M$	$1I + 5428M$
I/M range (case 1)		$I > 380M$	$I > 417M$	$I > 366M$
LG Scheme, case 1	$\mathcal{JQ}^e$	$5358M$	$4828M$	$4707M$
LG Scheme, case 2		$1I + 5234M$	$1I + 4717M$	$1I + 4655M$
I/M range (case 1)		$I > 124M$	$I > 111M$	$I > 52M$
LG Scheme, case 1	$\mathcal{IE}$	$5562M$	$5006M$	$4887M$
LG Scheme, case 2		$1I + 5445M$	$1I + 4914M$	$1I + 4874M$
I/M range (case 1)		$I > 117M$	$I > 92M$	$I > 13M$

Similarly to the case of single scalar multiplication (see Table 3.11), case 1 achieves the best performance for most common *I/M* ratios for n = 160 bits with $\mathcal{JQ}^e$ and $\mathcal{IE}$ coordinates. However, if n = 512 bits, the range of *I/M* ratios for which case 2 is more efficient increases significantly. Also, note that case 2 appears to be the best choice for $\mathcal{J}$ coordinates for a wide range of *I/M* ratios, especially for high levels of security, i.e., n = 512.

3.5. Other Applications of Conjugate Additions

We have discussed in detail the application of the conjugate addition strategy in the design of efficient precomputation tables with the forms d_iP and $c_iP \pm d_iQ$. However, this technique can be easily applied to other table forms such as the one required by the generalized JSF [Pro03], which involves the precomputation of $(3^k - 1)/2 - k$ non-trivial points. For instance, for k = 3 scalars, the previous algorithm requires the precomputation of $P \pm Q$, $P \pm R$, $Q \pm R$, $P + Q \pm R$, $P - Q \pm R$, which costs about 10 general additions (case 1). By using conjugate additions, the latter is reduced to only 5 addition/conjugate addition pairs. Note that the advantage grows exponentially with the number of scalars. As mentioned in Section 3.1.3, Järvinen et al. [JFS07] also proposed a method to precompute points with the form $dP \pm lQ \pm kR$. However, their approach makes use of Okeya's conjugate addition in *affine coordinates* in combination with Montgomery' simultaneous inversion method. Therefore, it is limited to the Weierstrass form and always requires one inversion (i.e., it only applies to case 2). Moreover, in its current format their scheme only applies to tables $dP \pm lQ \pm kR$ where $d, l, k \in \{0,1\}$.

Other obvious application is the extension of our strategy to other settings such as binary fields. Let us illustrate the latter with the addition formula due to [LD99] and later refined by [HT00]. The cost of adding two points $P + Q$ with the formula by [HT00] takes 13*M* + 4*S*. Then, if the value $P - Q$ is required right after, one can store most partial results from the original addition and obtain the previous value with a cost of only 5*M* by noticing that $-Q = (X_2, X_2Z_2 + Y_2, Z_2)$ in Lopez-Dahab ($\mathcal{LD}$) coordinates. Note that the partial term $Y_2Z_1^2$ from the original formula is replaced by $-Y_2Z_1^2 = (X_2Z_2 + Y_2)Z_1^2 = X_2Z_2Z_1^2 + Y_2Z_1^2$, which only cost *one* extra multiplication.

We have also analyzed other relevant settings such as Twisted Edwards using $\mathcal{E}/\mathcal{E}^e$. Unfortunately, conjugate additions in this case are relatively expensive. Accordingly, we use a traditional sequence to calculate precomputations on this system in the corresponding implementations of single scalar multiplication in Chapter 5 (see §5.6.1 and §5.6.2).

In summary, generalizations of this technique and the derived precomputation schemes may be applied to other scalar multiplication methods, coordinate systems and/or elliptic curve forms, provided the corresponding conjugate formulas are efficient. For instance, Goundar, Joye and Miyaji [GMJ10] recently proposed improved formulas combining the concept of conjugate

addition with the $\text{ADD}_{\text{Co-}Z}$ operation in order to improve side-channel-protected scalar multiplication methods.

A note on related work:

After developing the LG Scheme, we became aware of other (virtually simultaneous) efforts based on similar ideas. Avanzi, Heuberger and Prodinger [AHP08] also noticed the savings introduced by computations with the form $P \pm Q$ when precomputing points in projective coordinates. They, however, analyzed the applicability of this idea in the context of Koblitz curves with τ–adic representations using $\mathcal{LD}$ coordinates. In a talk in ECC2008 [Sco08], Scott described an approach similar to the LG Scheme for the case of single scalar multiplication. He also proposed to exploit similarities between $P + Q$ and $P - Q$ during precomputation but using a slightly different sequence to compute points. After an analysis on the settings discussed in this chapter, we conclude that our calculation sequence achieves better performance.

3.6. Conclusions

This chapter introduced new schemes for precomputing points, a basic ingredient to accelerate the fastest variable-scalar-variable-point scalar multiplication methods which are based on window-based strategies.

After presenting most relevant previous work in §3.1, we introduced in §3.2 the LM Scheme, which is intended for standard curves using Jacobian coordinates, and adapted it to *two* typical scenarios for precomputation: case 1, without using inversions; and case 2, using one inversion. For the latter, we presented *two* variants that have slightly different speeds and memory requirements. The theoretical costs for each case were derived (with the corresponding proofs in the appendix), exploiting state-of-the-art formulas and techniques for maximal performance. In particular, for a number L of non-trivial points, case 1 has a cost of $(5L+1)M + (2L+5)S$ (or $(6L+1)M + (3L+5)S$ when using operations with stored values) and case 2b has a cost of $1I + (9L)M + (2L+6)S$, which are the lowest in the literature for tables $d_i P$.

In §3.3, we introduced the highly-flexible LG Scheme, which is based on the concept of conjugate addition and that can be adapted to any curve form or type of scalar multiplication (i.e., single and multiple scalar versions). We also discussed its applicability to cases 1, 2 and 3, and analyzed its efficiency on *three* curve settings: standard curves using Jacobian coordinates, extended Jacobi quartics using extended Jacobi quartics coordinates and Twisted Edwards curves using inverted Edwards coordinates. Moreover, for the case of multiple scalar multiplication using Jacobian coordinates, we proposed a novel scheme combining the LG and LM approaches. The theoretical costs for each case were derived (with the corresponding proofs in the appendix),

exploiting state-of-the-art formulas and techniques for maximal performance.

In §3.4, we carried out an extensive analysis of the proposed methods, presenting detailed comparisons with previously most efficient methods in terms of speed and memory consumption and for different security levels. We showed that for most cases the LM Scheme remains as the most efficient method on standard curves using Jacobian coordinates for the case of a table with the form d_iP (implementers may consult Tables 3.9, A.1, and A.2 for the best scheme given the number of registers and precomputed points, *I/M* ratio and security level). This result is especially relevant for implementations following NIST recommendations. On the other hand, the LG Scheme was shown to achieve the lowest costs on the special curves under study for both table forms, d_iP and $c_iP \pm d_iQ$. Also, the combined LG/LM approach for tables $c_iP \pm d_iQ$ using Jacobian coordinates was shown to provide substantial cost reductions with advantage growing with the number of precomputations. Implementers may consult Tables 3.11 and 3.12 (Tables 3.14 and 3.15) for the best variant of the LG Scheme for a table d_iP (for a table $c_iP \pm d_iQ,$) given the curve form, number of precomputations, *I/M* ratio and security level. Extensions of this work would enable the use of the LG Scheme on other curve forms and coordinate systems. This is left for future work.

Finally, in §3.5 we discussed more possibilities for the use of conjugate addition. We detailed potential applications that could be fully explored in future work and discussed recent research that has already taken advantage of this idea.

Chapter 4

Scalar Multiplication using Multibase Chains

In this chapter, we describe efficient methods based on multibase representations and analyze their performance to compute elliptic curve scalar multiplication at the evaluation stage. Our contributions can be summarized as follows:

- We include a thorough discussion and analysis of the most relevant methods based on double- and multi-base representations in the literature. We categorize the different approaches and highlight their advantages and disadvantages.
- We provide an improved and more thorough exposition of the original multibase NAF (*mb*NAF) method and its variants, which were introduced by the author in [Lon07]. In particular, we include the analysis of the average density of these methods when using bases {2,3} and {2,3,5} that was deferred in [Lon07].
- We apply the concept of "fractional" windows to improve the flexibility of the windowed variant of *mb*NAF so that implementers can freely choose the optimal number of precomputations in a given application.
- We apply the concept of *operation cost per bit* to the derivation of efficient multibase algorithms able to find cheaper multibase chains for scalar multiplication. We argue that this approach, assuming unrestricted resources, leads to *optimal* multibase chains for any

given scalar. For practical scenarios, we present very compact algorithms that yield (conjecturally, close to optimal) multibase chains.

- Finally, we perform an exhaustive performance evaluation of the various methods for different security levels and for *three* different curve forms: standard curves using Jacobian coordinates ($\mathcal{J}$), extended Jacobi quartics using extended Jacobi quartic coordinates ($\mathcal{JQ}^e$) and Twisted Edwards curves using inverted Edwards coordinates ($\mathcal{IE}$). These results allow us to assess the state of affairs of the use of double bases and multibases in practice.

For the remainder of this chapter, we assume that curve parameters can be chosen such that the cost of multiplying a curve constant can be considered negligible in comparison with a regular multiplication. Also, additions and subtractions are neglected when performing cost analysis. These assumptions greatly simplify our analysis without affecting the conclusions.

This chapter is organized as follows. §4.1 discusses the most relevant previous work and categorizes the different approaches based on double- and multi-base representations. §4.2 discusses the *mb*NAF method and its variants, and provides the zero and nonzero density formulas obtained with the use of Markov chains. §4.3 details the application of "fractional" windows to *mb*NAF. §4.4 presents the methodology based on the *operation cost per bit* to derive more efficient multibase chains. §4.5 evaluates the performance of the different methods in comparison with other works in the literature for different security levels and memory constraints. §4.6 discusses potential variants of the proposed methods and their application to other settings. This section also discusses the challenges still faced by methods using double- and multi-base representations. Finally, some conclusions are drawn in §4.7.

4.1. Previous Work

As discussed in Section 2.2.4.3, in the last few years there have appeared a plethora of works proposing efficient methods to compute scalar multiplication. In the case under study, namely, when the initial point P is not known in advance, well-known methods to efficiently execute kP are non-adjacent form (NAF) [Rei60] and width-w NAF (wNAF) [Sol00], which use short signed radix 2-based representations of the scalar to minimize the number of point operations, namely doubling of a point ($2P$) and addition of points ($P+Q$). In particular, wNAF offers very high performance at the cost of a few precomputations.

Later, Möller [Möl03] generalized wNAF to any number of precomputations using "fractional" windows. The new recoding, called fractional width-w NAF (denoted by Frac-wNAF; see Section 2.2.4.3), allows a better coupling between the scalar multiplication computation and the memory resources available in a given implementation.

4.1.1. Double- and Multi-Base Number Representations

Recently, there have been proposed new methods for scalar multiplication using number representations based on double- and multi-base number systems, which basically mix different bases to decrease the number of terms required in the representation of integers. Based on previous work by Dimitrov and Cooklev [DC95], the use of the so-called Double Base Number System (DBNS) for cryptographic applications was first proposed by Dimitrov et al. in [DJM98]. In this number system an integer k is represented as follows:

$$k = \sum_{i=1}^{K} s_i 2^{b_i} \cdot 3^{c_i} , \tag{4.1}$$

where $s_i \in \{-1,1\}$.

To enable the use of DBNS in the setting of ECC, Dimitrov et al. [DIM05] were the first to introduce the concept of *double-base chains* where b_i and c_i must decrease as i increases. This was later generalized to *multi-base chains* (i.e., using two or more bases) by the author in [Lon07] and Mishra and Dimitrov in [MD07]. Of particular interest are the facts that multibase chains are redundant and that some representations are highly sparse, which, as consequence, allow a reduction in the Hamming weight of the scalar expansion (that is, a reduction in the number of additions in the point multiplication). Let us illustrate the latter with the following example.

Example 4.1. The representation of $k = 9750$ using NAF is given by $9750 = 2^{13} + 2^{11} - 2^9 + 2^5 - 2^3 - 2$, which requires 13DBL + 5ADD using Horner's scheme for scalar multiplication (i.e., the computation uses the expansion $9750P = 2(2^2(2^2(2^4(2^2(2^2P + P) - P) + P) - P) - P)$). If one, otherwise, uses the double-base chain $9750 = 2^{10} \times 3^2 + 2^6 \times 3 - 2^4 \times 3 + 2 \times 3$, the scalar multiplication takes the form $9750P = 2 \times 3\,(2^3(2^2 \times 3(2^4P + P) - P) + P)$ and costs 10DBL + 2TPL + 3ADD, which reduces the nonzero density in comparison with the NAF representation.

Multibase chains using $\{2^{b_i}3^{c_i}\}$-terms or $\{2^{b_i}3^{c_i}5^{d_i}\}$-terms are particularly attractive for ECC because operations associated with these bases (namely, point doubling, tripling and quintupling) are the *cheapest-per-bit* point operations available for some elliptic curves.

Nevertheless, multibase chains are not unique and this poses the conjecturally hard problem of determining (in a reasonable amount of time and utilization of resources) the *optimal* multibase chain for a given integer. Hereinafter, we use the term *optimal* to define a multibase chain for a given scalar k that achieves the lowest cost when applied to the computation of the point multiplication kP. In contrast to radix 2-based representations, the complexity of this analysis is significantly higher as the point operations involved (e.g., doubling, tripling, quintupling and addition) have different costs per bit that even vary with the type of elliptic

curve. Hence, it does not necessarily hold that representations with the lowest nonzero density achieve the lowest cost. Note that this complexity increases with the number of bases in the representation.

Although it remains an open problem to find the optimal double- or multi-base chains, there have appeared in the literature several efforts trying to find "efficient" multibase chains and using them advantageously in the computation of elliptic curve scalar multiplication. In general, there are *two* main approaches to find a double-base or multi-base representation for a given integer in the setting of elliptic curves: using a "Greedy" algorithm [DIM05, DI06] and using division chains [CJL+06, Lon07] (borrowing the term from [Wal98]).

4.1.1.1. Multibase Methods based on a "Greedy" Algorithm

The "Greedy"-based approach, first proposed in [DIM05], works as follows. To find a representation with the form (4.1) first establish "efficient" maximum bounds $b_{\max}$ and $c_{\max}$ for the powers of 2 and 3, respectively. Then search for the closest $\{2^{b_i}3^{c_i}\}$-term to the scalar k, subtract it from k and search again for the closest $\{2^{b_i}3^{c_i}\}$-term to the updated value. Repeat the procedure until $k = 0$. It can be easily deduced that b_i and c_i will form decreasing sequences $b_{\max} \geq b_1 \geq b_2 \geq \ldots \geq b_K \geq 0$ and $c_{\max} \geq c_1 \geq c_2 \geq \ldots \geq c_K \geq 0$, respectively. Later, Doche and Imbert [DI06] extended the "Greedy" algorithm to applications that can afford precomputations by allowing the precomputation of a table with the form $\{2,2^2,\ldots,2^{w_1},3,3^2,\ldots,3^{w_2}\}P$, where w_1 and w_2 represent the maximum exponents expanding the search range in the "Greedy" algorithm, or a table with the form d_iP, where d_i are odd digits coprime to 3 (for instance, $d_i \in D = \{1,5,7,11,\ldots\}$). This approach was later optimized by [BBL+07] with the use of precomputed tables using the digit sets $\{1,2,3,5,7,\ldots,m\}$, with m odd. Finally, Mishra and Dimitrov [MD07] extended the "Greedy"-based approach to chains using bases $\{2,3,5\}$.

The use of a "Greedy" algorithm has several drawbacks. First, from a theoretical point of view, double-base chains found with a "Greedy" algorithm cannot (until today) be defined adequately. Hence, the expected number of zero and nonzero terms for an n-bit scalar is estimated empirically. Also, looking for closest $\{2^{b_i}3^{c_i}\}$-terms implies having a table storing many powers of 2, 3 and combinations of these. This is directly impractical in constrained environments. One can trade memory for speed and store only part of the required table. However, this leads to higher conversion times (to double-base representation), lower performance and/or very expensive precomputation stages [BPP07]. This issue obviously worsens with expanded digit sets and more bases.

4.1.1.2. Multibase Methods based on Division Chains

This approach consists in the derivation of scalar representations by consecutive division with integers from a suitably chosen set of bases. When the partial result is not divisible by at least one

base then a particular rule defines how to approximate the value to a close number that is again divisible by one or more bases. Note that methods using division chains are apparently easier to analyze by using, for instance, Markov chains. Moreover, they do not rely on pre-stored tables for conversion, immediately enabling their use in memory-constrained applications. In the 90's several algorithms with different division rules were proposed for reducing the cost of exponentiation [DC95, CCY96, Wal98] (the term "division chain" was coined by Walter in [Wal98]). Walter [Wal02] also exploited these ideas to develop an exponentiation method with random selection of bases to protect against certain SCA attacks. Nevertheless, it seems that the binary/ternary algorithm by Ciet et al. [CJL+06] was the first method using division chains that was intended for ECC applications. In this case, a partial result obtained after dividing by bases 2 and 3 is approximated to the closest term that is congruent to $0 (\mathrm{mod}\, 6)$. Since this approximation gives roughly equivalent "weight" to bases 2 and 3, the method has some efficiency limitations especially in most common ECC settings where doubling is much faster than tripling and addition. In fact, if one does not take into account the memory/conversion overhead, it can be the case that "Greedy"-based approaches achieve better performance (see, for example, Table 2 in [DH08]). In [Lon07] (see also [LM08c]), the author introduced new algorithms able to find generalized multi-base chains, solving efficiently for first time the problem of memory penalty and difficulty to analyze the zero and nonzero density of a multibase expansion. Remarkably, it also achieves better cost performance than the "Greedy" approach. The new method finds multibase chains by creating a "window" with a fixed width with one of the bases (referred to as "main base") and then approximates the partial scalar value to it. The latter guarantees the execution of a minimum number of operations with the "main base" before the following addition happens, similar to the way NAF of a scalar is generated with base 2. Moreover, the nonzero density is further reduced because, once an addition is performed, not only doublings but also triplings, quintuplings, and so on, can be used. This new approach is called multibase NAF (denoted by *mb*NAF). Its window-based version using an extended digit set appears as a natural extension and is referred to as width-*w* multibase NAF (*wmb*NAF).

NOTE: one does not need to restrict the "window" in *mb*NAF to only *one* base. In fact, in [Lon07] (see also [LM08c, Section 5.3]), the author proposed an *extended wmb*NAF method that generalizes the use of windows, such that the approximation after the divisibility tests is performed to the generic value $a = a_1^{w_1} \cdot a_2^{w_2} \cdot \ldots \cdot a_J^{w_J}$ for a set of bases $\{a_1, a_2, \ldots, a_J\}$, where $w_j \geq 0$ are integers. For instance, the use of $a = 2^{w_1} \cdot 3^{w_2}$ was shown to be especially efficient on the elliptic curves with degree 3 isogenies proposed in [DIK06] and known as DIK curves (see Table 8 in [LM08c]). Note that this method was recently rediscovered by Adikari, Dimitrov and Imbert in [ADI10, Section 3.1] and [Adi10, Chapter 5] for the case of bases $\{2,3\}$. Also, note that the binary/ternary algorithm by [CJL+06] is a special case of *extended wmb*NAF when $a = 2 \cdot 3$.

More recently, Doche et al. [DH08] introduced a new method that also finds double-base chains using division chains, although using a somewhat more complex tree-based approach in comparison with multibase NAF. Their method basically divides by 2 and 3 values (k_i+1) and (k_i-1) for B distinct values k_i that are coprime to 6, and keeps the B division sequences that reach the lowest values. This procedure is repeated with the new values until reaching 1. Initialization proceeds as above although in this case the algorithm keeps all the possible sequences until B distinct values k_i are obtained. As will be evident later, the disadvantage of this method is that the division sequences that are chosen at each iteration are the ones whose final values are the lowest ones. However, a long sequence of divisions alone does not guarantee optimal cost. This drawback is somewhat minimized by keeping up to B values at each iteration (and then the probability that a long sequence is also among the cheapest ones increases). However, it is evident that one may avoid storing unnecessary sequences by applying an operation cost analysis instead.

In Section 4.4, we introduce a methodology to derive algorithms able to find more efficient multibase chains. Our technique is based on the careful analysis of the *operation cost per bit*, which helps to choose the most efficient division sequence per iteration. We argue that the inclusion of this analysis in the design of any multibase algorithm potentially enables the derivation of the fastest multibase chains.

4.2. Multibase NAF (*mb*NAF) and Width-*w* Multibase NAF (*wmb*NAF)

Determining and finding the *optimal* multibase chain in the setting of ECC seems to be a hard problem, mainly due to the fact that an *optimal* multibase chain is not necessarily the shortest one (with the minimal number of additions), but the one that requires the "right" balance in the number of additions and all other point operations (which depends on the chosen elliptic curve form). Although finding such optimal multibase chains remains an open problem, the author [Lon07] proposed a representation that adjusts more efficiently to most ECC settings, in which one of the point operations is usually significantly more efficient than the others. Such a generic multibase representation, known as *mb*NAF, has the form:

$$k = \sum_{i=1}^{K} s_i \prod_{j=1}^{J} a_j^{c_i(j)} \tag{4.2}$$

where: $a_1 \neq \ldots \neq a_J$ are prime integers from a set of bases $\mathcal{A} = \{a_1, \ldots, a_J\}$ (a_1: main base),
K is the length of the expansion,
s_i are signed digits from a given set $D \setminus \{0\}$, i.e., $|s_i| \geq 1$ and $s_i \in D \setminus \{0\}$,
$c_i(j)$ are decreasing exponents, s.t. $c_1(j) \geq c_2(j) \geq \ldots \geq c_K(j) \geq 0$ for $2 \leq j \leq J$, and

$c_i(1)$ are decreasing exponents for the main base a_1 (i.e., $j = 1$), s.t. $c_i(1) \geq c_{i+1}(1) + 2 \geq 2$ for $1 \leq i \leq K-1$.

Note that the last two conditions above guarantee that an expansion of the form (4.2) is efficiently executed by a scalar multiplication using Horner's method as follows:

$$kP = \left(\sum_{i=1}^{K} s_i \prod_{j=1}^{J} a_j^{c_i(j)} \right) P = \prod_{j=1}^{J} a_j^{d_K(j)} \left(\prod_{j=1}^{J} a_j^{d_{K-1}(j)} \left(\ldots \left(\prod_{j=1}^{J} a_j^{d_1(j)} (s_1 P) + s_2 P \right) + \ldots + s_{K-1} P \right) + s_K P \right) \quad (4.3)$$

where $d_K(1) \geq 0$, and $d_i(1) \geq 2$ for $1 \leq i \leq K-1$. The latter is equivalent to the last condition in (4.2) and incorporates the non-adjacency property in the multibase representation. Basically, it fixes the minimal number of consecutive operations with the "main base" (i.e., a_1) between any two additions to *two*. Note that an operation with the main base refers to doubling if $a_1 = 2$ or tripling if $a_1 = 3$, and so on.

On the other hand, if we relax the previous condition and allow larger window sizes (i.e., allowing 3, 4, or more, consecutive operations with the main base between any two additions) we can reduce further the average number of nonzero terms in the scalar representation at the expense of a larger digit set D and, consequently, a larger precomputed table. The previous technique is known as *wmb*NAF.

The *mb*NAF and *wmb*NAF representations require the following digit set [Lon07]:

$$D = \left\{ 0, \pm 1, \pm 2, \ldots, \pm \left\lfloor \frac{a_1^w - 1}{2} \right\rfloor \right\} \setminus \left\{ \pm 1 a_1, \pm 2 a_1, \ldots, \pm \left\lfloor \frac{a_1^{w-1} - 1}{2} \right\rfloor a_1 \right\} \quad (4.4)$$

where $w \geq 2 \in \mathbb{Z}^+$ ($w = 2$ for *mb*NAF). Without considering $\{\mathcal{O}, P\}$, the digit set (4.4) implies that a scalar multiplication would require precomputing $d_i P$, where $d_i \in D^+ \setminus \{0,1\}$ (note that only positive values $d_i P$ need to be stored in the table as the negative of points can be computed on-the-fly at negligible cost). Thus, the precomputation table consists of $(a_1^w - a_1^{w-1} - 2)/2$ points. Note that if $w = 2$ (*mb*NAF case), the requirement of precomputations is minimal. For instance, in the case $a_1 = 2$ we do not need to store any points besides $\{\mathcal{O}, P\}$.

It can be easily seen that selecting the main base according to the relative efficiency of its corresponding operation will guarantee that more of these operations are used in average, which potentially could decrease the computational cost of scalar multiplication. In the remainder (and following what is observed in most common ECC settings over prime fields), we will assume that doubling is the most efficient point operation available, and hence, $a_1 = 2$.

It is important to remark that, obviously, eq. (4.2) does not involve unique representations. For instance, both expressions $2^{10} \times 3^2 + 2^6 \times 3^2 - 2^4 \times 3 + 2 \times 3$ and $2^9 \times 3^3 - 2^7 \times 3^3 - 2^5 \times 3^3 + 2^3 \times 3^3 + 2 \times 3^2 + 3^2 + 3$ enable two different *mb*NAF representations for the integer 9750

following (4.2). In [Lon07], the author provided algorithms based on division chains that efficiently find an (*w*)*mb*NAF chain of the form (4.2) and, given a window width and set of bases, is unique for each integer. Note that we have integrated algorithms for finding *mb*NAFs and *wmb*NAFs in Algorithm 4.1.

Algorithm 4.1. Computing an *mb*NAF (*wmb*NAF) of a positive integer

Input: scalar k, bases $\mathcal{A} = \{a_1, a_2, \ldots, a_J\}$, where $a_j \in \mathbb{Z}^+$ are primes for $1 \le j \le J$,
window $w = 2$ for *mb*NAF, and window $w > 2$ for *wmb*NAF, where $w \in \mathbb{Z}^+$
Output: $(a_1, a_2, ..., a_J)\text{NAF}_w(k) = (..., k_2^{(b_2)}, k_1^{(b_1)})$, where $b_i \in \mathcal{A}$

1: $i = 1$
2: While $k > 0$ do
3: If $k \bmod a_1 = 0$ or $k \bmod a_2 = 0$ or … or $k \bmod a_J = 0$, then $k_i = 0$
4: Else
5: $k_i = k \bmod s a_1^w$
6: $k = k - k_i$
7: If $k \bmod a_1 = 0$, then $k = k/a_1$, $k_i^{(b_i)} = k_i^{(a_1)}$
8: Elseif $k \bmod a_2 = 0$, then $k = k/a_2$, $k_i^{(b_i)} = k_i^{(a_2)}$
⋮ ⋮
J+6: Elseif $k \bmod a_J = 0$, then $k = k/a_J$, $k_i^{(b_i)} = k_i^{(a_J)}$
J+7: $i = i + 1$
J+8: Return $(..., k_2^{(b_2)}, k_1^{(b_1)})$

$k_i^{(b_i)}$ in Algorithm 4.1 represent the digits in the multibase NAF representation, where $k_i \in D$, see (4.4); and the superscript (b_i) represents the base $b_i \in \mathcal{A}$ associated to the digit in position *i*. The function *mods* represents the following computation:

$$\begin{cases} \text{If } k \bmod a_1^w \ge a_1^w/2, \text{ then } k_i = \left(k \bmod a_1^w\right) - a_1^w \\ \text{Else, } k_i = k \bmod a_1^w \end{cases}$$

Let us illustrate the method using Algorithm 4.1 with the following example.

Example 4.2. The *mb*NAF representation of 9750 obtained with Algorithm 4.1 using the division sequence $\frac{9750}{2} \to \frac{4875}{3} \to 1625-1 \to \frac{1624}{8} \to 203+1 \to \frac{204}{4} \to \frac{51}{3} \to 17-1 \to \frac{16}{16} \to 1$ is $(2,3)\text{NAF}_2(9750) = 1^{(2)}0^{(2)}$ $0^{(2)}0^{(2)}1^{(2)}0^{(3)}0^{(2)}-1^{(2)}0^{(2)}0^{(2)}1^{(2)}0^{(3)}0^{(2)}$, which allows us to compute the corresponding scalar multiplication $9750P$ as $2\times3\,(2^3\,(2^2\times3(2^4P+P)-P)+P)$, using Horner's method. The latter involves 1mDBL + 9DBL + 2TPL + 3mADD. For instance, using Table 2.4 ($\mathcal{JQ}^e$, $1S = 0.8M$),

$9750P$ costs $107.2M$. Compare this to the cost using NAF: $\mathrm{NAF}(9750) = 1010\,{-}10\,0010\,{-}10\,{-}10$, given by 1mDBL + 12DBL + 5mADD = $119.6M$.

For brevity (and whenever understood in the context), we will refer as *the multibase NAF of an integer k* to the unique representation found through Algorithm 4.1.

4.2.1. Zero and Nonzero Density of Multibase NAF Methods

One of the attractive properties of multibase NAF representations found with Algorithm 4.1 is that the average number of operations can be precisely determined by using Markov chains. The following theorems are presented on this regard. With a slight abuse of notation, density refers to the number of certain point operation relative to the total number of zero and nonzero digits in a given representation.

Theorem 4.1. The average densities of additions, doublings and triplings for the (*w*)*mb*NAF using bases $\mathcal{A} = \{2,3\}$ are approximately:

$$\delta_1 = \frac{2^w}{3(2^{w-2} - s) + 2^w(w+1)},\quad \delta_{x^2} = \frac{2^w(w+1)}{3(2^{w-2} - s) + 2^w(w+1)},\quad \delta_{x^3} = \frac{3(2^{w-2} - s)}{3(2^{w-2} - s) + 2^w(w+1)},$$

respectively, where $s = \left\lfloor (2^{w-2}+1)/3 \right\rfloor$ and $w \geq 2 \in \mathbb{Z}^+$ ($w = 2$ for *mb*NAF).

Proof. The method can be modeled as a Markov chain with *three* states in the case of bases $\{2,3\}$: "$0^{(2)}$", "$0^{(3)}$" and "$\underbrace{0^{(2)}\ldots0^{(2)}}_{w\text{-}1}k_i^{(2)}$", with the following probability matrix:

$$\begin{pmatrix}
"0^{(2)}" & : & 1/2 & \dfrac{2^{w-2} - \left\lfloor (2^{w-2}+1)/3 \right\rfloor}{2^w} & \dfrac{2^{w-2} + \left\lfloor (2^{w-2}+1)/3 \right\rfloor}{2^w} \\
"0^{(3)}" & : & 0 & 1/3 & 2/3 \\
"\underbrace{0^{(2)}\ldots0^{(2)}}_{w\text{-}1}k_i^{(2)}" & : & 1/2 & \dfrac{2^{w-2} - \left\lfloor (2^{w-2}+1)/3 \right\rfloor}{2^w} & \dfrac{2^{w-2} + \left\lfloor (2^{w-2}+1)/3 \right\rfloor}{2^w}
\end{pmatrix}$$

This Markov chain is irreducible and aperiodic, and hence, it has stationary distribution, which is given by:

$$\left("\underbrace{0^{(2)}\ldots0^{(2)}}_{w\text{-}1}k_i^{(2)}", "0^{(2)}", "0^{(3)}" : \frac{2^w}{2^{w+1} + 3\left(2^{w-2} - s\right)} \quad \frac{2^w}{2^{w+1} + 3\left(2^{w-2} - s\right)} \quad \frac{3\left(2^{w-2} - s\right)}{2^{w+1} + 3\left(2^{w-2} - s\right)}\right).$$

Therefore, nonzero digits k_i appear 2^w out of $w \cdot 2^w + 2^w + 3\left(2^{w-2} - \left\lfloor (2^{w-2}+1)/3 \right\rfloor\right)$ digits,

which proves the assertion about the nonzero density. Doublings and triplings (i.e., number of zero and nonzero digits with bases 2 and 3, respectively) appear $2^w \cdot w + 2^w$ and $3\left(2^{w-2} - \left\lfloor (2^{w-2}+1)/3 \right\rfloor\right)$ out of $w \cdot 2^w + 2^w + 3\left(2^{w-2} - \left\lfloor (2^{w-2}+1)/3 \right\rfloor\right)$ digits, respectively. This proves assertion about the average density of doublings and triplings. □

Theorem 4.2. The average densities of additions, doublings, triplings and quintuplings for the (*w*)*mb*NAF using bases $\mathcal{A} = \{2,3,5\}$ are approximately:

$$\delta_1 = \frac{2^{w+3}}{17 \cdot 2^{w-1} - 5r - 24s - 5t + 2^{w+3}(w+1)}, \quad \delta_{x^2} = \frac{2^{w+3}(w+1)}{17 \cdot 2^{w-1} - 5r - 24s - 5t + 2^{w+3}(w+1)},$$

$$\delta_{x^3} = \frac{24(2^{w-2} - s)}{17 \cdot 2^{w-1} - 5r - 24s - 5t + 2^{w+3}(w+1)} \text{ and } \delta_{x^5} = \frac{5(2^{w-1} - r - t)}{17 \cdot 2^{w-1} - 5r - 24s - 5t + 2^{w+3}(w+1)},$$

respectively, where $r = \left\lfloor (2^{w-2}+2)/5 \right\rfloor$, $s = \left\lfloor (2^{w-2}+1)/3 \right\rfloor$ and $t = \left\lfloor (2^{w-2}+7)/15 \right\rfloor$.

Proof. For the case of bases $\mathcal{A} = \{2,3,5\}$, this method can be modeled with *four* states: "$0^{(2)}$", "$0^{(3)}$", "$0^{(5)}$" and "$\underbrace{0^{(2)} \ldots 0^{(2)}}_{w\text{-}1} k_i^{(2)}$". The probability matrix in this case is as follows:

$$\begin{pmatrix}
\text{"}0^{(2)}\text{"} & : & 1/2 & \frac{2^{w-2}-s}{2^w} & \frac{2^{w-2}-r+s-t}{2^{w+2}} & \frac{3\cdot 2^{w-2}+r+3s+t}{2^{w+2}} \\
\text{"}0^{(3)}\text{"} & : & 0 & 1/3 & 1/6 & 1/2 \\
\text{"}0^{(5)}\text{"} & : & 0 & 0 & 1/5 & 4/5 \\
\text{"}\underbrace{0^{(2)} \ldots 0^{(2)}}_{w\text{-}1} k_i^{(2)}\text{"} & : & 1/2 & \frac{2^{w-2}-s}{2^w} & \frac{2^{w-2}-r+s-t}{2^{w+2}} & \frac{3\cdot 2^{w-2}+r+3s+t}{2^{w+2}}
\end{pmatrix}$$

This Markov chain is irreducible and aperiodic with stationary distribution:

$$\left(\text{"}\underbrace{0^{(2)} \ldots 0^{(2)}}_{w\text{-}1} k_i^{(2)}\text{"}, \text{"}0^{(2)}\text{"}, \text{"}0^{(3)}\text{"}, \text{"}0^{(5)}\text{"} \; : \; \frac{2^{w+3}}{\omega} \quad \frac{2^{w+3}}{\omega} \quad \frac{24\left(2^{w-2}-s\right)}{\omega} \quad \frac{5\left(2^{w-1}-r-t\right)}{\omega} \right),$$

where $\omega = 49 \cdot 2^{w-1} - 5r - 24s - 5t$. Therefore, nonzero digits k_i appear 2^{w+3} out of $2^{w+3} \cdot w + 2^{w+3} + 24\left(2^{w-2}-s\right) + 5\left(2^{w-1}-r-t\right)$ digits, which proves our assertion about the nonzero density. Doublings, triplings and quintuplings (i.e., number of zero and nonzero digits with bases 2, 3 and 5, respectively) appear $2^{w+3} \cdot w + 2^{w+3}$, $24\left(2^{w-2}-s\right)$ and $5\left(2^{w-1}-r-t\right)$ out of $2^{w+3} \cdot w + 2^{w+3} + 24\left(2^{w-2}-s\right) + 5\left(2^{w-1}-r-t\right)$ digits, respectively. This proves our assertion about the average density for the aforementioned operations. □

Let us determine the average number of operations for the multibase NAF method with the

help of the presented theorems. First, it is known that the expected number of doublings, triplings and additions is given by $\#\text{DBL}=\delta_{x^2}\cdot digits$, $\#\text{TPL}=\delta_{x^3}\cdot digits$ and $\#\text{ADD}=\delta_1\cdot digits$, where *digits* represents the total number of (zero and nonzero) digits in the expansion (note that a nonzero digit involves one doubling and one addition). Then, we can assume that $2^{\#\text{DBL}}\cdot 3^{\#\text{TPL}}\approx 2^{n-1}$, where n represents the average bitlength of the scalar k. Thus, $\#\text{DBL}\cdot\log 2+\#\text{TPL}\cdot\log 3\approx(n-1)\log 2$, and replacing #DBL and #TPL, we can estimate *digits* with the following:

$$digits\approx\frac{(n-1)\log 2}{\delta_{x^2}\cdot\log 2+\delta_{x^3}\cdot\log 3}, \tag{4.5}$$

which allow us to determine #DBL, #TPL and #ADD using the expressions above and Theorem 4.1. For instance, in the case of *mb*NAF, bases $\mathcal{A}=\{2,3\}$ and w = 2, the average densities for doublings, triplings and additions derived from Theorem 4.1 are 4/5, 1/5 and 4/15. If n = 160 bits, we determine that *digits* = 142.35 using (4.5). Then, the average cost of a scalar multiplication using Table 2.4 ($\mathcal{JQ}^e$, $1S=0.8M$) is approximately 113.88DBL + 28.47TPL + 37.96mADD = 1321*M*. Similarly, if we use bases $\mathcal{A}=\{2,3,5\}$, the average cost can be estimated as approximately 97.06DBL + 24.27TPL + 10.11QPL + 32.35mADD = 1299.82*M*. Compare the previous costs to that one offered by NAF: 159DBL + 53mADD = 1399.2*M* (in this case, $\delta_{\text{NAF}}=1/3$). Hence, theoretically, it is determined that (2,3)NAF and (2,3,5)NAF surpasses NAF (case with no precomputations, $\mathcal{JQ}^e$) by about 5.6% and 7.1%, respectively.

It is still possible to find more efficient multibase chains at the expense of some increment in the complexity of the original multibase NAF. The improved algorithms will be discussed in Section 4.4. Following, we optimize the basic multibase NAF methods using a recoding based on fractional windows.

4.3. The Fractional Width-*w* Multibase Non-Adjacent Form (Frac-*wmb*NAF)

One disadvantage of *wmb*NAF is that it restricts the allowed number of non-trivial precomputed points to $(2^{w-2}-1)$ for $w>2\in\mathbb{Z}^+$, following the same restriction of its analogous counterpart in the radix-2 domain, namely *w*NAF. In some settings, it is possible that the optimal performance is achieved by precomputing a number of points that do not follow such a standard window size. Also, some applications could have memory constraints different to the ones dictated by the standard windows. In this section, we apply the concept of "fractional" windows due to Möller [Möl03] to the multibase NAF method to allow a flexible number of points in the precomputed table. The new representation is called fractional *wmb*NAF (denoted by Frac-*wmb*NAF).

For the remainder, we will assume that the main base a_1 is 2. First, let us establish our ideal table with unrestricted number of non-trivial points $d_i P$, where $d_i \in D^+ \backslash \{0,1\} = \{3, 5, \ldots, m\}$ and $m \geq 3 \in \mathbb{Z}^+$ is an odd integer. If we define *m* in terms of the standard windows *w*, it would be expressed as:

$$m = 2^{w-2} + h, \tag{4.6}$$

where $2^{w-2} < m < 2^{w-1}$ and $h \geq 1 \in \mathbb{Z}^+$ is odd.

We define the rules of the recoding scheme for bases $\mathcal{A} = \{2, a_2, \ldots, a_J\}$ in Algorithm 4.2.

Algorithm 4.2. Recoding rules for "fractional" windows ($r = k \bmod 2^w$)

1: If ($k \bmod 2 = 0$ or $k \bmod a_2 = 0$ or … or $k \bmod a_J = 0$), then $k_i = 0$
2: Elseif $0 < r \leq m$, then $k_i = r$
3: Elseif $m < r < (3m - 4h)$, then $k_i = r - 2^{w-1}$
4: Elseif $(3m - 4h) \leq r < 2^w$, then $k_i = r - 2^w$
5: $k = k - k_i$

Basically, the proposed recoding first detects if *k* is divisible by one of the bases. Else, it establishes a window *w* and checks if *k* can be approximated to the closest extreme of the window using any of the digits d_i available. It can be verified that the latter will be accomplished if steps 2 or 4 are satisfied. Otherwise, the established window is too large and, hence, it is "reduced" to the immediately preceding window size to which *k* can be approximated (condition in step 3).

An algorithm to convert any integer to Frac-*wmb*NAF representation can be easily derived by replacing steps 3-6 in Algorithm 4.1 by steps 1-5 of Algorithm 4.2. In this case, we will denote the Frac-*wmb*NAF of an integer *k* by $(2, a_2, ..., a_J)\mathrm{NAF}_{w,L}(k) = (..., k_2^{(b_2)}, k_1^{(b_1)})$, where *w* is the standard window width according to (4.6) and *L* represents the number of precomputed points, that is, $L = (m-1)/2$.

Let us illustrate the new recoding with the following example.

Example 4.3. If $k = 9750$ and $m = 5$, then $d_i \in D^+ \backslash \{0,1\} = \{3,5\}$, and $w = 4$ and $h = 1$ by means of eq. (4.6). Then, the Frac-*wmb*NAF of 9750 is $(2,3)\mathrm{NAF}_{4,2}(9750) = 1^{(2)}0^{(2)}0^{(2)}0^{(2)} - 3^{(2)}0^{(2)}0^{(2)}0^{(2)} - 5^{(2)}0^{(2)}0^{(2)}1^{(2)}0^{(3)}0^{(2)}$, and the conversion process can be visualized as the division chain $\frac{9750}{2} \to \frac{4875}{3} \to 1625 - 1 \to \frac{1624}{8} \to 203 + 5 \to \frac{208}{16} \to 13 + 3 \to \frac{16}{16} \to 1$.

Observe that, when 1625 is obtained, it requires an addition with 7 to approximate the value to 1632 (which is the closest number $\equiv 0 \pmod{2^4}$, as required by a standard window $w = 4$). However, 7 is not part of the precomputed table, so the window width is reduced accordingly to

$w=3$ and the value 1625 is approximated to the closest value in the new window (i.e., 1624) using an addition with -1.

We now present the following theorem regarding the average density of this method for the case $\mathcal{A}=\{2,3\}$.

Theorem 4.3. The average densities of nonzero terms, doublings and triplings of the Frac-*wmb*NAF using bases $\mathcal{A}=\{2,3\}$, window size w and L available points (represented by $(2,3)\text{NAF}_{w,L}$) are approximately:

$$\delta_1=\frac{2^w}{8(L+1)-3(u+v)+2^{w-2}(4w-1)},\quad \delta_{x^2}=\frac{8(L+1)+2^w(w-1)}{8(L+1)-3(u+v)+2^{w-2}(4w-1)}\quad\text{and}$$

$$\delta_{x^3}=\frac{3(2^{w-2}-(u+v))}{8(L+1)-3(u+v)+2^{w-2}(4w-1)},$$

respectively, where $u=\lfloor (L+2)/3\rfloor$ and $v=\lfloor (2^{w-2}-L)/3\rfloor$.

Proof. Let us consider the following states to model this fractional window method using Markov chains: "$0^{(2)}$", "$0^{(3)}$", "$\underbrace{0^{(2)}\ldots 0^{(2)}}_{w\text{-}2}k_i^{(2)}$" and "$\underbrace{0^{(2)}\ldots 0^{(2)}}_{w\text{-}1}k_i^{(2)}$". Then, the probability matrix is as follows:

$$\begin{pmatrix}
\text{"}0^{(2)}\text{"} & : & 1/2 & \frac{t-\lfloor (t+1)/3\rfloor}{4t} & \frac{\left(2^{w-2}-t\right)\left(t+\lfloor (t+1)/3\rfloor\right)}{2^w t} & \frac{t+\lfloor (t+1)/3\rfloor}{2^w} \\
\text{"}0^{(3)}\text{"} & : & 0 & 1/3 & \frac{2^{w-2}-t}{3\cdot 2^{w-3}} & \frac{t}{3\cdot 2^{w-3}} \\
\text{"}\underbrace{0^{(2)}\ldots 0^{(2)}}_{w\text{-}2}k_i^{(2)}\text{"} & : & 0 & \alpha=\frac{(2^{w-2}-t)-\lfloor (2^{w-2}-t+1)/3\rfloor}{2(2^{w-2}-t)} & \beta=\frac{(2^{w-2}-t)+\lfloor (2^{w-2}-t+1)/3\rfloor}{2^{w-1}} & 1-\alpha-\beta \\
\text{"}\underbrace{0^{(2)}\ldots 0^{(2)}}_{w\text{-}1}k_i^{(2)}\text{"} & : & 1/2 & \frac{t-\lfloor (t+1)/3\rfloor}{4t} & \frac{\left(2^{w-2}-t\right)\left(t+\lfloor (t+1)/3\rfloor\right)}{2^w t} & \frac{t+\lfloor (t+1)/3\rfloor}{2^w}
\end{pmatrix}$$

This Markov chain is irreducible and aperiodic with the stationary distribution:

$$\left(\text{"}0^{(2)}\text{"},\text{"}0^{(3)}\text{"},\text{"}\underbrace{0^{(2)}\ldots 0^{(2)}}_{w\text{-}2}k_i^{(2)}\text{"},\text{"}\underbrace{0^{(2)}\ldots 0^{(2)}}_{w\text{-}1}k_i^{(2)}\text{"}:\ \frac{16t}{\mu}\ \ \frac{12(2^{w-2}-(u+v))}{\mu}\ \ \frac{16(2^{w-2}-t)}{\mu}\ \ \frac{16t}{\mu}\right),$$

where $\mu=16t-12(u+v)+7\cdot 2^w$ and $t=L+1$. Therefore, the nonzero digits k_i appear 2^w out of $8t-3(u+v)+2^{w-2}(4w-1)$ digits, proving the assertion about the nonzero density. Doublings and triplings (i.e., the number of zero and nonzero digits with bases 2 and 3, respect.) appear $8t+2^w(w-1)$ and $3\left(2^{w-2}-(u+v)\right)$ out of $8t-3(u+v)+2^{w-2}(4w-1)$ digits, respectively. This proves the assertion about the average density of doublings and triplings. □

With Theorem 4.3, it is possible to theoretically estimate the expected number of doublings, triplings and additions using this method. For instance, following the procedure detailed in Section 4.2.1, we can estimate the cost of scalar multiplication (without including precomputation cost) for $n = 160$ bits using $L = 2$ points ($w = 4$) as 132.7DBL + 16.6TPL + 29.5mADD = $1229.9M$ ($\mathcal{JQ}^e$, $1S = 0.8M$). Compare to the cost achieved by Frac-wNAF, namely 159DBL + 35.3mADD = $1250.5M$ ($\delta_{\text{Frac-}w\text{NAF}} = 1/4.5$ when using $m = 5$; see Section 2.2.4.3). Further cost reductions are observed for the case of $\mathcal{A} = \{2,3,5\}$.

4.4. A Methodology to Find Faster Multibase Chains

The multibase NAF method and its variants are simple and straightforward to implement and analyze theoretically. However, if we increase the complexity of the derivation algorithms it is still possible to find more efficient multibase chains. In this section, we propose a new methodology based on the *operation cost per bit* for deriving multibase algorithms. The method is illustrated in detail for the case of bases $\{2,3\}$.

Definition 4.1. The operation cost per bit of an elliptic curve point operation is given by $\varsigma(operation) = cost(operation)/bitlength(operation)$.

Following a common practice in the literature, we express operation costs in terms of field multiplications and squarings, assuming the approximation $1S = 0.8M$. For instance, a point doubling in Jacobian coordinates costs $\varsigma(\text{DBL}) = \text{DBL}/\log_2 2 = 7$ field multiplications per bit, where DBL = $3M + 5S$ (see Table 2.2).

Note that the definition above can be readily extended to division sequences. In this case, one should take into account the cost of all the operations involved and their corresponding bitlengths.

Corollary 4.1. From all possible chains using a given set of bases $\mathcal{A}$, the *optimal* chains for a given integer k are the ones with the lowest cost per bit.

If, for instance $\mathcal{A} = \{2,3\}$, Corollary 4.1 implies that the optimal chains for a given integer k have $\varsigma(chain) = \dfrac{\#\text{DBL}\times\text{DBL} + \#\text{TPL}\times\text{TPL} + \#\text{ADD}\times\text{ADD}}{\#\text{DBL}\times bitlength(\text{DBL}) + \#\text{TPL}\times bitlength(\text{TPL}) + \#\text{ADD}\times bitlength(\text{ADD})}$ minimal, where OP and #OP denote the cost of certain operation and the number of times this operation is used, respectively. With a slight abuse of notation, *bitlength*(ADD) represents the number of bits added or subtracted from the total bitlength after addition with a digit from a given digit set.

Obviously, an exhaustive search evaluating costs per bit of all possible division sequences from k would yield the optimal chains for this scalar. Nevertheless, for cryptographic purposes, one should constrain the search to "smaller" ranges. For instance, it seems natural to limit the

cost-per-bit evaluation to sequences between additions.

The following proposition slightly relaxes the definition of an optimal chain while simplifies significantly the cost analysis.

Proposition 4.1. Let a digit set $D\setminus\{0\}=\{\pm1,\pm3,\pm5,\ldots,\pm m\}$, where $m<<k$ for a scalar multiplication *kP*. Then, the "bitlength" of an addition with any digit $d_i \in D\setminus\{0\}$ (i.e., the bit reduction or increase due to the addition operation) is negligible in comparison with the total bitlength and approximates to zero in average.

Following Proposition 4.1, we can eliminate *bitlength*(ADD) from the denominator of $\varsigma(chain)$ without losing too much precision in our cost approximations. For the remainder, we focus our analysis on "measuring" costs between additions. As stated, nothing really deters from extending the cost analysis to wider ranges of division sequences, in which case cheaper multibase chains could be found at the expense of a more complex "searching" algorithm.

Proposition 4.2. Let $\mathcal{A}=\{a_1,a_2,\ldots,a_J\}$ be a set of bases where $a_j \in \mathcal{A}$ are all primes and $a_1^{w_1}\cdot\ldots\cdot a_J^{w_J} \mid (k-k_i)$ for an integer *k*, a digit $k_i \in D=\{0,\pm1,\pm3,\pm5,\ldots,\pm m\}$ and integers $w_j \geq 0$. Then, for given values *kP*, $k_iP \in E(\mathbb{F}_p)$, the cost per bit of computing $a_1^{w_1}\cdot\ldots\cdot a_J^{w_J}\cdot(kP-k_iP)$, which is denoted by $\varsigma(k-k_i)$, can be estimated as follows:

$$\varsigma(k-k_i)=\frac{w_1(a_1P)+w_2(a_2P)+\ldots+w_J(a_JP)+v\cdot\mathrm{ADD}}{w_1\log_2 a_1+w_2\log_2 a_2+\ldots+w_J\log_2 a_J}, \tag{4.7}$$

where a_jP represents the cost of the point operation corresponding to base a_j (for instance, $a_jP=\mathrm{DBL}$ if $a_j=2$) and *v* represents the number of additions such that $v=2$ if $k_i\neq0$ and $v=1$ if $k_i=0$.

Equation (4.7) employs the function cost(*operations*)/bitlength(*operations*) to determine the cost per bit and can be used to compare the practical efficiency of various possible sequences $a_1^{w_1}\cdot\ldots\cdot a_J^{w_J}$ between an addition with a digit k_i and the next addition.

There are different ways of exploiting this tool for finding efficient multibase chains. For instance, the costs per bit of the possible sequences can be calculated and compared *on-the-fly*. Another approach would involve the use of *on-line* congruency tests with pre-determined combinations of bases for which the costs per bit are known [LG09]. In this case, the evaluated congruencies are fixed off-line according to the chosen curve, window size *w* and set of bases $\mathcal{A}$.

In this thesis, we choose to implement the second approach based on *on-line* congruency evaluations. A detailed description of the method follows.

4.4.1. Refined Multibase Chains

In this section we propose a new algorithm that has been derived by rewriting the original multibase NAF and adding a few conditional statements. The refined multibase algorithm is shown as Algorithm 4.3. In the remainder, we will refer to chains obtained from this algorithm as *refined multibase chains*.

Algorithm 4.3. Computing "refined" multibase chains of a positive integer

Input: scalar k, bases $\mathcal{A}=\{a_1,a_2,\ldots,a_J\}$, where $a_j \in \mathbb{Z}^+$ are primes for $1 \le j \le J$, window $w \ge 2$, where $w \in \mathbb{Z}^+$
Output: a multibase chain $(\ldots,k_2^{(b_2)},k_1^{(b_1)})$, where $b_i \in \mathcal{A}$

1: $i = 1$
2: While $k > 0$ do
3: If $k \bmod a_1 = 0$, $k_i = 0$, $b_i = a_1$
4: Elseif $k \bmod a_2 = 0$
4.1: If *CONDITION2.1* = true, $k_i = d_i$, $b_i = a_1$
4.2: Else $k_i = 0$, $b_i = a_2$
⋮ ⋮
J+2: Elseif $k \bmod a_J = 0$
(J+2).1: If *CONDITION2.(J−1)* = true, $k_i = d_i$, $b_i = a_1$
(J+2).2: Else $k_i = 0$, $b_i = a_J$
J+3: Else
J+4: $b_i = a_1$, $k_i = k \bmods a_1^w$
J+5: If *CONDITION1* = true, $k_i = d_i$
J+6: $k = (k - k_i)/b_i$
J+7: $i = i + 1$
J+8: Return $(\ldots,k_2^{(b_2)},k_1^{(b_1)})$

We remark that Algorithm 4.3 is a straightforward generalization of the Refined *mb*NAF algorithm introduced by the author and Gebotys in PKC2009 [LG09] to a generic set of bases $\mathcal{A}=\{a_1,a_2,\ldots,a_J\}$. Moreover, statements have been reordered and modified to improve readability. To add the capability of using fractional windows, one should simply replace $k_i = k \bmods a_1^w$ in step (J+4) of Algorithm 4.3 by steps 2-4 of Algorithm 4.2.

Similar to multibase NAF, Algorithm 4.3 evaluates congruency with a pre-ordered set of bases $\mathcal{A}=\{a_1,a_2,\ldots,a_J\}$, which again is chosen according to the targeted setting and fixes a_1 as the main base. The main difference is the insertion of conditional statements that are intended for

evaluating division sequences according to the cost per bit function (4.7).

Now, let k be a partial value of the scalar during execution of Algorithm 4.3, $w_j, w_j' \geq 0$ be integers, $d_i, k_i \in D = \{0, \pm 1, \pm 3, \pm 5, \ldots, \pm(2^{w-1} - 1)\}$, $d_i \neq 0$, with a standard window width w, e be a parameter > 0 and ϕ, μ be odd integers such that $\{a_1, a_2, \ldots, a_J\} \nmid \phi, \mu$. The conditional statements to be inserted in Algorithm 4.3 follow the next criteria:

- *CONDITION1*: given an approximation $k - k_i = a_1^{w} \cdot a_2^{w_2} \cdot \ldots \cdot a_J^{w_J} \cdot \phi$ using a standard window w, if there exists some value d_i such that $k - d_i = a_1^{w_1'} \cdot a_2^{w_2'} \cdot \ldots \cdot a_J^{w_J'} \cdot \mu$, $d_i \neq k_i$, and its associated cost per bit $\varsigma(k - d_i)$ is lower than the cost per bit $\varsigma(k - k_i)$ associated to the sequence guaranteed by the standard window w, that is, if $\varsigma(k - d_i) + e < \varsigma(k - k_i)$, the approximation $k - d_i$ replaces $k - k_i$.
- *CONDITION2.j*, for $1 \leq j \leq J - 1$: given the partial scalar value $k = a_2^{w_2} \cdot \ldots \cdot a_J^{w_J} \cdot \phi$, if there is a nonzero digit $d_i \in D \backslash \{0\}$ such that $k - d_i = a_1^{w_1'} \cdot a_2^{w_2'} \cdot \ldots \cdot a_J^{w_J'} \cdot \mu$ and its associated cost per bit $\varsigma(k - d_i)$ is lower than the cost per bit $\varsigma(k)$, that is, if $\varsigma(k - d_i) + e < \varsigma(k)$, the approximation $k - d_i$ replaces the zero digit $k_i = 0$.

CONDITION1 aims at reducing the length of the expansion by using more expensive point operations (i.e., operations with bases a_j, where $j > 1$) that yield *cheaper-per-bit* chains than the usual sequence of operations with base a_1, after each nonzero term. Similarly, *CONDITION2* determines if there is a chain involving an addition that is *cheaper-per-bit* than the sequence directly dividing by the bases.

Note that if one assumes that values after executing a given sequence followed by an addition are approximately uniformly distributed over odd numbers, then choosing the *cheapest-per-bit* sequence for a partial value k would ultimately yield a multibase chain for the full point multiplication that is cheaper in average. However, Algorithm 4.3 does not necessarily execute the full sequence that was chosen. It instead re-evaluates and analyzes the costs of new sequences after each doubling, tripling or quintupling. Hence, *CONDITION1* and *2* above include a security parameter, namely e, to guarantee that the chosen sequence is *significantly* better than the usual one.

Although the number of divisibility tests with different combinations of bases $\mathcal{A}$ that can be evaluated in *CONDITION1* and *2* of Algorithm 4.3 is potentially high, we show in the following that only a few tests are necessary to achieve performance (conjecturally) close to optimal.

Next, we illustrate the design of efficient *CONDITION1* and *2* for the case $\mathcal{A} = \{2, 3\}$. Since extension to other cases easily follows, we simply sketch the design for the case $\mathcal{A} = \{2, 3, 5\}$ (see Section 4.4.1.2). As before, we fix the main base a_1 to 2.

4.4.1.1. Refined Multibase Chains with Bases {2,3}

We discuss next the design of *CONDITION1* and *2* for the case $\mathcal{A}=\{2,3\}$.

CONDITION1:

Following the criteria discussed previously and given a partial scalar *k*, standard window width *w* and set of bases $\mathcal{A}=\{a_1,a_2\}=\{2,3\}$, where $a_j\in\mathbb{Z}^+$, we propose the following format for *CONDITION1* in Algorithm 4.3:

$$\begin{aligned}
1:&\quad \text{if } ((k-d_{i,1}) \bmod 2^{w'_{1,1}}\cdot 3^{w'_{2,1}}=0 \text{ and } (k-k_i)\bmod 2^{w+1}\neq 0) \text{ or } \ldots\\
2:&\quad ((k-d_{i,2}) \bmod 2^{w'_{1,2}}\cdot 3^{w'_{2,2}}=0 \text{ and } (k-k_i)\bmod 2^{w+2}\neq 0) \text{ or } \ldots\\
&\vdots\\
C:&\quad ((k-d_{i,C}) \bmod 2^{w'_{1,C}}\cdot 3^{w'_{2,C}}=0 \text{ and } (k-k_i)\bmod 2^{w+C}\neq 0)
\end{aligned} \tag{4.8}$$

where $w'_{j,c}\geq 0$ are integers, $k_i=k \bmod 2^w$, *c* is the condition number such that $1\leq c\leq C$ and $d_{i,c}\in D\setminus\{0\}=\{\pm1,\pm3,\pm5,\ldots,\pm(2^{w-1}-1)\}$. In order to guarantee a cheaper-per-bit sequence at each evaluation of *CONDITION1* it is required that $\varsigma(k-d_{i,c})+e_c<\varsigma(k-k_i)$, which compares the sequence costs up to the next addition using positive values e_c for $1\leq c\leq C$. Using function (4.7), this is roughly equivalent to the following comparison:

$$\frac{w'_{1,c}\cdot \mathrm{DBL}+w'_{2,c}\cdot \mathrm{TPL}+2\mathrm{ADD}}{w'_{1,c}+w'_{2,c}\log_2 3}+e_c<\frac{(w+c-1)\cdot \mathrm{DBL}+2\mathrm{ADD}}{w+c-1}. \tag{4.9}$$

We next illustrate the procedure for selecting values $w'_{j,c}$ and e_c for format (4.8) using eq. (4.9) when $w=2$. The procedure can be easily extended to other window sizes.

First, we build *two* tables: one with the costs per bit corresponding to sequences containing exactly *d* doublings (for congruency of $(k-k_i)$) and another with the costs per bit corresponding to sequences divisible by $2^{w'_{1,c}}\cdot 3^{w'_{2,c}}$ (for congruency of $(k-d_{i,c})$). Note that since $w=2$ it always holds that $w'_{1,c}=1$. We show in Table 4.1 the results for Jacobian coordinates using costs from Table 2.2 (assuming that $1S=0.8M$). Since $w=2$ calculations are performed with mixed additions (the cost of one mixed addition is obtained as $m\mathrm{ADD}=m\mathrm{DBLADD}-\mathrm{DBL}$).

Using Table 4.1, it is easy to see that $(k-k_i)\equiv \bmod 4, (k-k_i)\not\equiv \bmod 8$ yields a sequence that is more expensive per bit than, at least, $(k-d_{i,c})\equiv \bmod 3$; $(k-k_i)\equiv \bmod 8, (k-k_i)\not\equiv \bmod 16$ yields a sequence that is more expensive per bit than, at least, $(k-d_{i,c})\equiv \bmod 9$; $(k-k_i)\equiv \bmod 16, (k-k_i)\not\equiv \bmod 32$ yields a sequence that is more expensive per bit than, at least, $(k-d_{i,c})\equiv \bmod 27$; and so on. This analysis gives a close idea about the statements that should be defined in (4.8) for *CONDITION1*. In fact, if we plug the congruency evaluations above

Table 4.1. Cost-per-bit for statements in *CONDITION1*, bases {2,3}, $w = 2$, $\mathcal{J}$ coordinates.

Congruency of $(k - k_i)$	d	Cost per bit	Congruency of $(k - d_{i,c})$	$(w'_{1,c},\ w'_{2,c})$	Cost per bit
$\equiv \bmod 4, \not\equiv \bmod 8$	2	16.6	$\equiv \bmod 3$	(1, 1)	15.0
$\equiv \bmod 8, \not\equiv \bmod 16$	3	13.4	$\equiv \bmod 9$	(1, 2)	12.3
$\equiv \bmod 16, \not\equiv \bmod 32$	4	11.8	$\equiv \bmod 27$	(1, 3)	11.1
$\equiv \bmod 32, \not\equiv \bmod 64$	5	10.8	$\equiv \bmod 81$	(1, 4)	10.4
$\equiv \bmod 64, \not\equiv \bmod 128$	6	10.2	$\equiv \bmod 243$	(1, 5)	10.0
$\equiv \bmod 128, \not\equiv \bmod 256$	7	9.7	$\equiv \bmod 729$	(1, 6)	9.7
$\equiv \bmod 256, \not\equiv \bmod 512$	8	9.4			

to (4.8) for conditions c = 1, 2, 3, and so on (in that order), the multibase chains obtained are expected to be cheaper in average than those produced by the case without conditions (i.e., *mb*NAF, given by Algorithm 4.1). Nevertheless, choosing the minimal condition for which congruency with $(k - k_i)$ is more expensive is not necessarily optimal. In other words, it is still possible to do better by choosing the optimal parameter e_c for each case.

For the latter, it is necessary to perform an analysis of costs of the possible combinations. For instance, consider the evaluation "if $((k - d_{i,1}) \bmod 3^t = 0$ and $(k - k_i) \bmod 8 \neq 0)$" with $w = 2$, $t \geq 1 \in \mathbb{Z}$ and $C = c = 1$ in (4.8) to implement *CONDITION1*. The cost per bit in this case is approximately given by:

$$\frac{1}{4}\left(\alpha + \frac{2\text{DBL} + 1\text{TPL} + 2\text{ADD}}{2 + \log_2 3}\right) + \frac{1}{4}\left(\frac{3\text{DBL} + 2\text{ADD}}{3} + \frac{3\text{DBL} + 1\text{TPL} + 2\text{ADD}}{3 + \log_2 3}\right), \tag{4.10}$$

where $\alpha = \left(1 - \frac{1}{3^{(t-1)}}\right)\left(\frac{1}{2} \times \frac{2\text{DBL} + 2\text{ADD}}{2} + \frac{1}{2} \times \frac{3\text{DBL} + 2\text{ADD}}{3}\right) + \frac{1}{3^{(t-1)}}\left(\frac{1\text{DBL} + t \cdot \text{TPL} + 2\text{ADD}}{1 + t \cdot \log_2 3}\right)$.

It can be seen from (4.10) that optimality is achieved with $\min(\alpha)$. For example, for $\mathcal{J}$, $\mathcal{JQ}^e$ and $\mathcal{IE}$ coordinates (using operation costs from Tables 2.2, 2.3 and 2.4 and assuming $1S = 0.8M$), $\min(\alpha)$ is obtained with $t = 2$. Notice that analysis in α can go deeper and include a higher number of consecutive doublings and triplings. However, the occurrence decreases rapidly with the number of consecutive operations and so their impact in the cost. A similar analysis can be carried out to determine optimal values for following conditions *c* in (4.8).

Additionally, it is necessary to determine the influence of *C* in the cost performance. A probability analysis similar to the one performed above can be carried out to determine the optimal *C*. However, the analysis increases in complexity very rapidly. Instead, we ran several

tests to evaluate the cost performance of full 160-bit scalar multiplications. The results are discussed in the subsection "Analysis of Multiple Conditions", pp. 90.

CONDITION2:

Following the criteria discussed previously and given a partial scalar k, standard window width w and set of bases $\mathcal{A} = \{a_1, a_2\} = \{2,3\}$, where $a_j \in \mathbb{Z}^+$, we propose the following format for *CONDITION2* in Algorithm 4.3:

$$\begin{aligned} 1: &\quad \text{if } ((k - d_{i,1}) \bmod 2^{w'_{1,1}} = 0 \text{ and } k \bmod 3^2 \neq 0) \text{ or } \ldots \\ 2: &\quad ((k - d_{i,2}) \bmod 2^{w'_{1,2}} = 0 \text{ and } k \bmod 3^3 \neq 0) \text{ or } \ldots \\ &\vdots \\ C: &\quad ((k - d_{i,C}) \bmod 2^{w'_{1,C}} = 0 \text{ and } k \bmod 3^{C+1} \neq 0) \end{aligned} \tag{4.11}$$

where again $w'_{j,c} \geq 0$ are integers, c is the condition number s.t. $1 \leq c \leq C$, and $d_{i,c} \in D \setminus \{0\} = \{\pm 1, \pm 3, \pm 5, \ldots, \pm(2^{w-1} - 1)\}$. To guarantee a cheaper-per-bit sequence at each evaluation of *CONDITION2* it is required that $\varsigma(k - d_{i,c}) + e_c < \varsigma(k)$, which compares the sequence costs up to the next addition using positive values e_c for $1 \leq c \leq C$. Using function (4.7), this is roughly equivalent to the following comparison:

$$\frac{1}{2}\left(\frac{w'_{1,c} \cdot \mathrm{DBL} + 2\mathrm{ADD}}{w'_{1,c}} + \frac{(w'_{1,c} + 1) \cdot \mathrm{DBL} + 2\mathrm{ADD}}{w'_{1,c} + 1} \right) + e_c < \frac{(c+1)\mathrm{TPL} + 1\mathrm{ADD}}{(c+1) \cdot \log_2 3}. \tag{4.12}$$

Let us now illustrate the procedure for selecting values $w'_{j,c}$ and e_c for format (4.11) using eq. (4.12) when $w = 2$.

Similarly to the case with *CONDITION1*, we first build *two* tables: one with the costs per bit corresponding to sequences divisible by $2^{w'_{1,c}}$ (for congruency of $(k - d_{i,c})$) and another with the costs per bit corresponding to sequences with exactly t triplings (for congruency of k). In Table 4.2, we show the results for Jacobian coordinates using costs from Table 2.2 (assuming that $1S = 0.8M$). Again, we assume that $w = 2$, calculations are performed with mixed additions and the cost of one mixed addition is obtained as $m\mathrm{ADD} = m\mathrm{DBLADD} - \mathrm{DBL}$.

Using Table 4.2, we can see that $k \equiv \bmod 3, k \not\equiv \bmod 9$ yields a sequence that is more expensive per bit than, at least, $(k - d_{i,c}) \equiv \bmod 8$; $k \equiv \bmod 9, k \not\equiv \bmod 27$ yields a sequence that is more expensive per bit than, at least, $(k - d_{i,c}) \equiv \bmod 32$; $k \equiv \bmod 27, k \not\equiv \bmod 81$ yields a sequence that is more expensive per bit than, at least, $(k - d_{i,c}) \equiv \bmod 128$; and so on. If these congruency evaluations are directly plugged into (4.11) for conditions c = 1, 2, 3, and so on (in that order), the multibase chains obtained are expected to be cheaper in average than those produced

Table 4.2. Cost-per-bit for statements in *CONDITION2*, bases {2,3}, $w = 2$, $\mathcal{J}$ coordinates.

Congruency of k	t	Cost per bit	Congruency of $(k - d_{i,c})$	$w'_{1,c}$	Cost per bit
$\equiv \bmod 3, \not\equiv \bmod 9$	1	14.0	$\equiv \bmod 4$	2	15.0
$\equiv \bmod 9, \not\equiv \bmod 27$	2	11.0	$\equiv \bmod 8$	3	12.6
$\equiv \bmod 27, \not\equiv \bmod 81$	3	10.0	$\equiv \bmod 16$	4	11.3
$\equiv \bmod 81, \not\equiv \bmod 243$	4	9.5	$\equiv \bmod 32$	5	10.5
$\equiv \bmod 243, \not\equiv \bmod 729$	5	9.2	$\equiv \bmod 64$	6	10.0
$\equiv \bmod 729, \not\equiv \bmod 2187$	6	9.0	$\equiv \bmod 128$	7	9.6
			$\equiv \bmod 256$	8	9.3

by the case without conditions (i.e., *mb*NAF; Algorithm 4.1). However, choosing the minimal condition for which congruency with k is more expensive is not necessarily optimal. In this case, it is necessary to perform a more in detail analysis of costs of the possible combinations. For instance, consider the evaluation "if $((k - d_{i,1}) \bmod 2^d = 0$ and $k \bmod 9 \neq 0)$" with $w = 2$, $d > 1 \in \mathbb{Z}$ and $C = c = 1$ in (4.11) to implement *CONDITION2* in Algorithm 4.1. The cost per bit in this case is approximately given by:

$$\frac{2}{3}\beta + \frac{1}{3}\left(\frac{2\text{TPL} + 1\text{ADD}}{2\log_2 3}\right), \tag{4.13}$$

where $\beta = \left(1 - \frac{1}{2^{d-2}}\right)\left(\frac{2}{3} \times \frac{1\text{TPL} + 1\text{ADD}}{\log_2 3} + \frac{1}{3} \times \frac{2\text{TPL} + 1\text{ADD}}{2\log_2 3}\right) + \frac{1}{2^{d-2}}\left(\frac{1}{2} \times \frac{d \cdot \text{DBL} + 2\text{ADD}}{d} + \frac{1}{2} \times \frac{(d+1) \cdot \text{DBL} + 2\text{ADD}}{d+1}\right)$.

By analyzing (4.13), it can be seen that optimality is achieved with $\min(\beta)$. For instance, for $\mathcal{J}$, $\mathcal{JQ}^e$ and $\mathcal{IE}$ coordinates (using operation costs from Tables 2.2, 2.3 and 2.4 and assuming $1S = 0.8M$), $\min(\beta)$ is obtained with $d = 4$. Although analysis in β can go deeper and include higher numbers of consecutive doublings and triplings, the occurrence decreases rapidly with the number of consecutive operations and so the impact in the cost. A similar analysis can be carried out to determine optimal values for following conditions c in (4.11).

In the following example, we illustrate the derivation of a multibase chain using Algorithm 4.3 with an efficient selection of parameters for *CONDITION1* (4.8) and *CONDITION2* (4.11), as discussed in this section. In the remainder, conditions from (4.8) and (4.11) are denoted by pairing values $2^{w'_{1,c}} \cdot 3^{w'_{2,c}}$ and 2^{w+c}, and values $2^{w'_{1,c}}$ and 3^{c+1}, respectively, as follows:

$$(2^{w'_{1,1}} \cdot 3^{w'_{2,1}} - 2^{w+1}, 2^{w'_{1,2}} \cdot 3^{w'_{2,2}} - 2^{w+2}, \ldots, 2^{w'_{1,C}} \cdot 3^{w'_{2,C}} - 2^{w+C} \mid 2^{w'_{1,1}} - 3^2, 2^{w'_{1,2}} - 3^3, \ldots, 2^{w'_{1,C}} - 3^{C+1}),$$

where paired values for *CONDITION1* and *2* are separated by "|". For instance, in Example 4.4 conditions denoted by (9-8|32-9) mean that Algorithm 4.3 includes the evaluation "if $((k-d_{i,1}) \bmod 9 = 0$ and $(k-k_i) \bmod 8 \neq 0)$" as *CONDITION1* and the evaluation "if $((k-d_{i,1}) \bmod 32 = 0$ and $k \bmod 9 \neq 0)$" as *CONDITION2*.

Example 4.4. Using Algorithm 4.3, we find the following refined multibase chain for computing $8821P$ by using bases $\{2,3\}$, $w = 2$ and conditions (9-8|32-9): $8821 = 1^{(3)}0^{(2)}0^{(2)}0^{(2)} - 1^{(2)}0^{(2)}0^{(2)}0^{(2)} - 1^{(2)}0^{(3)}0^{(2)}1^{(2)}$, which has been derived using the division sequence $8821-1 \rightarrow \frac{8820}{2} \rightarrow \frac{4410}{2} \rightarrow \frac{2205}{3} \rightarrow 735+1 \rightarrow \frac{736}{2} \rightarrow \frac{368}{2} \rightarrow \frac{184}{2} \rightarrow \frac{92}{2} \rightarrow \frac{46}{2} \rightarrow 23+1 \rightarrow \frac{24}{2} \rightarrow \frac{12}{2} \rightarrow \frac{6}{2} \rightarrow \frac{3}{3} \rightarrow 1$.

Notice that, for instance, the partial value 735 is conveniently approximated to 736, by means of *CONDITION1*, instead of dividing it by 3, allowing the efficient insertion of several consecutive doublings that ultimately reduce the nonzero density of the expansion. If we compare the performance of this multibase chain when computing $8821P$ against the basic multibase NAF approach using the same window size, we can observe that the cost reduces from 8DBL + 3TPL + 4mADD = 115.2*M* to only 10DBL + 2TPL + 3mADD = 107.6*M* ($\mathcal{JQ}^e$, $1S = 0.8M$).

Finally, a probability analysis can be carried out to determine the optimal *C* for *CONDITION1* and *2*. As stated before, this analysis increases in complexity very rapidly, so instead we have run many tests to evaluate the cost performance of full 160-bit scalar multiplications. The results are discussed in the following subsection.

Analysis of Multiple Conditions

The use of multiple conditions in *CONDITION1* (4.8) and *CONDITION2* (4.11) enable a wider search for cheaper multibase chains. However, as the number of conditions *C* increases the impact on the cost decreases. We have run several tests with 160-bit point multiplications to explore empirically the behavior of Algorithm 4.3 when increasing *C*. The results are displayed in Fig. 4.1-4.2.

In our tests, we average the cost of 1000 point multiplications using 160-bit random scalars. To determine the conditional statements in each case, we performed the analysis described in the previous section. We also carried out multiple tests to confirm our parameter selection and when analysis got excessively complex.

In Figures 4.1 and 4.2, *x* and *y* in (*x*|*y*) denote the number of conditions *C* for *CONDITION1* and *2*, respectively.

As can be seen, selection (1|1) achieves the higher relative speed up. As *C* increases the gain also decreases. In general, for $C \geq 2$ the costs do not vary significantly. On the negative side, this implies that even deeper searching for efficient sequences will only provide smaller speed ups. On the positive side, this feature enables the possibility of very compact algorithms that achieve,

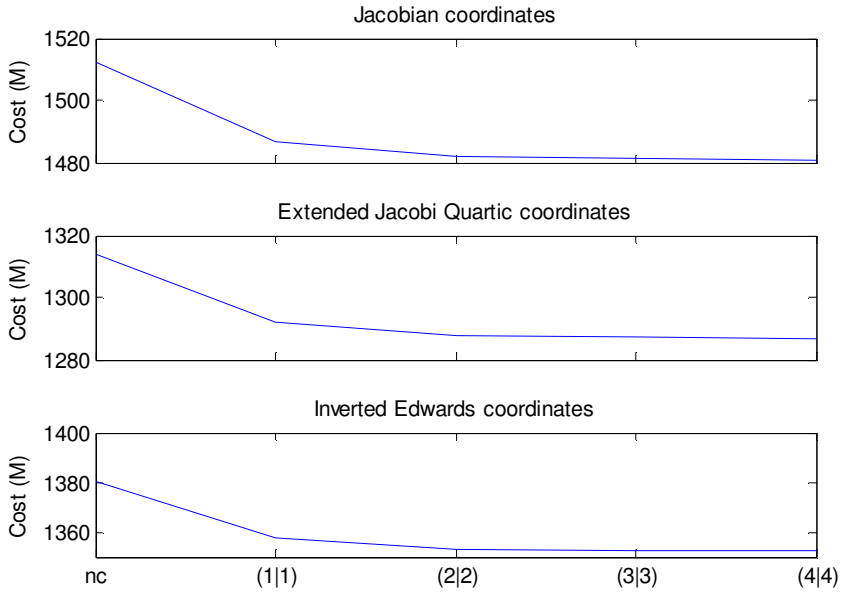


Figure 4.1. Cost of 160-bit point multiplication without precomputations using refined multibase chains. Conditional statements: nc = no conditions, (1|1) = (9-8|32-9), (2|2) = (9-8,27-16|32-9,64-27), (3|3) = (9-8,27-16,81-32|32-9,64-27,128-81) ,(4|4) = (9-8,27-16,81-32,243-64|32-9,64-27,128-81,256-243).

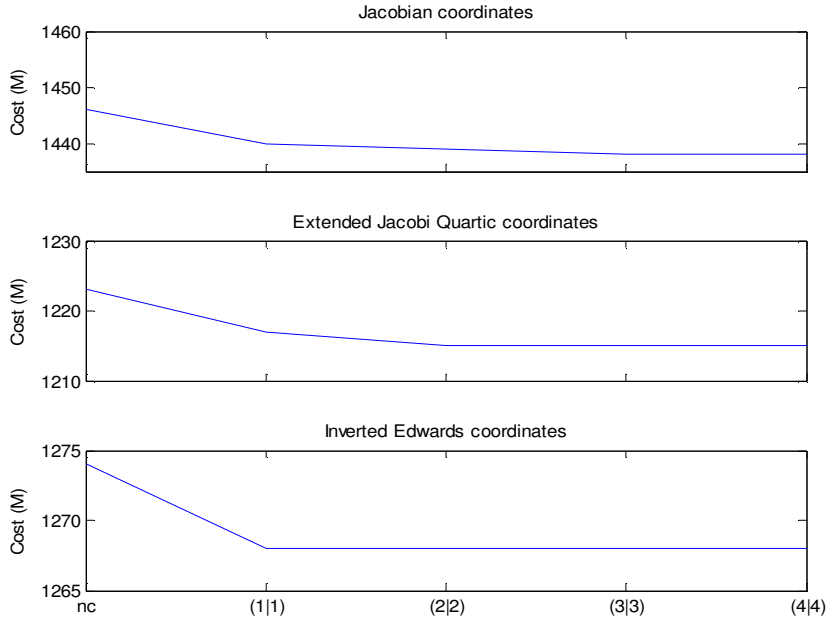


Figure 4.2. Cost of 160-bit point multiplication with $w = 5$ using refined multibase chains. Conditional statements: nc = no conditions, (1|1) = (144-64|64-9), (2|2) = (144-64,324-128|64-9,512-27), (3|3) = (144-64,324-128,648-256|64-9,512-27,1024-81), (4|4) = (144-64,324-128,648-256,972-512|64-9,512-27,1024-81,2048-243).

conjecturally, close to optimal performance.

Highly Compact Multibase Algorithms, Bases {2,3}

The generalized formats for conditions (4.8) and (4.11) allow one to perform a relatively simple and scalable analysis of Algorithm 4.3. However, following observations by Walter [Wal11] it is obvious that much more compact algorithms can be easily derived once the design parameters (i.e., conditional statements, window width and bases) are fixed. The following examples illustrate how the original algorithm for finding refined multibase chains can be rearranged to obtain very compact algorithms with fixed parameters.

Example 4.5. If we select conditions (9-8|32-9), window size $w = 2$ and bases $\{2,3\}$, then it is straightforward to transform Algorithm 4.3 and replace lines 3 to J+5 with the following:

$$\begin{cases} k_i = k \bmod 4,\ b_i = 2 \\ \text{if } k \bmod 2 = 0,\ k_i = 0 \\ \text{elseif } k \bmod 3 = 0 \text{ and } \sim\left[(k - k_i) \bmod 32 = 0 \text{ and } k \bmod 9 \neq 0\right],\ k_i = 0,\ b_i = 3 \\ \text{elseif } (k + k_i) \bmod 9 = 0 \text{ and } (k - k_i) \bmod 8 \neq 0,\ k_i = -k_i \end{cases}$$

Example 4.6. If we select conditions (144-64|64-9), window size $w = 5$ and bases $\{2,3\}$, then lines 3 to J+5 of Algorithm 4.3 can be replaced with the following:

$$\begin{cases} k_i = k \bmod 32,\ b_i = 2 \\ \text{if } k \bmod 2 = 0,\ k_i = 0 \\ \text{elseif } k \bmod 3 = 0 \text{ and } \sim\left[(k - k_i) \bmod 64 = 0 \text{ and } k \bmod 9 \neq 0\right],\ k_i = 0,\ b_i = 3 \\ \text{elseif } k - (k \bmod 16) = 0 \bmod 9 \text{ and } (k - k_i) \bmod 64 \neq 0,\ k_i = k \bmod 16 \end{cases}$$

Modified algorithms above are obtained by removing redundancy in the evaluations and rearranging conditional statements once design parameters are fixed. We remark that these algorithms are equivalent to Algorithm 4.3 and yield the same output for a given scalar when using the same design parameters. As consequence, we observe that the refined multibase methodology described in this section can achieve (conjecturally) close to optimal performance with highly compact algorithms.

4.4.1.2. Refined Multibase Chains with Bases {2,3,5}

A methodology similar to the one described in §4.4.1.1 can be applied to the case $\mathcal{A} = \{2,3,5\}$.

Suggested formats for *CONDITION1* and *2* in this case are provided below.

CONDITION1:

Given a partial scalar k, standard window width w and set of bases $\mathcal{a} = \{a_1, a_2, a_3\} = \{2,3,5\}$, where $a_j \in \mathbb{Z}^+$, we propose the following format for *CONDITION1* in Algorithm 4.3:

$$\begin{aligned}
1: &\quad \text{if } ((k - d_{i,1}) \bmod 2^{w'_{1,1}} \cdot 3^{w'_{2,1}} \cdot 5^{w'_{3,1}} = 0 \text{ and } (k - k_i) \bmod 2^{w+1} \neq 0) \text{ or } \ldots \\
2: &\quad ((k - d_{i,2}) \bmod 2^{w'_{1,2}} \cdot 3^{w'_{2,2}} \cdot 5^{w'_{3,2}} = 0 \text{ and } (k - k_i) \bmod 2^{w+2} \neq 0) \text{ or } \ldots \\
&\vdots \\
C: &\quad ((k - d_{i,C}) \bmod 2^{w'_{1,C}} \cdot 3^{w'_{2,C}} \cdot 5^{w'_{3,C}} = 0 \text{ and } (k - k_i) \bmod 2^{w+C} \neq 0)
\end{aligned} \tag{4.14}$$

where $w'_{j,c} \geq 0$ are integers, $k_i = k \bmod 2^w$, c is the condition number such that $1 \leq c \leq C$ and $d_{i,c} \in D \setminus \{0\} = \{\pm 1, \pm 3, \pm 5, \ldots, \pm(2^{w-1} - 1)\}$.

CONDITION2:

Given a partial scalar k, standard window width w and set of bases $\mathcal{a} = \{a_1, a_2, a_3\} = \{2,3,5\}$, where $a_j \in \mathbb{Z}^+$, we propose the following format for *CONDITION2.1* in Algorithm 4.3:

$$\begin{aligned}
1: &\quad \text{if } (k \bmod 5 \neq 0) \text{ and} \\
1.1: &\quad [\text{if } ((k - d_{i,1}) \bmod 2^{w'_{1,1}} = 0 \text{ and } k \bmod 3^2 \neq 0) \text{ or } \ldots \\
&\vdots \\
1.B: &\quad ((k - d_{i,B}) \bmod 2^{w'_{1,B}} = 0 \text{ and } k \bmod 3^{B+1} \neq 0)] \text{ or } \ldots \\
2: &\quad \text{if } (k \bmod 5 = 0) \text{ and} \\
2.1: &\quad [\text{if } ((k - d_{i,1}) \bmod 2^{w''_{1,1}} = 0 \text{ and } k \bmod 3 \cdot 5 \neq 0) \text{ or } \ldots \\
&\vdots \\
2.C: &\quad ((k - d_{i,C}) \bmod 2^{w''_{1,C}} = 0 \text{ and } k \bmod 3^u \cdot 5^v \neq 0)]
\end{aligned} \tag{4.15}$$

where $w'_{j,b}, w''_{j,c}, u, v \geq 0$ are integers, b and c are the condition numbers such that $1 \leq b \leq B$, $1 \leq c \leq C$, and $d_{i,b}, d_{i,c} \in D \setminus \{0\} = \{\pm 1, \pm 3, \pm 5, \ldots, \pm(2^{w-1} - 1)\}$. Note that the upper section of (4.15) evaluates conditions when sequences are not divisible by 5, whereas the lower section evaluates conditions when sequences are divisible by both 3 and 5.

For *CONDITION2.2*, we propose the following format:

$$\begin{aligned}
1: &\quad \text{if } ((k-d_{i,1}) \bmod 2^{w'_{1,1}} = 0 \text{ and } k \bmod 5^2 \neq 0) \text{ or } \ldots \\
2: &\quad\quad ((k-d_{i,2}) \bmod 2^{w'_{1,2}} = 0 \text{ and } k \bmod 5^3 \neq 0) \text{ or } \ldots \\
&\vdots \\
C: &\quad\quad ((k-d_{i,C}) \bmod 2^{w'_{1,C}} = 0 \text{ and } k \bmod 5^{C+1} \neq 0)
\end{aligned} \tag{4.16}$$

where again $w'_{j,c} \geq 0$ are integers, c is the condition number s.t. $1 \leq c \leq C$, and $d_{i,c} \in D \setminus \{0\} = \{\pm 1, \pm 3, \pm 5, \ldots, \pm(2^{w-1}-1)\}$.

4.5. Performance Comparison

We have carried out extensive tests to evaluate the performance of the multibase algorithms discussed in this chapter when applied on standard, extended Jacobi quartic and Twisted Edwards curves using Jacobian ($\mathcal{J}$), extended Jacobi quartic ($\mathcal{JQ}^e$) and inverted Edwards ($\mathcal{IE}$) coordinates, respectively. We implemented the traditional *w*NAF, *wmb*NAF (Algorithm 4.1) and the refined multibase method (Algorithm 4.3) in Matlab, and ran the algorithms with different window sizes for 1000 160- and 256-bit scalars chosen randomly. In the case of multibase algorithms, we evaluated the methods when using the sets of bases {2,3} and {2,3,5}.

We distinguish *two* cases: scenarios with minimal storage (without precomputations) and scenarios with no memory constraints (with optimal number of precomputations).

To estimate costs for each method, we first counted the required number of point operations per scalar, averaged the results and then calculated the cost using Tables 2.2, 2.3 and 2.4 (costs labeled as "Using *S-M* tradings"), ignoring costs of additions and multiplications by curve parameters for simplification purposes. Also, for scenarios with no memory constraints we included in the overall cost the cost of calculating the precomputed points. For computing these points, we consider two cases (see Chapter 3): points are left in projective coordinates (case 1), and points are converted to affine using *one* inversion (case 2). As observed in Section 3.4.1 (Table 3.9, $L_{ES} = 17$; and Table A.1, $L_{ES} = 27$), n = 160 and 256 bits, case 2 is advantageous using Jacobian coordinates for low and intermediate *I/M* ratios, whereas case 1 is more efficient for high *I/M* values. Thus, the particular *I/M* ratio of an implementation will decide which case is more effective on a standard curve. In the case of $\mathcal{JQ}^e$ and $\mathcal{IE}$ coordinates, we only consider case 1 as this scheme should be largely preferred because of the minimal difference of costs between general and mixed additions (see Section 3.4.2, Table 3.11, $w = 5$; and Table 3.12, $w = 6$). Following the analysis in Section 3.4, for Jacobian coordinates, we use the LM Scheme, case 1 and case 2b, whose costs are given by formulas (3.4) and (3.6), respectively, and for $\mathcal{JQ}^e$ and $\mathcal{IE}$ coordinates we apply the LG Scheme, whose costs are displayed in Table 3.3.

The costs using the different methods are summarized in Tables 4.3 and 4.4 for n = 160 and 256 bits, respectively. We have sped up further the proposed multibase methods by saving some initial computations. This technique is similar to that proposed in [Elm06, Section 4.2.2] plus some additional savings gained with the use of composite operations (i.e., tripling, quintupling).

Note that for Jacobian coordinates we use the efficient doubling-addition (DBLADD) operation instead of traditional addition for all the proposed methods. This operation has also been used to improve the performance of the tree-based approach by Doche et al. [DH08].

As can be seen, in the scenario without precomputations, the new refined multibase chains obtained from Algorithm 4.3 achieve the lowest costs for all curves under analysis and security levels. For instance, our results reduce costs in 3% and 10% in comparison with the tree-based method and NAF, respectively, on both $\mathcal{JQ}^e$ and $\mathcal{J}$ coordinates with $n=160$ bits. On the other hand, the basic multibase NAF using bases {2,3} and {2,3,5} achieves better performance than the original double-base method based on the "Greedy" algorithm [DIM05]. That is in addition to the attractive features of *mb*NAF such as simplicity, memory efficiency and easiness to be analyzed theoretically. The tree-based method achieves slightly lower costs than *mb*NAF for bases {2,3} when using $\mathcal{IE}$ coordinates. However, *mb*NAF with bases {2,3,5} surpasses the performance of this method in all the remaining cases. We remark that the tree-based method also finds double-base chains using division chains, although using a search-based approach that consumes more memory than the basic multibase NAF.

Remarkably, in some scenarios using $\mathcal{J}$, refined multibase chains with bases {2,3,5} and no precomputations surpasses the performance of the fastest NAF-based method using an optimal number of precomputed points. For instance, if n = 160 bits the multibase method is superior always when $1I > 19M$.

For comparison in the scenario with optimal number of precomputations, we include results by Bernstein et al. [BBL+07]. This work uses a double-base method based on the "Greedy" algorithm that has been optimized for the use of precomputations. We can see that both the basic *wmb*NAF and the refined multibase chains offer lower computing costs for all the cases under analysis. Note that in this case the performance gap is due to a combination of superior multibase chains and precomputation schemes, faster point operations (e.g., we use the doubling-addition operation in Jacobian coordinates) and the inclusion of the technique to save initial computations.

A more serious competition is brought by the recent work by Meloni and Hasan [MH09], which proposes the use of DBNS representations in combination with Yao's algorithm. This method, denoted by Yao-DBNS, is not based on division chains and has been shown to be efficient when using DBNS representations obtained with the "Greedy" algorithm. Therefore, it is intended for platforms where memory is not scarce.

If there are no memory restrictions, the refined multibase chains using bases {2,3,5} and Yao-

Table 4.3. Comparison of double-base and triple-base scalar multiplication methods (n = 160 bits; $1S = 0.8M$).

Method	# pts	ExtJQuartic ($\mathcal{JQ}^e$)		# pts	InvEdw ($\mathcal{IE}$)		# pts	Jacobian ($\mathcal{J}$)	
		Precomp	Total		Precomp	Total		Precomp	Total
Refined multibase (this work)	0	0	$1261M$ (2)	0	0	$1351M$ (1)	0	0	$1451M$ (2)
*mb*NAF (this work)	0	0	$1292M$ (2)	0	0	$1380M$ (1)	0	0	$1485M$ (2)
Tree-based double-base, Doche et al. [DH08]	0	0	$1303M$	0	0	$1377M$	0	0	$1493M$
Double-base (Greedy), Dimitrov et al. [DIM05]	0	0	$1328M$	0	0	$1403M$	0	0	$1545M$ †
NAF	0	0	$1394M$	0	0	$1448M$	0	0	$1616M$
Refined multibase (this work)	7	$59.6M$	$1214M$ (2)	7	$62.2M$	$1267M$ (1)	6	$55.4M$	$1427M$ (2)
							6	$1I$+$68.4M$	$1I + 1388M$ (2)
(Frac-)*wmb*NAF (this work)	7	$59.6M$	$1222M$ (2)	7	$62.2M$	$1274M$ (1)	6	$55.4M$	$1432M$ (2)
							6	$1I$+$68.4M$	$1I + 1397M$ (2)
Yao-DBNS (Greedy), Meloni et al. [MH09]	N/A	N/A	$1211M$	N/A	N/A	$1259M$	N/A	N/A	$1475M$
Double-base (Greedy), Bernstein et al. [BBL+07]	7	N/A	$1311M$	7	N/A	$1290M$	7	N/A	$1504M$ †
(Fractional) *w*NAF	7	$59.6M$	$1246M$	7	$62.2M$	$1291M$	6	$55.4M$	$1476M$
							6	$1I$+$68.4M$	$1I + 1432M$

† Without using doubling-addition operation [LM08b].

(1) Bases {2,3}.

(2) Bases {2,3,5}.

Table 4.4. Comparison of double-base and triple-base scalar multiplication methods (n = 256 bits; $1S = 0.8M$).

Method	# pts	ExtJQuartic ($\mathcal{JQ}^e$)		# pts	InvEdw ($\mathcal{IE}$)		# pts	Jacobian ($\mathcal{J}$)	
		Precomp	Total		Precomp	Total		Precomp	Total
Refined multibase (this work)	0	0	$2026M$ [2]	0	0	$2174M$ [1]	0	0	$2330M$ [2]
*mb*NAF (this work)	0	0	$2077M$ [2]	0	0	$2218M$ [1]	0	0	$2387M$ [2]
Tree-based double-base, Doche et al. [DH08]	0	0	$2084M$	0	0	$2202M$	0	0	$2388M$
Double-base (Greedy), Dimitrov et al. [DIM05]	0	0	$2125M$	0	0	$2244M$	0	0	$2472M$ †
NAF	0	0	$2244M$	0	0	$2329M$	0	0	$2601M$
Refined multibase (this work)	8	$69M$	$1925M$ [2]	8	$72M$	$2013M$ [1]	8	$72.2M$	$2277M$ [2]
							8	$1I$+$89.6M$	$1I + 2204M$ [2]
(Frac-)*wmb*NAF (this work)	8	$69M$	$1940M$ [2]	8	$72M$	$2025M$ [1]	8	$72.2M$	$2291M$ [2]
							8	$1I$+$89.6M$	$1I + 2219M$ [2]
Yao-DBNS (Greedy), Meloni et al. [MH09]	N/A	N/A	$1911M$	N/A	N/A	$1993M$	N/A	N/A	$2316M$
Double-base (Greedy), Bernstein et al. [BBL+07]	8	N/A	$2071M$	8	N/A	$2041M$	7	N/A	$2379M$ †
(Fractional) *w*NAF	8	$69M$	$1954M$	8	$72M$	$2023M$	8	$72.2M$	$2326M$
							8	$1I$+$89.6M$	$1I + 2235M$

† Without using doubling-addition operation [LM08b].

(1) Bases {2,3}.

(2) Bases {2,3,5}.

DBNS achieve very close performance for all cases and security levels under analysis. The gap when using $\mathcal{JQ}^e$ and $\mathcal{IE}$ coordinates is between ~0%-1% in favor of Yao-DBNS. Given the small theoretical gap and because factors such as cache performance and operation cost variations influence computing time in practice, both methods are expected to achieve equivalent performance for all practical purposes. When using $\mathcal{J}$ coordinates the refined multibase chains remain faster than Yao-DBNS with an advantage between 2%-3%.

Comparison with High-Speed Curves using Radix-2 Methods

Recently, new special curve forms with very efficient group arithmetic have been proposed. These curves achieve high performance in part because they have very efficient doubling and addition formulas. Among them, Twisted Edwards curves (2.12) with parameter $a=-1$ using mixed $\mathcal{E}/\mathcal{E}^e$ coordinates seem to currently offer the best operation count over prime fields [HWC+09]. Unfortunately, there are no known efficient formulas for tripling and quintupling and, hence, these curves cannot benefit from multibase methods.

A performance comparison with the best results from this chapter is relevant. Table 4.5 shows the results using NAF and *w*NAF for Twisted Edwards with mixed $\mathcal{E}/\mathcal{E}^e$ coordinates, refined multibase chains with bases {2,3} for Twisted Edwards using $\mathcal{IE}$ coordinates and refined multibase chains with bases {2,3,5} for extended Jacobi quartics using $\mathcal{JQ}^e$ coordinates. Since curve settings using $\mathcal{JQ}^e$ and $\mathcal{IE}$ involve formulas with multiplications by curve parameters and $\mathcal{E}/\mathcal{E}^e$ coordinates do not, in this case we consider *three* scenarios: $1D=0.1M$, $1D=0.5M$ and $1D=1M$. Operation costs are taken from Tables 2.3 and 2.4 and cost of precomputation is not included for simplification purposes.

It can be seen that the fastest curve using refined multibase chains (ExtJQuartic, $\mathcal{JQ}^e$) with no precomputations outperforms the best performer using radix-2 methods (TEdwards, $\mathcal{E}/\mathcal{E}^e$) if curve parameters do not introduce a significant overhead. For other cases, $\mathcal{E}/\mathcal{E}^e$ using (*w*)NAF is

Table 4.5. Comparison of lowest costs using multibase and radix-2 methods for scalar multiplication, n = 160 bits (cost of precomputation is not included).

Curve	Method	Cost (0 pts)			Cost (7 pts)		
		$1D=0.1M$	$1D=0.5M$	$1D=1M$	$1D=0.1M$	$1D=0.5M$	$1D=1M$
ExtJQuartic, $d=1$, $\mathcal{JQ}^e$	refined (2,3,5)	**1281*M***	**1346*M***	1428*M*	1186*M*	1254*M*	1339*M*
TEdwards, $a=1$, $\mathcal{IE}$	refined (2,3)	1372*M*	1444*M*	1534*M*	1233*M*	1303*M*	1390*M*
TEdwards, $a=-1$, $\mathcal{E}/\mathcal{E}^e$	(*w*)NAF	1353*M*	1353*M*	**1353*M***	**1181*M***	**1181*M***	**1181*M***

the clear winner given the significant overhead introduced by extra multiplications by constants and/or the reduced gain margin obtained with the use of multibases.

In conclusion, if curve parameters are suitably chosen then curves using multibase methods (which otherwise would be slower) may become competitive and even faster than the fastest known curves using radix-2 in memory-constrained devices. For other applications with no memory constraints, it is suggested the use of the fastest curves using (Frac-)*w*NAF.

4.6. Other Applications, Variants and Challenges

In this chapter, we have argued that an analysis based on the operation cost per bit should allow one to find the *optimal* multibase chain(s) for a given scalar. We showed that constraining that analysis to a "portion" of the chain at a time still enables efficient performance. Nevertheless, there are many unexplored possibilities that arise from this new approach. In particular, we have used Algorithm 4.3 for evaluating the different sequences using the operation cost per bit. However, the same methodology can be applied to different variants of this algorithm that could achieve better performance in settings with different relative operation costs.

As stated before, provided formats for *CONDITION1* and *2* evaluate sequences up to the next addition only. However, expanding the "range" of testing could improve performance further. The study of the potential improvement is left as future work.

Also, as discussed in Section 4.4, Algorithm 4.3 employs an *on-line* congruency testing approach to select the division sequences. An alternative approach would involve on-the-fly calculation and comparison of the costs per bit of the possible sequences. This *on-the-fly* approach would lead to alternative algorithms different to the ones proposed here.

Remarkably, other methods in the literature can take advantage of the proposed method. For instance, the tree-based approach by [DH08] can be optimized by employing the operation cost per bit to select division sequences instead of simply selecting the sequences that reach the lowest values. This could potentially allow one to select only *one node* (and to avoid keeping *B* nodes each time, saving memory). In this case, the method would take the form of the *on-the-fly* approach described above. Similar ideas apply to the case of multiple scalar multiplication [DKS09].

In the proposed algorithms, the congruency testing is performed after every performed (doubling, tripling, quintupling) operation (see Algorithm 4.3). In this case the number of iterations for conversion is determined by the total number of doublings, triplings and quintuplings. A simplified variant with faster conversion to multibase would involve the full execution of the chosen sequence until another addition is required, which reduces dramatically

the number of iterations required to the number of additions. The impact in the cost of scalar multiplication is left as future work.

Closely following developments for single scalar multiplication, there have appeared recent efforts for speeding up multiple scalar multiplication with the form $kP + lQ$ using double-base chains. See for instance [DKS09] that presents the analogous of the original tree-based approach [DH08], or [ADI10]. All these works employ division chains and can be improved by exploiting the methodology based on the operation cost per bit exposed in this chapter. The different variants discussed in this subsection could also be adapted to this case.

A note on recent work in the literature:

Very recently and working on top of our techniques published in PKC2009 [LG09], Walter [Wal11] also proposed the use of the cost per bit to derive multibase algorithms based on division chains. Although his methodology is based on a slightly more elaborated cost function, results are expected to be similar to the ones obtained with the methodology in Section 4.4.1.1. Algorithms in [Wal11] are similar (with some variations) to the ones proposed in PKC2009 and revisited here. Although [Wal11] presented slightly better results, we implemented and tested the modified algorithms under the same conditions in which all our algorithms were tested and they achieved equivalent or slightly lower performance than our results. Walter proposed to simplify algorithms to obtain much more compact versions. Following these suggestions, we derived compact versions for our algorithms in the subsection "Highly Compact Multibase Algorithms, Bases {2,3}", pp. 92.

4.6.1. Open Challenges

It has been shown in this chapter that the use of double- and multi-base representations enables faster scalar multiplication in terms of field multiplications and squarings. However, the conversion step in double-base and multi-base methods is more time consuming than using methods based on radix 2. This may or may not be a limiting factor depending on the characteristics of a particular implementation and the chosen platform.

If scalar conversion to multibase representation is expensive, then it must be performed *off-line*, limiting the applicability of these methods to scenarios in which the same scalar k is reused several times or the conversion can be carried out during an *idle* time (e.g., between the first and second phases of the ECDH scheme during data transmission). To overcome this restriction, more research is necessary for developing efficient conversion mechanisms for popular

platforms, accompanied by stringent benchmarking strategies (an inefficient implementation with an "optimized" binary-to-multibase conversion function would obviously lead to misleading conclusions).

Another challenge relates to the efficiency gain that these methods provide. Following results from Section 4.5, theoretical estimates indicate that cost reductions for constrained and non-constrained devices are in the ranges 7%-10% and 1%-3% in comparison with (w)NAF, respectively. The gain margin is expected to reduce further on certain platforms or even vanish in the case of non-constrained devices when considerations such as cache performance are taken into account; see illustrative test results on x86-64 processors in Section 5.6.4, subsection "Timings using Multibase Methods". These observations are confirmed by recent results in the literature that achieve very close theoretical performance at the cost of highly expensive conversion steps [SEI11]. Moreover, for non-constrained devices there are curves that offer higher performance using classical radix 2 methods (see Table 4.5). In conclusion, implementers would probably prefer the adoption of multibase strategies when conversion (if expensive) can be performed *off-line* and the platform is a *memory-constrained device*, for which the cost reductions are non-negligible in comparison with radix-2 methods.

A more somber horizon is envisioned for multiple scalar multiplication methods using double bases in the literature. A popular application of the operation $uP+vQ$ is signature verification (as needed for ECDSA; see §2.2.3). However, in this case integers u and v are calculated *on-line* as part of the verification process and, hence, conversion time from binary to double-base must be included in the computing cost. This reduces drastically (or completely eliminates) the possible gain obtained with these methods in multiple scalar multiplication.

4.7. Conclusions

This chapter discussed the efficient design of scalar multiplication algorithms based on double and multibase chains.

In §4.1, we categorized and analyzed the most relevant methods using double-base and multi-base representations in the literature, highlighting advantages and disadvantages. Then in §4.2 we formally described the original (width-w) multibase NAF method, presenting the theoretical analysis of the different variants using Markov chains. In §4.3, we applied the fractional window recoding to multibase NAF. The revised method allows any number of precomputations, enabling lower costs and/or better coupling to memory-constrained environments.

In §4.4, we introduced a novel methodology based on the analysis of *point operation cost per bit* to design flexible algorithms able to find more efficient multibase chains. This approach was

implicitly used in Longa and Gebotys [LG09] to derive *refined mbNAF chains*, although an explicit description of the algorithm derivation was missing. We have filled the gap in this chapter. Intuitively, given unlimited resources this approach is expected to lead to *optimal* multibase chains. We demonstrated that very compact algorithms are still able to achieve high performance. We derived algorithms for the case of bases {2,3} and {2,3,5}, and analyzed the performance gain with the increase in the complexity of the multibase evaluation. For illustration purposes, we focused the analysis on *three* scenarios: standard curves using Jacobian coordinates, extended Jacobi quartics using extended Jacobi quartics coordinates and Twisted Edwards curves using inverted Edwards coordinates.

In §4.5 we carried out a detailed comparison of the studied methods with the best approaches in the literature. For further cost improvement, we applied the best precomputation method developed in Chapter 3 for each scenario. After extensive comparisons with the most efficient methods in the literature, we concluded that the *refined multibase chains* achieve the highest performance on all scenarios with no precomputations, introducing cost reductions in the range 7%-10% in comparison with NAF. For the case of optimal use of precomputations, we show that the proposed algorithms are among the fastest ones, achieving practically equivalent performance to recent methods such as Yao-DBNS [MH09]. In this case, the theoretical cost reductions are in the range 1%-3% in comparison with (Frac)-*w*NAF.

Finally, in §4.6 we discussed many potential possibilities for the multibase approach based on the analysis of the *operation cost per bit*. We detailed how this tool could potentially lead to different variants of the proposed multibase algorithms and how it could even improve existent methods in the literature. Other possible applications such as multiple scalar multiplications were also covered, as well as a discussion of open problems that challenge the practicality of double-base and multi-base methods in real applications. In conclusion, we suggested the use of multibases for memory-constrained devices when the conversion step (if expensive) can be performed off-line. When precomputations are allowed, the gain may be negligible and faster curves without exploiting multibases are available.

Chapter 5

Efficient Techniques for Implementing Elliptic Curves in Software

In this chapter, we analyze and present experimental data evaluating the efficiency of several techniques for speeding up the computation of elliptic curve point multiplication on emerging x86-64 processor architectures. Our approach is based on a careful optimization of elliptic curve operations at all arithmetic layers in combination with techniques from computer architecture. Our contributions can be summarized as follows:

- We analyze the efficient combination of two well-known techniques: elimination of conditional branches and incomplete reduction (IR), to achieve high-speed field arithmetic over $\mathbb{F}_p$. Specifically, we apply these techniques to the optimization of field arithmetic modulo a pseudo-Mersenne prime.
- We study the impact of *true* data dependencies on elliptic curve operations. Moreover, to reduce the number of pipeline stalls, memory reads/writes and function calls in the execution of field and point arithmetic operations, we propose *three* generic techniques: field arithmetic scheduling, merging of point operations and merging of field operations.
- The cost of explicit formulas is reduced further by minimizing the number of additions/

subtractions and small constants and maximizing the use of operations exploiting IR.

- We study the extension of all the previous techniques to field arithmetic over $\mathbb{F}_{p^2}$, which has several applications in cryptography including its use as underlying field by the recently proposed Galbraith-Lin-Scott (GLS) method.
- We explicitly state the improved explicit formulas using IR, with minimal number of operations and reduced number of data dependencies between contiguous field operations for *two* relevant cases: standard curves using $\mathcal{J}$ coordinates and Twisted Edwards curves using mixed homogeneous/extended homogeneous ($\mathcal{E}/\mathcal{E}^e$) coordinates.
- Finally, to illustrate the significant savings obtained by combining all the previous techniques with state-of-the-art ECC algorithms we present high-speed implementations of point multiplication that are up to 34% faster than the best previous results on x86-64 processors. Our software takes into account results from Chapter 3 and includes the best precomputation scheme corresponding to each setting.

Analysis and tests presented in this chapter are carried out and applied on emerging x86-64 processors, which are getting widespread use in notebooks, desktops, workstations and servers. The reader should note, however, that some techniques and analysis are generic and can be extended to other computing devices based on 32-, 16- or 8-bit architectures. Whenever relevant, we briefly discuss the applicability of the techniques under analysis to other architectures.

This chapter is organized as follows. After discussing some relevant previous work and background related to x86-64 processors in §5.1, we describe the techniques for optimizing modular reduction using a pseudo-Mersenne prime, namely incomplete reduction and elimination of conditional branches, in §5.2. Then, in §5.3 we study data dependencies between field operations and analyze some efficient countermeasures when their effect is potentially negative to performance. In §5.4, we describe our optimizations to explicit formulas that enable a reduction in the number of additions and other "small" operations. The extension of the techniques above to quadratic extension fields is presented in §5.5. Our high-speed implementations with and without exploiting the GLS method that illustrate the performance gain obtained with the techniques under analysis are presented in §5.6. Some conclusions are drawn in §5.7.

5.1. Previous Work and the x86-64 Processor Family

Since the 80's and 90's there have appeared an increasing number of studies focused on the optimization of the arithmetic of elliptic curves with application to cryptography. For example, some works have proposed methods using efficient arithmetic representations for scalars [Mor90, Sol00, DIM05, Lon07] and efficiently computable endomorphisms [GLV01, GLS08] to reduce

the number of point operations required for computing scalar multiplication. Other approaches have focused on constructing curve forms with fast group arithmetic and/or improved resilience against certain side-channel analysis (SCA) attacks [Sma01, BJ03b, Edw07], complemented by research studying efficient projective systems and optimized explicit formulas for point operations [CC86, CMO98, HWC+08, LM08b, HWC+09]. Yet another important aspect refers to the efficient implementation of long integer modular arithmetic [Kar95, Mon85, Com90, YSK02]. Given the myriad of possibilities, it is a very difficult task to determine which methods, once combined for the computation of scalar multiplication, are the most efficient ones for a specific platform. Notorious efforts in this direction are the efficient implementations on constrained 8-bit microcontrollers by [GPW+04, UWL+07], on 32-bit embedded devices by [XB01, GAS+05], on Graphical Processing Units (GPUs) by [SG08], on processors based on the Cell Broadband Engine Architecture (CBEA) by [CS09], on 32-bit processors by [BHL+01, Ber06], among others. In this work, we try to cover this analysis for the increasingly popular x86-64-based processors.

Elliptic Curve Scalar Multiplication on x86-64 Processors

Modern CPUs from the notebook, desktop and server classes are decisively adopting the 64-bit x86 instruction set (a.k.a. x86-64) developed by [AMD]. The most relevant features of this new instruction set are the expansion of the general-purpose registers (GPRs) from 32 to 64 bits, the execution of arithmetic and logical operations on 64-bit integers and an increment in the number of GPRs, among other enhancements. In addition, these processors usually exhibit a highly pipelined architecture, improved branch predictors and complex execution stages that offer parallelism at the instruction level. Thus, this increasingly high complexity brings new paradigms to the software and compiler developer.

It seems that the move to 64 bits, with the inclusion of a powerful 64-bit integer multiplier, favors prime fields. Although the analysis becomes complex and processor dependent, our tests on the targeted processors suggest that SSE2 and its extensions [Intel] are apparently not advantageous by themselves for the prime field arithmetic. This is probably due to the lack of carry handling and the fact that SSE2 multipliers can perform vector-multiplication with operands up to 32 bits only [HMS08]. However, this outcome could change in the near future with improved SIMD extensions, such as the upcoming Advanced Vector Extensions (AVX) that supports 256-bit SIMD vector registers.

As consequence, it is still expected that a traditional approach for computing multiplication such as the “schoolbook” method performs better in this case. Methods such as Karatsuba multiplication theoretically reduce the number of integer multiplications but increase the number of other (usually cheaper) operations. Given the high performance of the multiplier on these

platforms, the cost of addition and other similar operations becomes non-negligible. Another aspect from this observation is that it now becomes relevant for the targeted 64-bit platforms the optimization of these usually neglected "small" operations.

Another important feature is the highly pipelined architectures of these processors. For instance, experiments by [Fog2] suggest that Intel Atom, Intel Core 2 Duo and AMD architectures have pipelines with 16, 15 and 12 stages, respectively. Although sophisticated branch prediction techniques exist, it is expected that the "random" nature of cryptographic computations, specifically of modular reduction, causes expensive mispredictions that force the pipeline to flush.

In this work, we analyze the performance of combining incomplete reduction (IR) and the elimination of conditional branches to obtain high-speed field arithmetic for performing operations such as addition, subtraction and multiplication/division by small constants using a very efficient pseudo-Mersenne prime. This effort puts together in an optimal way techniques by [YSK02], which only provided IR algorithms targeting primes of general form, with branchless field arithmetic recently adopted by some cryptographic libraries [mpFq, MIR]. In the process, we present experimental data quantifying the performance improvement obtained by eliminating branches in the field arithmetic.

We also analyze the influence of deeply pipelined architectures in the ECC point multiplication execution. In particular, the increased number of pipeline stages makes (true) data dependencies between instructions in contiguous field operations expensive because these can potentially stall the execution for several clock cycles. These dependencies, also known as read-after-write (RAW), are typically found between several field operations when the result of an operation is required as input by a following operation. In this work, we demonstrate the potentially high cost incurred by these dependencies, which is hardly avoided by compilers and dynamic schedulers in processors, and propose *three* techniques to reduce its effect: field arithmetic scheduling, merging of field operations and merging of point operations.

The techniques above are first applied to modular operations using a prime p, which are used for performing the $\mathbb{F}_p$ arithmetic in ECC over prime fields. However, some of these techniques are generic and can also be extended to different scenarios using other underlying fields. For instance, Galbraith et al. [GLS09] recently proposed a faster way to do ECC that exploits an efficiently computable endomorphism to accelerate the execution of point multiplication over a quadratic extension field (a.k.a. GLS method); see Section 2.2.6. Accordingly, we extend our analysis to $\mathbb{F}_{p^2}$ arithmetic and show that the proposed techniques also lead to significant gains in performance in this case.

Our extensive tests assessing the techniques under analysis cover at least one representative x86-64-based CPU from each processor class: 1.66GHz Intel Atom N450 from the notebook/

netbook class, 2.66GHz Intel Core 2 Duo E6750 from the desktop class, and 2.6GHz AMD Opteron 252 and 3.0GHz AMD Phenom II X4 940 from the server/workstation class.

Finally, to assess their effectiveness for a full point multiplication, the proposed techniques are applied to state-of-the-art implementations using Jacobian ($\mathcal{J}$) and mixed Twisted Edwards homogeneous/extended homogeneous ($\mathcal{E}/\mathcal{E}^e$) coordinates on the targeted processors. Our measurements show that the proposed optimizations (in combination with state-of-the-art point formulas/coordinate systems, precomputation schemes and exponentiation methods) significantly speed up the execution time of point multiplication, surpassing by considerable margins best previous results. For instance, we show that a point multiplication at the 128-bit security level can be computed in only 181000 cycles (in about 60μsec.) on an AMD Phenom II X4 when combining $\mathcal{E}/\mathcal{E}^e$ with GLS. This represents a cost reduction of about 29% over the closest previous result; see Section 5.6.4 for complete details.

5.2. Optimizing Modular Reduction using a Pseudo-Mersenne Prime

In this section, we evaluate the performance gain of *two* techniques, namely incomplete reduction and elimination of conditional branches, and combine them to devise highly efficient field arithmetic with very fast modular reduction for operations such as addition, subtraction and division/multiplication by constants. We also show that incomplete reduction is not exclusive to addition/subtraction and can be easily extended to other operations, and that subtraction does not necessarily benefit from incomplete reduction when p is a suitably chosen pseudo-Mersenne prime. All tests described in this section were performed on our assembly language module implementing the field arithmetic over $\mathbb{F}_p$ and compiled with GCC version 4.4.3.

5.2.1. Incomplete Reduction (IR)

This technique was introduced by Yanik et al. [YSK02] for the case of primes of general form. Given two numbers in the range $[0, p-1]$, it consists of allowing the result of an operation to stay in the range $[0, 2^s - 1]$ instead of executing a complete reduction, where $p < 2^s < 2p - 1$, $s = n \cdot w$, w is the basic wordlength (typically, $w = 8, 16, 32, 64$) and n is the number of words. If the modulus is a pseudo-Mersenne prime of the form $2^m - c$ such that $m = s$ and $c < 2^w$, then the method gets even more advantageous. In the case of addition, for example, the result can be reduced by first discarding the carry bit in the most significant word and then adding the correction value c, which fits in a single w-bit register. Also note that this last addition does not produce an overflow because $2 \times (2^m - c - 1) - (2^m - c) < 2^m$. The procedure is illustrated for the case of modular addition in Algorithm 5.1(b), for which the reduction step described above is

performed in steps 4-8. In contrast, it can be seen in Algorithm 5.1(a) that a complete reduction requires additionally the execution of steps 9-14 that perform a subtraction $r - p$ in case $p \le r < 2^m$, where r is the partial result from step 3.

Yanik et al. [YSK02] also shows that subtraction can benefit from IR when using a prime p of arbitrary form. However, we show in the following that for primes of special form, such as pseudo-Mersenne primes, that is not necessarily the case.

Algorithm 5.1. Modular addition with a pseudo-Mersenne prime

Input: integers $a, b \in [0, p-1]$, $p = 2^m - c$, $m = n \cdot w$, where $n, w, c \in \mathbb{Z}^+$ and $c < 2^w$

Output: (a) $r = a + b \pmod p$; (b) $r = a + b \in [0, 2^m - 1]$

(a) With complete reduction

1: $carry = 0$
2: For i from 0 to $n-1$ do
3: $(carry, r[i]) \leftarrow a[i] + b[i] + carry$
4: If $carry = 1$
5: $carry = 0$
6: $(carry, r[0]) \leftarrow r[0] + c$
7: For i from 1 to $n-1$ do
8: $(carry, r[i]) \leftarrow r[i] + carry$
9: Else
10: $borrow = 0$
11: For i from 1 to $n-1$ do
12: $(borrow, R[i]) \leftarrow r[i] - p[i] - borrow$
13: If $borrow = 0$
14: $r \leftarrow R$
15: Return r

(b) With incomplete reduction

1: $carry = 0$
2: For i from 0 to $n-1$ do
3: $(carry, r[i]) \leftarrow a[i] + b[i] + carry$
4: If $carry = 1$
5: $carry = 0$
6: $(carry, r[0]) \leftarrow r[0] + c$
7: For i from 1 to $n-1$ do
8: $(carry, r[i]) \leftarrow r[i] + carry$
9: Return r

Modular Subtraction:

Let us consider Algorithm 5.2. After step 3 we obtain the completely reduced value $r = a - b$ if $borrow = 0$. If, otherwise, $borrow = 1$ then this bit is discarded and the partial result is given by $r = a - b + 2^m$, where $b > a$. This value is incorrect, because it has the extra addition with 2^m. Step 6 performs the computation $r + p = (a - b + 2^m) + (2^m - c) = a - b - c + 2^{m+1}$, where $2^m < a - b - c + 2^{m+1} < 2^{m+1}$ since $-2^m + c < a - b < 0$. Then, by simply discarding the final carry from this result (i.e., by subtracting 2^m) we obtain the correct, completely reduced result $a - b - c + 2^{m+1} - 2^m = a - b + p$, where $0 < a - b + p < p$. Since Algorithm 5.2 gives the

correct result without evaluating both values of *borrow* after step 3 (similarly to the case of *carry* in Algorithm 5.1(b)), there is no need for incomplete reduction in this case.

Algorithm 5.2. Modular subtraction with a pseudo-Mersenne prime and complete reduction

Input: integers $a, b \in [0, p-1]$, $p = 2^m - c$, $m = n \cdot w$, where $n, w, c \in \mathbb{Z}^+$ and $c < 2^w$
Output: $r = a - b \pmod{p}$

1: $borrow = 0$
2: For i from 0 to $n-1$ do
3: $(borrow, r[i]) \leftarrow a[i] - b[i] - borrow$
4: If $borrow = 1$
5: $carry = 0$
6: For i from 1 to $n-1$ do
7: $(carry, r[i]) \leftarrow r[i] + carry$
8: Return r

Nevertheless, there are other types of "small" operations that may benefit from the use of IR. Next we analyze the cases that are useful to the setting of ECC over prime fields.

Modular Addition with IR, $a + b \in [0, 2^m - 1]$, where $a \in [0, p-1]$ and $b \in [0, 2^m - 1]$:

In this case, after addition we get $0 \le a + b \le 2^{m+1} - c - 2$, where $2^m < 2^{m+1} - c - 2 < 2^{m+1}$ for practical values of *m*. Thus, if there is no final carry the result *r* is incompletely reduced such that $r \in [0, 2^m - 1]$, as wanted. Otherwise, for the case $2^m \le a + b \le 2^{m+1} - c - 2$ we discard the carry and add the correction value c such that $0 < c \le a + b - 2^m + c \le 2^m - 2 < 2^m$ to obtain an incompletely reduced result $r \in [0, 2^m - 1]$. Consequently, Algorithm 5.1(b) also allows adding two terms where one of them can be in incompletely reduced form.

Modular Multiplication by 3 with IR, $3a \in [0, 2^m - 1]$, where $a \in [0, p-1]$:

If this operation is performed by executing $a + a + a$, internally, the first addition $r = a + a$ can be left incompletely reduced using Algorithm 5.1(b). Then, following the previous subsection, we can again use Algorithm 5.1(b) to perform the addition of the incompletely reduced value *r* with the completely reduced operand *a* to obtain the final result $r + a \in [0, 2^m - 1]$.

Modular Division by 2 with IR, $a/2 \in [0, 2^m - 1]$, where $a \in [0, 2^m - 1]$:

This operation is illustrated when using IR by Algorithm 5.3(b). If the value *a* is even, then a division by 2 can be directly applied through steps 5-7, where $(carry, r[i]) \leftarrow (carry, r[i])/2$ represents the concurrent assignments $r[i] \leftarrow \lfloor (carry \cdot 2^{(i+1).w} + r[i])/2 \rfloor$ and $carry \leftarrow r[i] \pmod 2$.

In this case, if $a \in [0, 2^m - 2]$ then the result $r \in [0, 2^{m-1} - 1]$ is completely reduced since $2^{m-1} - 1 << 2^m - c$ for practical values of *m*, such that $c < 2^w$ and $w < m - 1$. If, otherwise, the operand *a* is odd, we first add *p* to *a* in steps 3-4 to obtain an equivalent from the residue class that is even. Then, $2^m - c + 1 < p + a < 2^{m+1} - c - 1$, where the partial result has $m + 1$ bits maximum and is stored in $(carry, r)$. The operation is then completed by dividing by 2 through steps 5-7, where the final result $2^{m-1} - (c-1)/2 < (p+a)/2 < 2^m - (c+1)/2$. Hence, the result is incompletely reduced because $2^m - c \le 2^m - (c+1)/2 \le 2^m - 1$. If the result needs to be completely reduced then, for the case that $(p+a)/2 \in [p, 2^m - \lceil (c+1)/2 \rceil]$, one needs to additionally compute a subtraction with *p* such that $0 \le (p+a)/2 - p < (c-1)/2 < 2^m - c$, as performed in steps 9-12 of Algorithm 5.3(a).

It is also interesting to note that in the case that input *a* is in completely reduced form, i.e., if $a \in [0, p-1]$, after steps 6-7 in Algorithm 5.3(b) we get $2^{m-1} - (c+1)/2 < (p+a)/2 < 2^m - c$, which is in completely reduced form.

Algorithm 5.3. Modular division by 2 with a pseudo-Mersenne prime

Input: integer $a \in [0, 2^m - 1]$, $p = 2^m - c$, $m = n \cdot w$, where $n, w, c \in \mathbb{Z}^+$ and $c < 2^w$

Output: (a) $r = a/2 \pmod p$; (b) $r = a/2 \in [0, 2^m - 1]$

(a) With complete reduction

1: $carry = 0$
2: If a is odd
3: For i from 0 to $n-1$ do
4: $(carry, r[i]) \leftarrow a[i] + p[i] + carry$
5: $(carry, r[n-1]) \leftarrow (carry, r[n-1])/2$
6: For i from $n-2$ to 0 do
7: $(carry, r[i]) \leftarrow (carry, r[i])/2$
8: $borrow = 0$
9: For i from 0 to $n-1$ do
10: $(borrow, R[i]) \leftarrow r[i] - p[i] - borrow$
11: If $borrow = 0$
12: $r \leftarrow R$
13: Return *r*

(b) With incomplete reduction

1: $carry = 0$
2: If a is odd
3: For i from 0 to $n-1$ do
4: $(carry, r[i]) \leftarrow a[i] + p[i] + carry$
5: $(carry, r[n-1]) \leftarrow (carry, r[n-1])/2$
6: For i from $n-2$ to 0 do
7: $(carry, r[i]) \leftarrow (carry, r[i])/2$
8: Return *r*

To evaluate in practice the advantage of using incomplete reduction, we implemented in assembly language both versions with and without IR of each operation discussed in this section. In Table 5.1, we summarize our results on the targeted Intel and AMD processors.

Table 5.1. Cost (in cycles) of modular operations when using incomplete reduction (IR) and complete reduction (CR); $p = 2^{256} - 189$.

Modular Operation	Atom N450			Core 2 Duo E6750			Opteron 252		
	IR	CR	Cost reduction (%)	IR	CR	Cost reduction (%)	IR	CR	Cost reduction (%)
Addition	31	45	31%	20	25	20%	13	20	35%
Multiplication by 2	27	40	33%	19	24	21%	10	17	41%
Multiplication by 3	43	69	38%	28	43	35%	15	23	35%
Division by 2	57	61	7%	20	25	20%	11	18	39%

As can be seen, in our experiments using the pseudo-Mersenne prime $p = 2^{256} - 189$ we obtain significant reductions in cost ranging from 7% to up to 41% when using IR.

It is important to note that, because multiplication and squaring may accept inputs in the range $[0, 2^m - 1]$, an operation using IR can precede any of these two operations. Thus, the reduction process (which is left "incomplete" by the operation using IR) is fully completed by these multiplications or squarings without any additional cost. If care is taken when implementing point operations, virtually all additions and multiplications/divisions by small constants can be implemented with IR because most of them have results that are later required by multiplications or squarings only. See Appendix B1 for details about the scheduling of field operations $\mathbb{F}_p$ suggested for point formulas using $\mathcal{J}$ and $\mathcal{E}/\mathcal{E}^e$ coordinates.

5.2.2. Elimination of Conditional Branches

Conditional branches may be expensive in several modern processors with deep pipelines if the prediction strategy fails in most instances in a particular implementation. Recovering from a mispredicted branch requires the pipeline to flush, wasting several clock cycles that may increase the overall cost significantly. In particular, the reduction portion of modular addition, subtraction and other similar operations is traditionally expressed with a conditional branch. For example, let us consider the evaluation in step 4 of Algorithm 5.1(b) for performing a modular addition with IR. Because $a, b \in [0, p-1]$ and $2^m - p = c$ (again considering $p = 2^m - c$ and $m = s$), where c is a relatively small number such that $2^m \approx p$ for practical estimates, the possible values for *carry* after computing $a + b$ in steps 2-3, where $(a+b) \in [0, 2p-2]$, are (approximately) equally distributed and describe a "random" sequence for all practical purposes. In this scenario, only an average of 50% of the predictions can be correct in the best case. Similar results are expected for conditional branches in other operations (see Algorithms 5.1, 5.2, 5.3).

To avoid the latter effect, it is possible to eliminate conditional branches by using techniques

such as look-up tables or branch predication. In Figure 5.1, we illustrate the replacement of the conditional branch in step 4 of Algorithm 5.1(b) by a predicated `move` instruction (Figure 5.1(a)) and by a look-up table with indexed indirect addressing (Figure 5.1(b)). In both cases, the strategy is to perform an addition with 0 if there is no carry-out (i.e., the reduction step is not required) or an addition with $c = 189$, where $p = 2^{256} - 189$, if there is carry-out and the computation $(a + b - 2^{256}) + 189$ is necessary. On the targeted CPUs, our tests reveal that branch predication performs slightly better in most cases. This conclusion is platform-dependent and, in the case of the targeted processors, may be due to the faster execution of `cmov` in comparison to the memory access required by the look-up table approach.

(a)

```
>  ⋮
> cmovnc %rax,%rcx
> addq %rcx,%r8
> movq %r8,8(%rdx)
> adcq $0,%r9
> movq %r9,16(%rdx)
> adcq $0,%r10
> movq %r10,24(%rdx)
> adcq $0,%r11
> movq %r11,32(%rdx)
> ret
```

(b)

```
>  ⋮
> adcq $0,%rax
> addq (%rcx,%rax,8),%r8
> movq %r8,8(%rdx)
> adcq $0,%r9
> movq %r9,16(%rdx)
> adcq $0,%r10
> movq %r10,24(%rdx)
> adcq $0,%r11
> movq %r11,32(%rdx)
> ret
```

Figure 5.1. Steps 4-9 of Alg. 5.1(b) for executing modular addition using IR, where $p = 2^{256} - 189$. The conditional branch is replaced by (a) `cmov` instruction (initial values `%rax=0`, `%rcx=189`) and (b) look-up table using indexed indirect addressing mode (preset values `%rax=0`, `(%rcx)=0`, `8(%rcx)=189`). Partial addition $a + b$ from step 3 is stored in registers `%r8-r11` and final result is stored in `x(%rdx)`. x86-64 assembly code uses AT&T syntax.

To quantify in practice the difference in performance obtained by implementing modular arithmetic with and without conditional branches, we tested both schemes on the targeted Intel and AMD processors. The results are summarized in Table 5.2. For addition, subtraction and division by 2, we use Algorithms 5.1(a), 5.2 and 5.3(a), respectively. In the case of addition and division by 2 using IR, we use Algorithms 5.1(b) and 5.3(b), respectively. Multiplication by 2 is a variation of the addition operation for which $2a$ is computed as $a + a \pmod p$.

In Table 5.2, the cost reductions obtained by eliminating CBs can be as high as 50%. Remarkably, the greatest performance gains are obtained in the cases of operations exploiting IR. For instance, on Core 2 Duo, an addition using IR reduces its cost in 46% when CBs have been eliminated in comparison to only the 36% reduction obtained by an addition with complete reduction. Thus, elimination of CBs favors more strongly modular arithmetic using IR. This is

Table 5.2. Cost (in cycles) of modular operations without conditional branches (w/o CB) against operations using conditional branches (with CB); $p = 2^{256} - 189$.

Modular Operation	Atom N450			Core 2 Duo E6750			Opteron 252		
	w/o CB	With CB	Cost reduction (%)	w/o CB	With CB	Cost reduction (%)	w/o CB	With CB	Cost reduction (%)
Subtraction	34	37	8%	21	37	43%	16	23	30%
Addition with IR	31	35	11%	20	37	46%	13	21	38%
Addition	45	43	−4.4%	25	39	36%	20	23	13%
Mult. by 2 with IR	27	34	21%	19	38	50%	10	19	47%
Mult. by 2	40	42	5%	24	38	37%	17	20	15%
Div. by 2 with IR	57	66	14%	20	36	44%	11	18	39%
Div. by 2	61	70	13%	25	39	36%	18	27	33%

due to the fact that modular operations exploiting IR allow very compact implementations that are even easier to schedule efficiently when branches are removed. It is also interesting to note that, when comparing Core 2 Duo's and Opteron's performances, gains are higher for the former processor, which has more stages in its pipeline. Roughly speaking, the gain obtained by eliminating (poorly predictable) CBs on these architectures grows proportionally with the number of stages in the pipeline. In contrast, the gains on Intel Atom are significantly smaller since the pipeline execution and *Instruction-Level Parallelism* (ILP) on this in-order processor are much less efficient and, hence, the relative cost of misprediction penalty reduces.

Following the conclusions above, we have implemented ECC point formulas such that the gain obtained by combining IR and the elimination of CBs is maximal. The reader is referred to Appendix B1 for details about the cost of point formulas in terms of field operations when using $\mathcal{J}$ and $\mathcal{E}/\mathcal{E}^e$ coordinates.

Next, we evaluate the cost of point doubling and doubling-addition (using Jacobian coordinates) when their "small" field operations are implemented with complete or incomplete reduction and with or without conditional branches. For the analysis, we use the revised doubling formula (5.2), Section 5.4, and the doubling-addition formula introduced in [Lon07, formula (3.5), Section 3.2]. The results are shown in Table 5.3.

As can be seen, the computing costs of point doubling and doubling-addition on the AMD processor reduce in 12% and 9%, respectively, by combining the elimination of conditional branches with the use of incomplete reduction. Without taking into account precomputation and the final inversion to convert to affine, these reductions represent about 11% of the computing cost of point multiplication. A similar figure is observed for Intel Core 2 Duo in which doubling

Table 5.3. Cost (in cycles) of point operations with Jacobian coordinates when using incomplete reduction (IR) or complete reduction (CR) and with or without conditional branches (CB); $p = 2^{256} - 189$.

Point operation	Atom N450			Core 2 Duo E6750			Opteron 252		
	CR and CBs	CR and no CBs	IR and no CBs	CR and CBs	CR and no CBs	IR and no CBs	CR and CBs	CR and no CBs	IR and no CBs
Doubling (DBL)	3480	3430	3381	1184	1094	1051	910	824	803
Relative reduction (%)	-	1%	3%	-	8%	11%	-	9%	12%
Doubling-addition	8828	8697	8663	2656	2468	2443	2037	1851	1849
Relative reduction (%)	-	1%	2%	-	7%	8%	-	9%	9%
Estimated relative reduction for 256-bit point multiplication (%)	-	1%	3%	-	8%	10%	-	9%	11%

and doubling-addition are reduced by approx. 11% and 8%, respectively. These savings represent a reduction of about 10% in the cost of point multiplication (again, without considering precomputation and the final inversion). In contrast, following previous observations (see Table 5.2) the techniques are less effective on architectures such as Intel Atom, where the ILP is less powerful and branch misprediction penalty is relatively less expensive. In this case, the cost reduction of point multiplication is only about 3%.

We remark that the algorithms discussed in this section combining completely and incompletely reduced numbers are generic and can be applied to different platforms. Also, the gain obtained by eliminating conditional branches is strongly tied to the pipeline length. So in general it is expected to provide a performance improvement on any architecture with high number of pipeline stages such as most AMD and Intel processors.

5.3. Minimizing the Effect of Data Dependencies

In this section, we analyze (true) data dependencies between "close" field operations and propose *three* techniques to minimize their effect in the point multiplication performance.

Definition 5.1. Let *i* and *j* be the computer orders of instructions I_i and I_j in a given program flow. We say that instruction I_j depends on instruction I_i if:

$$[W(I_i) \cap R(I_j)] \cup [R(I_i) \cap W(I_j)] \cup [W(I_i) \cap W(I_j)] \neq \varnothing, \quad (5.1)$$

where $R(I_x)$ is the set of memory locations or registers that are read by I_x and $W(I_x)$ is the set of memory locations or registers written by I_x.

Modern out-of-order processors and compilers deal relatively well with anti-dependencies ($R(I_i) \cap W(I_j)$, i.e., if I_i reads a location later updated by I_j) and output dependencies ($W(I_i) \cap W(I_j)$, i.e., if both I_i and I_j write on the same location) through register renaming. However, true or RAW dependencies ($W(I_i) \cap R(I_j)$, i.e., if I_j reads something written by I_i) cannot be removed in the strict sense of the term and are more dangerous to the performance of architectures exploiting ILP.

Corollary 5.1. Let I_i and I_j be write and read instructions, respectively, holding true data dependence, i.e., $W(I_i) \cap R(I_j) \neq \varnothing$, where $i < j$ and I_i and I_j are scheduled to be executed at the i^{th} and j^{th} cycle, respectively, in a non-superscalar pipelined architecture. Then, if $\rho = j - i < \delta_{write}$ the pipeline is to be stalled for at least $(\delta_{write} - \rho)$ cycles, where δ_{write} specifies the number of cycles required by the write instruction I_i to complete its pipeline latency after instruction fetching.

Although Corollary 5.1 considers an *ideal* non-superscalar pipeline, it allows us to simplify the analysis on more complex processors. In particular, the value δ_{write}, which strongly depends on the particular characteristics of a given architecture, can be considered for practical purposes roughly equal to the pipeline size. There are *two* approaches to minimize the appearance of pipeline stalls due to RAW dependencies: by instruction scheduling and using data forwarding. Although modern compilers and out-of-order schedulers of processors have powerful capabilities, in our targeted application these still have great limitations to calculate addresses so that rescheduling of instructions between neighboring field operations is possible. On the other hand, hardware techniques such as data forwarding allow a significant reduction in the value δ_{write} by sending back the result of an operation into the decode stage so that this result is immediately available to a coming instruction before the current instruction commit/store the output. Unfortunately, in our application most field operations are not able to efficiently exploit forwarding in case the result is required by the following operation because several consecutive writings to memory are involved in the process.

The problems above are illustrated by the execution of two consecutive field additions in Figure 5.2. For the remainder, given a field operation "$*$", the computation `res`$\leftarrow$`op1*op2` is denoted by `operation(op1,op2,res)`.

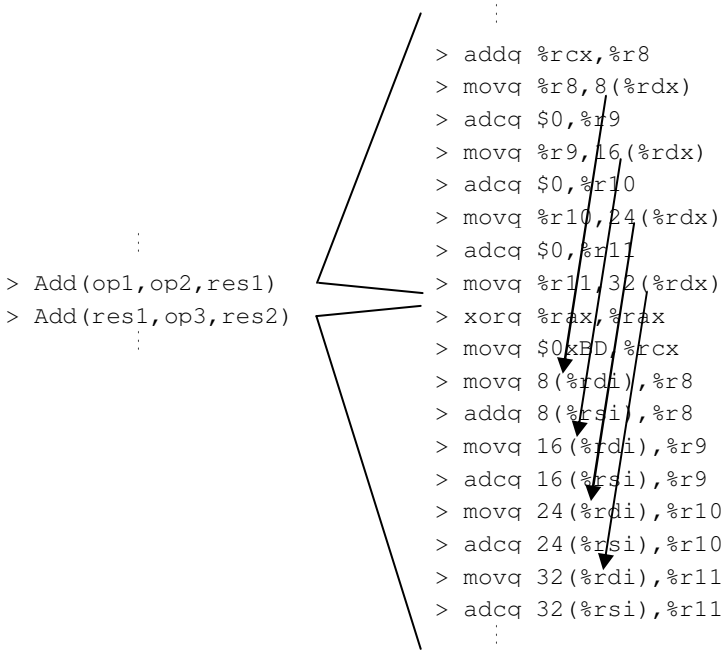


Figure 5.2. Field additions with RAW dependencies on an x86-64 CPU ($p = 2^{256} - 189$). High-level field operations are in the left column and low-level assembly instructions corresponding to each field operation are to the right. Destination *x*(%rdx) of first field addition = source *x*(%rdi) of second field addition. RAW dependencies are indicated by arrows.

As can be seen in Figure 5.2, results stored in memory in the last stage of the first addition are read in the beginning of the second addition. First, if a compiler or out-of-order scheduler is unable of identifying the common addresses then it will not be able of exploiting rescheduling to prevent pipeline stalls due to *inter-field operation* dependencies. Moreover, *four* consecutive writings to memory and then *four* consecutive readings need to be performed because operands are 256-bit long distributed over *four* 64-bit registers. This obviously complicates the extraction of any benefit from data forwarding. If $\delta_{write} > \rho_x$ for at least one of the dependences *x* indicated by arrows then the pipeline is expected to stall for at least $(\delta_{write} - \rho_x)$ cycles. Thus, for the writing/reading sequence in Figure 5.2, the pipeline is roughly stalled by $\max(\delta_{write} - \rho_x)$ for $0 \le x < 4$.

Definition 5.2. Two field operations $OP_i(op_m, op_n, res_p)$ and $OP_j(op_r, op_s, res_t)$ are said to be *data dependent at the field arithmetic level* if $i < j$ and $res_p = op_r$ or $res_p = op_s$, where OP_i and OP_j denote the field operations performed at positions i^{th} and j^{th} during a program execution, and *op* and *res* are registers holding the inputs and result, respectively. Then, this is called a *contiguous* data dependence in the field arithmetic if $j - i = 1$, i.e., OP_i and OP_j are consecutive in the execution sequence. When understood in the context we refer to these

dependencies happening at the field arithmetic level as simply *contiguous data dependencies* for brevity.

For the applications targeted in this work all field operations follow a similar writing/reading pattern to that one shown in Figure 5.2, and hence, two contiguous, data dependent field operations hold several data dependencies x between their internal write/read instructions. Following Definition 5.2 and Corollary 5.1, contiguous data dependencies pose a problem when $\delta_{write} > \rho_x$ in a given program execution, in which case the pipeline is stalled by roughly $\max(\delta_{write} - \rho_x)$ cycles for all dependencies x. Note that at fewer dependent write/read instruction pairs (i.e., at smaller field sizes) the expression $\max(\delta_{write} - \rho_x)$ grows as well as the number of potential stalled cycles. Similarly, at larger computer wordlengths w the value $\max(\delta_{write} - \rho_x)$ is expected to increase, worsening the effect of contiguous data dependencies. For instance, neglecting other architectural factors and assuming a fixed pipeline length, these dependencies are expected to affect performance more dramatically in 64-bit architectures in comparison with 32-bit architectures.

Closely following the analysis above, we propose *three* techniques that help to reduce the number of contiguous data dependencies and study several practical scenarios in which this would allow us to improve the execution performance of point multiplication. As a side effect our techniques also reduce the number of function calls and memory accesses. The reader should note that these additional benefits are processor-independent.

5.3.1. Field Arithmetic Scheduling

A straightforward solution to eliminate contiguous data dependencies is to perform a careful scheduling of field operations inside point formulas in such a way that data-dependent field operations are not contiguous. For all practical purposes, we can consider that any field operation has an executing latency δ_{ins} that is longer than the latency of a write instruction, i.e., $\delta_{ins} > \delta_{write}$. Hence, by inserting any "independent" field operation between two consecutive operations holding contiguous data dependence we guarantee that the new relative positions $\rho_{new,x}$ of the data-dependent instructions accomplishes $\rho_{new,x} = \rho_x + \delta_{ins} > \delta_{write}$ for all data dependencies x, where ρ_x denotes the original relative positions between data-dependent write/read instructions.

We have tested several field operation "arrangements" to observe the latter behavior on different processors. We detail here a few of our experiments with field multiplication on an Intel Core 2 Duo. For example, let us consider the field multiplication sequences given in Table 5.4. As can be seen, **Sequence 1** involves a series of "ideal" data-independent field multiplications, where the output of a given operation is not an input to the immediately following operation. In

this case, the execution reaches its maximal performance with an average of 110 cycles per multiplication because for any pair of data-dependent multiplications we have $\rho_x >> \delta_{write}$. In contrast, **Sequence 2** is highly dependent because each output is required as input in the following operation. In this case, $\delta_{write} > \rho_x$ for at least one dependence *x*. This is the worst-case scenario with an average of 128 cycles per multiplication, which is about 14% less efficient than the "ideal" case. We have also studied other possible arrangements such as **Sequence 3**, in which operands of **Sequence 2** have been reordered. This slightly amortizes the impact of contiguous data dependencies because ρ_x is increased, improving the performance to 125 cycles/mult.

Table 5.4. Various sequences of field operations with different levels of contiguous data dependence.

Sequence 1	Sequence 2	Sequence 3
`> Mult(op1,op2,res1)`	`> Mult(op1,op2,res1)`	`> Mult(op2,op1,res1)`
`> Mult(op3,op4,res2)`	`> Mult(res1,op3,res2)`	`> Mult(op3,res1,res2)`
`> Mult(res1,op5,res3)`	`> Mult(res2,op4,res3)`	`> Mult(op4,res2,res3)`
`> Mult(res2,op6,res4)`	`> Mult(res3,op5,res4)`	`> Mult(op5,res3,res4)`

Similarly, we have also tested the effect of contiguous data dependencies on other field operations. In Table 5.5, we summarize the most representative field operation "arrangements" and their costs. As can be seen, the reductions in cost obtained by switching from an execution with strong contiguous data dependence (worst-case scenario with **Sequence 2**) to an execution

Table 5.5. Average cost (in cycles) of modular operations using best-case (no contiguous data dependencies, **Sequence 1**) and worst-case (strong contiguous data dependence, **Sequence 2**) "arrangements" ($p = 2^{256} - 189$, on a 2.66GHz Intel Core 2 Duo E6750).

Modular Operation	Core 2 Duo E6750		
	Sequence 1	Sequence 2	Cost reduction (%)
Subtraction	21	23	9%
Addition with IR	20	24	17%
Multiplication by 2 with IR	19	23	17%
Multiplication by 3 with IR	28	34	18%
Division by 2 with IR	20	30	33%
Squaring	101	113	11%
Multiplication	110	128	14%

with no contiguous data dependencies (best-case scenario with **Sequence 1**) range from approximately 9% to up to 33% on an Intel Core 2 Duo. Similar results were observed for the targeted AMD Opteron and Phenom II processors, where the high performance of their architectures significantly reduce relative positions ρ_x between their data-dependent write/read instructions, increasing the value $\max(\delta_{write} - \rho_x)$. Thus, minimizing contiguous data dependencies is expected to improve the execution of point multiplication on all these processors. In contrast, **Sequence 1** and **Sequence 2** perform similarly on processors such as Intel Atom, in which the much less powerful architecture tends to increase values ρ_x such that $\delta_{write} < \rho_x$ for all dependencies *x*.

5.3.2. Merging Point Operations

This technique complements and increases the gain obtained by scheduling field operations. As expected, in some cases it is not possible to eliminate all contiguous data dependencies in a point formula. A clever way to increase the chances of eliminating more of these dependencies is by "merging" successive point operations into unified functions.

For example, let us consider the following sequence of field operations for computing a point doubling using Jacobian coordinates, $2(X_1 : Y_1 : Z_1) \rightarrow (X_1 : Y_1 : Z_1)$ (`DblSub(b,c,a)` represents the operation $a \leftarrow b - 2c \pmod{p}$; see Section 5.3.3):

`> Sqr(Z1,t3)`	`> Mult(X1,t2,t4)`	`> Sqr(t1,t2)`
`> Sqr(Y1,t2)`	`> Mult(t1,t0,t3)`	`> DblSub(t2,t4,X1)` •
`> Add(X1,t3,t1)`	`> Sqr(t2,t0)`	`> Sub(t4,X1,t2)` •
`> Sub(X1,t3,t3)`	`> Div2(t3,t1)`	`> Mult(t1,t2,t4)` •
`> Mult3(t3,t0)` •	`> Mult(Y1,Z1,Z1)`	`> Sub(t4,t0,Y1)` •

In total, there are *five* contiguous data dependencies between field operations (denoted by "•") in the sequence above. Note that the last stage accounts for most dependencies, which are very difficult to eliminate. However, if another point doubling follows, one could merge both successive operations and be able to reduce the number of contiguous data-dependent operations. Consider, for example, the following arrangement of *two* consecutive doublings:

`> Sqr(Z1,t3)`	`> Mult(t1,t0,t3)`	`> DblSub(t2,t4,X1)`	**`> Mult3(t3,t1)`**
`> Sqr(Y1,t2)`	`> Sqr(t2,t0)`	`> Sub(t4,X1,t2)` •	**`> Sqr(Y1,t2)`**
`> Add(X1,t3,t1)`	`> Div2(t3,t1)`	**`> Add(X1,t3,t5)`**	**`> Mult(t1,t5,t3)`**
`> Sub(X1,t3,t3)`	`> Mult(Y1,Z1,Z1)`	`> Mult(t1,t2,t4)`	**`> Mult(t2,X1,t4)`**
`> Mult3(t3,t0)` •	`> Sqr(t1,t2)`	**`> Sub(X1,t3,t3)`**	**`> Div2(t3,t1)`**
`> Mult(X1,t2,t4)`	**`> Sqr(Z1,t3)`**	`> Sub(t4,t0,Y1)`	**`> ...`**

As can be seen, the sequence above (instructions from the second doubling are in **bold**) allows us to further reduce the number of dependencies from *five* to only *two*.

In ECC implementations, it appears natural to merge successive doubling operations or a doubling and an addition. Efficient elliptic curve point multiplications kP use NAF in combination with some windowing strategy to recode the scalar k (see Section 2.2.4.3). For instance, wNAF guarantees at least w successive doublings between point additions. Also, one could exploit the efficient doubling-addition operation by [Lon07] for Jacobian coordinates or the combined (dedicated) doubling-(dedicated) addition by [HWC+08] for mixed Twisted Edwards homogeneous/extended homogeneous coordinates (see Table 2.4). Hence, an efficient solution for these systems is to merge $(w-1)$ consecutive doublings (for an optimal choice of w) in a separate function and merge each addition with the precedent doubling in another function. On the other hand, if an efficient doubling-addition formula is not available for certain setting, then it is suggested to merge w consecutive doublings in one function and have the addition in a separate function. Note that for different coordinate systems/curve forms/point multiplication methods the optimal merging strategy may vary or include different operations.

Remarkably, a side-effect of this technique is that the number of function calls to point formulas is also reduced.

5.3.3. Merging Field Operations

This technique consists in merging various field operations with common operands to implement them in a joint function. There are *two* scenarios where this approach becomes attractive:

- The result of a field operation is required as input by a following operation: merging reduces the number of memory reads/writes and eliminates directly potential contiguous data dependencies.

- Operands are required by more than one field operation: merging reduces the number of memory reads/writes.

We remark that the feasibility of merging certain field operations strictly depends on the chosen platform and the number of general purpose registers available to the programmer/ compiler. Also, before deciding on a merging option implementers should analyze and test the increase in the code size and how this affects the performance of the cache for example. Accordingly, in the setting of ECC over prime fields, multiplication and squaring are not recommended to be merged with other operations if multiple functions containing these operations are necessary. The code increase could potentially affect the cache performance.

Example 5.1. Taking into account the considerations above, the following merged field operations can be advantageous on x86-64-based processors using $\mathcal{J}$ and $\mathcal{E}/\mathcal{E}^e$ coordinates: $a-2b \pmod p$, $a+a+a \pmod p$, and the merging of $a-b \pmod p$ and $(a-b)-2c \pmod p$.

We remark that the list in the example above is not exhaustive. Different platforms with more registers may enable a much wider range of merging options. Also, other possibilities for merging could be available for different coordinate systems and/or underlying fields (for instance, see Section 5.5.2 for the merging options suggested for ECC implementations over quadratic extension fields).

To illustrate the impact of scheduling field operations, merging point operations and merging field operations, we show in Table 5.6 the cost of point doubling using Jacobian coordinates when using these techniques in comparison with a naïve implementation with a high number of dependencies. As can be seen, by reducing the number of dependencies from *ten* to about *one* per doubling, minimizing function calls and reducing the number of memory reads/writes, we are able to reduce the cost of a doubling by 12% and 8% on Intel Core 2 Duo and AMD Opteron processors, respectively. It is also important to note that on a processor such as AMD Opteron, which has a smaller pipeline and consequently less lost due to contiguous data dependencies (smaller δ_{write} with roughly the same values ρ_x as Intel Core 2 Duo), the estimated gain obtained with these techniques in the point multiplication is lower (5%) in comparison with the Intel processor (9%). Finally, following our analysis in previous sections, Intel Atom only obtains a very small improvement in this case because contiguous data dependencies do not affect the execution performance significantly (see Section 5.3.1).

Table 5.6. Cost (in cycles) of point doubling using Jacobian coordinates with different number of contiguous data dependencies and the corresponding reduction in the cost of point multiplication. "Unscheduled" refers to implementations with a high number of dependencies (here, 10 dependencies per doubling). "Scheduled and merged" refers to implementations optimized through the scheduling of field operations, merging of point operations and merging of field operations (here, 1.25 dependencies per doubling); $p = 2^{256} - 189$.

Point Operation	Atom N450		Core 2 Duo E6750		Opteron 252	
	Unscheduled	Scheduled and merged	Unscheduled	Scheduled and merged	Unscheduled	Scheduled and merged
Doubling	3390	3332	1115	979	786	726
Relative reduction (%)	-	2%	-	12%	-	8%
Estimated reduction for 256-bit point mult. (%)	-	1%	-	9%	-	5%

The reader is referred to Appendix B1 for the explicit formulas optimized by scheduling or merging field operations and merging point operations for the case of $\mathcal{J}$ and $\mathcal{E}/\mathcal{E}^e$.

5.4. Minimizing the Cost of Point Operations

Several explicit formulas in the literature can be further optimized with the insertion of divisions by 2 by means of the equivalence relation (2.9) of projective coordinates. This trick helps to eliminate constants or reduce their value, which minimizes the need of multiple additions.

Let us illustrate this technique with point doubling. Consider, for example, the doubling formula using Jacobian coordinates in pp. 90-91 of [HMV04] that has a cost of $4\text{Mul} + 4\text{Sqr} + 1\text{Add} + 4\text{Sub} + 2\text{Mul2} + 1\text{Mul3} + 1\text{Div2}$ [1]. If we fix $\lambda = 2^{-1} \in \mathbb{F}_p^*$ in the projective equivalence relation (2.10) that formula can be modified as follows:

$$X_2 = \alpha^2 - 2\beta, \quad Y_2 = \alpha(\beta - X_2) - Y_1^4, \quad Z_2 = Y_1 Z_1, \tag{5.2}$$

where $\alpha = 3(X_1 + Z_1^2)(X_1 - Z_1^2)/2$ and $\beta = X_1 Y_1^2$. With formula (5.2), the operation count is reduced to $4\text{Mul} + 4\text{Sqr} + 1\text{Add} + 5\text{Sub} + 1\text{Mul3} + 1\text{Div2}$, replacing *two* multiplications by 2 with *one* subtraction. Moreover, because constants are minimized, there are greater chances that more "small" operations are executed using incomplete reduction. In Algorithm 5.4, we show an efficient implementation of point doubling (5.2) with optimal use of incomplete reduction (every addition and multiplication/division by constant precedes a multiplication or squaring), minimized number of contiguous data dependencies between field operations and exploiting the use of merged field operations. This execution costs $4\text{Mul} + 4\text{Sqr} + 1\text{Add}_{\text{IR}} + 3\text{Sub} + 1\text{DblSub} + 1\text{Mul3}_{\text{IR}} + 1\text{Div2}_{\text{IR}}$ (where $operation_{\text{IR}}$ represents an operation using incomplete reduction) and has 5 contiguous data dependencies. In Algorithm 5.4, operators $\oplus$, $\otimes$ and $\oslash$ represent addition, multiplication by constant and division by constant using incomplete reduction, respectively. These operations are computed with Algorithm 5.1(b) for addition and multiplication by 3, and with Algorithm 5.3(b) for division by 2 (see Section 5.2.1 for details).

In certain formulas, another optimization is possible. If $1\text{Mul} - 1\text{Sqr} > 4\text{Add}$ and the values a^2 and b^2 are available, one can compute $a \cdot b$ as $\left[(a+b)^2 - a^2 - b^2\right]/2$. See for example addition and doubling-addition formulas, option 1, of the online database EPAF [Lon08].

We remark that the optimizations above are not limited to 64-bit architectures and that are in general advantageous on any platform whenever division by 2 is approximately as efficient as field addition.

Finally, we observe that in some settings field subtraction is more efficient than addition with complete reduction (see for example Table 5.2, when using a pseudo-Mersenne prime). Thus,

[1] Mul = multiplication, Sqr = squaring, Add = addition, Sub = subtraction, Mul*x* = multiplication by *x*, Div*x* = division by *x*.

Algorithm 5.4. Point doubling using Jacobian coordinates

Input: point $P = (X_1 : Y_1 : Z_1) \in E(\mathbb{F}_p)$

Output: $2P = (X_{out} : Y_{out} : Z_{out}) \in E(\mathbb{F}_p)$

1: $t_4 \leftarrow Z_1^2,\ t_3 \leftarrow Y_1^2$
2: $t_1 \leftarrow X_1 \oplus t_4$ (use Algorithm 5.1(b))
3: $t_4 \leftarrow X_1 - t_4$
4: $t_0 \leftarrow 3 \otimes t_4$ (use Algorithm 5.1(b))
5: $t_5 \leftarrow X_1 \times t_3$
6: $t_4 \leftarrow t_0 \times t_1$
7: $t_0 \leftarrow t_3^2$
8: $t_1 \leftarrow t_4 \oslash 2$ (use Algorithm 5.3(b))
9: $t_3 \leftarrow t_1^2,\ Z_{out} \leftarrow Y_1 \times Z_1$
10: $X_{out} \leftarrow t_3 - 2 \times t_5$
11: $t_3 \leftarrow t_5 - X_{out}$
12: $t_5 \leftarrow t_1 \times t_3$
13: $Y_{out} \leftarrow t_5 - t_0$
14: Return $2P = (X_{out} : Y_{out} : Z_{out})$

whenever possible, one can convert those additions that cannot exploit IR to subtractions. For this case, one applies $\lambda = -1 \in \mathbb{F}_p^*$ to the corresponding formula.

5.5. Optimizations for the Quadratic Extension Field Arithmetic

The techniques and optimizations described so far are not exclusive to the popular $\mathbb{F}_p$ field arithmetic. In fact, the scheduling/merging of field operations and merging of point operations are generic and can be extended to different finite fields with similar benefits and results. In this section, we analyze how the aforementioned techniques can be applied to the arithmetic over a quadratic extension field $\mathbb{F}_{p^2}$. This application has gained sudden importance thanks to the recently proposed GLS method [GLS09], which exploits an efficiently computable homomorphism to speed up the execution of point multiplication over $\mathbb{F}_{p^2}$.

For our study, we consider the highly-optimized assembly module of the field arithmetic over $\mathbb{F}_{p^2}$ written by M. Scott [MIR]. This module exploits the "nice" Mersenne prime $p = 2^{127} - 1$, which allows a very simple reduction step with no conditional branches. Although IR can also be applied to this scenario, in practice we observe that the gain is negligible on the platforms under study. Future work may consider the analysis of this technique on different platforms.

5.5.1. Scheduling of Field Operations

As described in Section 2.2.6.1, each $\mathbb{F}_{p^2}$ operation consists of a few field operations over $\mathbb{F}_p$. Thus, the analysis of data dependencies and scheduling of operations should be performed taking into account this underlying layer. For instance, let us consider the execution of a $\mathbb{F}_{p^2}$ multiplication followed by a subtraction shown in Figure 5.3. Note that multiplication is implemented using Karatsuba with 3 $\mathbb{F}_p$ multiplications and 5 $\mathbb{F}_p$ additions/subtractions.

As can be seen in Figure 5.3, the scheduling of the internal $\mathbb{F}_p$ operations of the $\mathbb{F}_{p^2}$ multiplication has been performed in such a way that contiguous data dependencies are minimal between $\mathbb{F}_p$ operations (there is only *one* dependency between `DblSub` and `Sub` in the last stage of multiplication). A similar analysis can be performed between contiguous higher-layer $\mathbb{F}_{p^2}$ operations. In Figure 5.3, the last $\mathbb{F}_p$ operation of the multiplication and the first $\mathbb{F}_p$ operation of the subtraction hold contiguous data dependence. There are different solutions to eliminate this problem. For example, it can be eliminated by rescheduling the $\mathbb{F}_{p^2}$ subtraction and addition, as shown in Figure 5.4(a). Note that addition does not hold any dependence with the multiplication or subtraction, as required. Alternatively, if internal $\mathbb{F}_p$ field operations of the subtraction in $\mathbb{F}_{p^2}$ are rescheduled, as shown in Figure 5.4(b), the contiguous data dependence is also eliminated.

These strategies can be applied to point formulas to minimize the appearance of such dependencies. The reader is referred to Appendix B2 for details about the scheduling of $\mathbb{F}_{p^2}$ operations suggested for point formulas using $\mathcal{J}$ and $\mathcal{E}/\mathcal{E}^e$ coordinates.

5.5.2. Merging of Point and Field Operations

In the case of the GLS method, merging of point doublings is not as advantageous as in the traditional scenario of ECC over $\mathbb{F}_p$ because most contiguous data dependencies can be eliminated by simply rescheduling field operations inside point formulas using the techniques from the previous subsection (see Appendix B2). Moreover, GLS employs point multiplication techniques such as interleaving, which do not guarantee a long series of consecutive doublings between additions. Nevertheless, it is still advantageous the use of the merged doubling-addition operation (when applicable), which is a recurrent operation in interleaving.

On the other hand, merging field operations is more advantageous in this scenario than over $\mathbb{F}_p$. There are two reasons for this to happen. First, arithmetic over $\mathbb{F}_{p^2}$ works on top of the arithmetic over $\mathbb{F}_p$, which opens new possibilities to merge more $\mathbb{F}_p$ operations. Second, operations are on fields of half size, which means that fewer registers are required for representing field elements and more registers are available for holding intermediate operands.

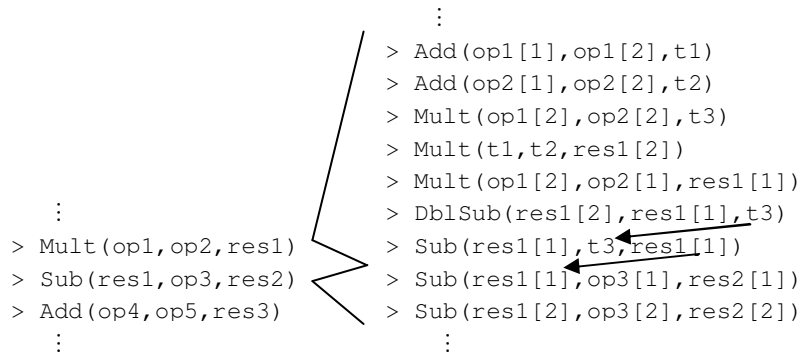


Figure 5.3. $\mathbb{F}_{p^2}$ operations with contiguous data dependencies. High-level $\mathbb{F}_{p^2}$ operations are in the left column and their corresponding low-level $\mathbb{F}_p$ operations are in the right column. $\mathbb{F}_{p^2}$ elements $(a+bi)$ are represented as (`op[1],op[2]`). Dependencies are indicated by arrows.

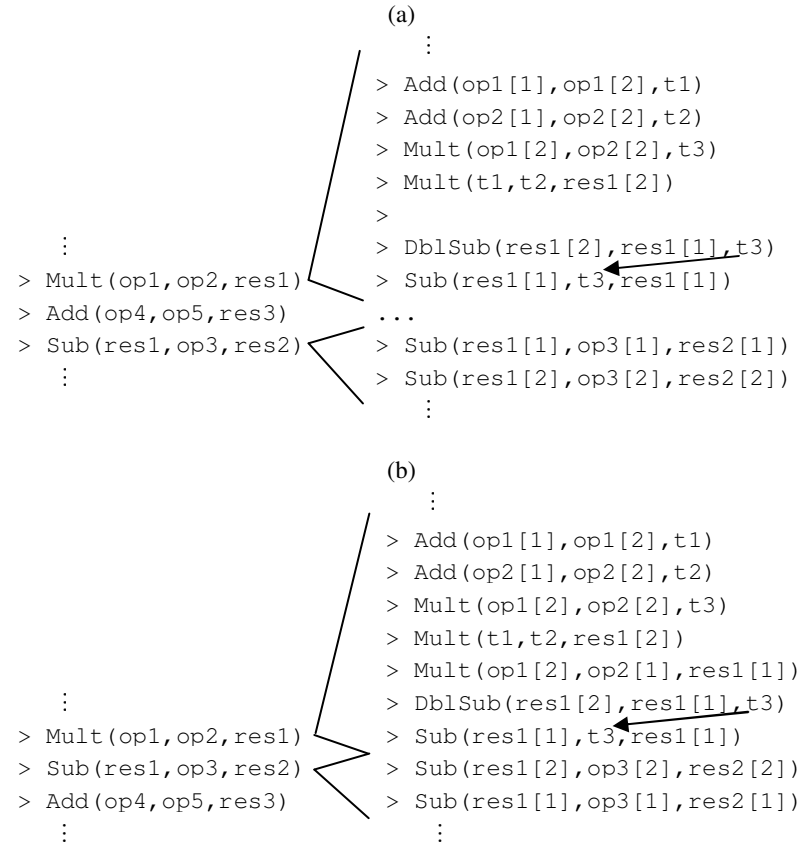


Figure 5.4. (a) Contiguous data dependencies eliminated by scheduling $\mathbb{F}_{p^2}$ field operations; (b) Contiguous data dependencies eliminated by scheduling $\mathbb{F}_p$ field operations.

Example 5.2. The following merged field operations can be advantageous on x86-64-based processors using $\mathcal{J}$ and $\mathcal{E}/\mathcal{E}^e$ coordinates over quadratic extension fields: $a-2b \pmod p$, $(a+a+a)/2 \pmod p$, $a+b-c \pmod p$, the merging of $a+b \pmod p$ and $a-b \pmod p$, the merging of $a-b \pmod p$ and $c-d \pmod p$, and the merging of $a+a \pmod p$ and $a+a+a \pmod p$.

Again, we remark that the list above is not intended to be exhaustive and different merging options could be more advantageous or be available on different platforms with different coordinate systems or underlying fields. The reader is referred to Appendix B2 for the explicit formulas optimized with the proposed techniques for the case of $\mathcal{J}$ and $\mathcal{E}/\mathcal{E}^e$ coordinates using the GLS method.

5.6. Performance Evaluation

In this section, we combine and demonstrate the efficiency of the techniques described in this chapter to accelerate the computation of a full point multiplication. For our implementations, we use the well-known MIRACL library by M. Scott [MIR], which contains an extensive set of cryptographic functions that simplified the development and optimization process of our cryptographic routines. Comparisons focus on implementations of variable-scalar-variable-point elliptic curve point multiplication with approximately 128 bits of security.

5.6.1. Details of the "Traditional" Implementations

Field Arithmetic

As previously described, the field arithmetic over $\mathbb{F}_p$ using the pseudo-Mersenne prime $p = 2^{256} - 189$ was written using x86-64 compatible assembly language and optimized by exploiting incomplete reduction and elimination of conditional branches for modular addition, subtraction and multiplication/division by constants (see Section 5.2). For the case of squaring and multiplication, there are *two* methods that are commonly preferred in the literature for implementation on general purpose processors: schoolbook (or operand scanning method) and Comba [Com90] (or product scanning method) (see Section 5.3 of [EYK09] or Section 2.2.2 of [HMV04]). Both methods require n^2 w-bit multiplications when multiplying two n-digit numbers. However, we choose to implement Comba's method since it requires approx. $3n^2$ w-bit additions, whereas schoolbook requires $4n^2$. Modular reduction for both operations was performed exploiting the fact that $2^{256} \equiv 189$ so $r \equiv (r \,\%\, 2^{256}) + 189(r >> 256)$, where r is the result of integer multiplication or squaring. Our code was aggressively optimized by carefully scheduling instructions to exploit the instruction-level parallelism.

Point Arithmetic

For our implementations, we chose $\mathcal{J}$ and $\mathcal{E}/\mathcal{E}^e$ coordinates and used the execution patterns based on doublings and doubling-additions proposed by [Lon07] and [HWC+08] for $\mathcal{J}$ and $\mathcal{E}/\mathcal{E}^e$, respectively. The costs in terms of multiplications and squarings can be found in Tables 2.2 and 2.4. Note that we use general additions (or general doubling-additions) because inversion is relatively expensive and its inclusion during precomputation cancels any gain using addition with mixed coordinates during the evaluation stage.

This arithmetic layer was optimized through the use of the techniques described in Sections 5.3 and 5.4, namely field arithmetic scheduling, merging of field and point operations and minimization of field operations. Because the maximal performance was found with a window of size 5 for the scalar recoding using *w*NAF (see next subsection), we merged *four* consecutive doublings into a joint function and every addition with the precedent doubling into another function. Please refer to Appendix B1 for complete details about the employed formulas exhibiting minimal number of field operations, different merged field operations and reduced number of contiguous data dependencies.

Point Multiplication and Precomputation

For scalar recoding we use *w*NAF, which offers minimal nonzero density among signed binary representations for a given window width (i.e., for certain number of precomputed points) [Ava05]. In particular, we use Alg. 3.35 of [HMV04] for conversion from integer to *w*NAF representation. Although left-to-right conversion algorithms exist [Ava05], which save memory and allow on-the-fly computation of point multiplication, they are not advantageous on the targeted CPUs. In fact, our tests show that converting the scalar to *w*NAF and then executing the point multiplication achieves higher performance than interleaving conversion and point multiplication. That is because the latter approach "interrupts" the otherwise smooth flow of point multiplication by calling the conversion function at every iteration of the double-and-add algorithm. Our choice is also justified because there are no stringent constraints in terms of memory in the targeted platforms.

For precomputation on $\mathcal{J}$ coordinates, we choose the variant of the LM scheme that does not require inversions, whose cost is given by formula (3.4) (Section 3.2.2). This method achieves the lowest cost for precomputing points, given by $(6L+2)M+(3L+4)S$, where L represents the number of non-trivial points (note that we avoid here the *S*-*M* trading in the first doubling). On $\mathcal{E}/\mathcal{E}^e$, we precompute points in the traditional way using the sequence $P+2P+2P+\ldots+2P$, adding $2P$ with general additions. Because precomputed points are left in projective form no inversion is required and the cost is given by $(8L+4)M+2S$. This involves computing $2P$ as $2\mathcal{A}\rightarrow\mathcal{E}^e$, which costs $5M+2S$ (*one* squaring is saved because $Z_P=1$; *one* extra multiplication

is required to compute T coordinate of $2P$), *one* mixed addition to compute $P+2P$ as $\mathcal{A}+\mathcal{E}^e \rightarrow \mathcal{E}^e$ that costs $7M$ and $(L-1)$ general additions $\mathcal{E}^e+\mathcal{E}^e \rightarrow \mathcal{E}^e$ that cost $8M$ each. For both coordinate systems, we chose a window width w = 5 (i.e., precomputing $\{3P,5P,\ldots,15P\}$, L = 7), which is optimal and slightly better than fractional windows using L = 6 or 8.

5.6.2. Details of the GLS-based Implementations

For this case we make use of the optimized assembly module of the field arithmetic over $\mathbb{F}_{p^2}$ written by M. Scott [MIR], which exploits the Mersenne prime $p=2^{127}-1$ allowing the use of a very simple reduction step with no conditional branches.

For the point arithmetic, we slightly modify formulas for the "traditional" implementations since in this case these require a few extra multiplications with the twisted curve parameter μ (see Section 2.2.6). For example, the (dedicated) addition using extended Twisted Edwards coordinates with cost $8M$ (pp. 332 of [HWC+08]) cannot be used in this case and has to be replaced by a formula that costs $9M$ (also discussed in pp. 332 of [HWC+08] as "$9M$+$1D$"), which is *one* multiplication more expensive ("$1D$" is avoided because parameter a is still set to -1). Accordingly (and also following our discussions in Sections 5.3.1 and 5.5.1), the scheduling of the field arithmetic slightly differs. Moreover, different merging options for the field and point arithmetic are exploited (see Section 5.5.2). The reader is referred to Appendix B2 for complete details about the revised formulas exhibiting minimal number of field operations, different merged operations and reduced number of contiguous data dependencies.

For point multiplication, each of the two scalars k_0 and k_1 in the multiple point multiplication $k_0P+k_1(\lambda P)$ is converted using fractional wNAF [Möl05], and then the evaluation stage is executed using interleaving (see Alg. 3.51 of [HMV04]). Similarly to our experiments with the "traditional" implementations, we remark that the separation of the conversion and evaluation stages yields better performance in the targeted platforms.

For precomputation on $\mathcal{J}$, we use the LM scheme that has minimal cost among methods using only *one* inversion. The cost in this case is given by eq. (3.6). We avoid here the *S-M* trading in the first doubling, so the precomputing cost is $1I+(9L+1)M+(2L+5)S$, where L represents the number of non-trivial points. A fractional window with L = 6 achieves the optimal performance in our case.

Again, on $\mathcal{E}/\mathcal{E}^e$ coordinates we precompute points using general additions in the sequence $P+2P+\ldots+2P$. Precomputed points are better left in projective coordinates, in which case the cost is given by $(9L+4)M+2S$. This cost involves the computation of $2P$ as $2\mathcal{A} \rightarrow \mathcal{E}^e$, which costs $5M+2S$ (*one* squaring is saved because $Z_P=1$; *one* extra multiplication is required to compute T coordinate of $2P$), *one* mixed addition to compute $P+2P$ as $\mathcal{A}+\mathcal{E}^e \rightarrow \mathcal{E}^e$ that costs $8M$ and $(L-1)$ general additions $\mathcal{E}^e+\mathcal{E}^e \rightarrow \mathcal{E}^e$ that cost $9M$ each. In this case, an integral

window of size $w = 5$ (i.e., $L = 7$) achieves optimal performance. As pointed out by [GLS09], precomputing $\{P,[3]\psi(P),[5]\psi(P),\ldots,[2L+1]\psi(P)\}$ can be done on-the-fly at low cost.

5.6.3. The Curves

Next, we detailed the curves used for our implementations. These curves provide approximately 128 bits of security and were found with a modified version of the Schoof's algorithm provided with MIRACL.

- For the implementation on short Weierstrass form over $\mathbb{F}_p$ using $\mathcal{J}$, we chose the curve $E_W : y^2 = x^3 - 3x + B$, where $p = 2^{256} - 189$, $B = 0\times$fd63c3319814da55e88e9328e962 73c483dca6cc84df53ec8d91b1b3e0237064 and $\#E_W(\mathbb{F}_p) = 10r$ where r is the 253-bit prime: 1157920892373161954235709850086879078539455137283671276828741723279050031851 7. The implementation corresponding to this curve is referred to as *jac256189* in the remainder.

- For Twisted Edwards over $\mathbb{F}_p$ using $\mathcal{E}/\mathcal{E}^e$ coordinates, we chose the curve E_{TE}: $-x^2 + y^2 = 1 + 358\,x^2y^2$, where $p = 2^{256} - 189$ and $\#E_{TE}(\mathbb{F}_p) = 4r$ where r is the 255-bit prime: 2894802230932904885589274625217197696338165364456679332971653119013681560794 9. The implementation corresponding to this curve is referred to as *ted256189* in the remainder.

- Let $E_{W-GLS} : y^2 = x^3 - 3x + 44$ be defined over $\mathbb{F}_p$, where $p = 2^{127} - 1$. For the case of Weierstrass form using GLS, we use the quadratic twist $E'_{W-GLS} : y^2 = x^3 - 3\mu x + 44\mu$ of $E_{W-GLS}/\mathbb{F}_{p^2}$, where $\mu = 2 + i \in \mathbb{F}_{p^2}$ is non-square. $\#E'_{W-GLS}(\mathbb{F}_{p^2})$ is the 254-bit prime: 2894802230932904885589274625217197696264992223610339014758410951787459246770 1. The same curve is also used in [GLS09]. Our implementation corresponding to this curve is referred to as *jac1271gls* in the remainder.

- Let $E_{TE-GLS} : -x^2 + y^2 = 1 + 109x^2y^2$ be defined over $\mathbb{F}_p$, where $p = 2^{127} - 1$. For the case of Twisted Edwards using the GLS method, we use the quadratic twist $E'_{TE-GLS} : -\mu x^2 + y^2 = 1 + 109\mu x^2y^2$ of $E_{TE-GLS}/\mathbb{F}_{p^2}$, where $\mu = 2 + i \in \mathbb{F}_{p^2}$ is non-square. In this case, $\#E'_{TE-GLS}(\mathbb{F}_{p^2}) = 4r$ where r is the 252-bit prime: 7237005577332262213973186563042994240709941236554960197665975021634500559269. The implementation corresponding to this curve is referred to as *ted1271gls* in the remainder.

5.6.4. Timings

Here we summarize the timings obtained by the "traditional" implementations labeled as *ted256189* and *jac256189* and the implementations using GLS labeled as *ted1271gls* and *jac1271gls*, when running them on a single core of Intel and AMD processors based on the x86-64 ISA. For verification of each implementation, the results of 10^4 point multiplications with "random" scalars were all validated using MIRACL. Several "random" point multiplications were also verified with Magma.

All the tested programs were compiled with GCC v4.3.4 on the AMD Opteron 252 and with GCC v4.4.3 on the AMD Phenom II X4, Intel Core 2 Duo E6750 and Intel Atom N450 processors. For measuring computing time, we follow [GT07b] and use a method based on cycle counts. To obtain our timings, we ran each implementation 10^5 times with randomly generated scalars, averaged and approximated the results to the nearest 1000 cycles. Table 5.7 summarizes our results, labeled as *ted1271gls*, *jac1271gls*, *ted256189* and *jac256189*. All costs include scalar conversion, the point multiplication computation (precomputation and evaluation stages) and the final normalization step to affine. For comparison purposes, Table 5.7 also includes the cycle counts that we obtained when running the implementations by M. Scott (displayed as *gls1271-ref4* and *gls1271-ref3* [MIR]) on exactly the same platforms. Finally, the last 5 rows of the table detail cycle counts of several state-of-the-art implementations as reported in the literature. However, these referenced results are used only to provide an approximate comparison since the processor platforms are not identical (though they use very similar processors).

As can be seen, our fastest implementation on the targeted platforms is *ted1271gls*, using $\mathcal{E}/\mathcal{E}^e$ with the GLS method. This implementation is about 28% faster than the previous record set by *gls1271-ref4* [GLS08] on a slightly different processor (1.66GHz Intel Core 2 Duo). A more precise comparison, however, would be between measurements on identical processor platforms. In this case, *ted1271gls* is approx. 20%, 29%, 28% and 29% faster than *gls1271-ref4* [MIR] on Atom N450, Core 2 Duo E6750, Opteron 252 and Phenom II X4 940, respectively. Although [MIR] uses inverted Twisted Edwards coordinates ($\mathcal{IE}$), the improvement with the change of coordinates only explains a small fraction of the speed-up. Similarly, in the case of $\mathcal{J}$ combined with GLS, *jac1271gls* is about 30% faster than the record set by *gls1271-ref3* [GLS09] on a 1.66GHZ Intel Core 2 Duo. When comparing cycle counts on identical processor platforms, *jac1271gls* is 23%, 31%, 30% and 34% faster than *gls1271-ref3* [MIR] on Atom N450, Core 2 Duo E6750, Opteron 252 and Phenom II X4 940, respect. Our implementations are also significantly faster than the implementation of Bernstein's *curve25519* by Gaudry and Thomé [GT07b]. For instance, *ted1271gls* is 46% faster than *curve25519* [GT07b] on a 2.66GHz Intel Core 2 Duo.

If the GLS method is not considered, the fastest implementations using $\mathcal{E}/\mathcal{E}^e$ and $\mathcal{J}$

coordinates are *ted256189* and *jac256189*, respectively. In this case, *ted256189* and *jac256189* are 22% and 28% faster than the previously best cycle counts due to Hisil et al. [HWC+09] using also $\mathcal{E}/\mathcal{E}^e$ and $\mathcal{J}$ coordinates, respectively, on a 2.66GHz Intel Core 2 Duo.

It is also interesting to note that the performance boost given by the GLS method strongly depends on the characteristics of a given platform. For instance, *ted1271gls* and *jac1271gls* are about 40% and 45% faster than their "counterparts" over $\mathbb{F}_p$, namely *ted256189* and *jac256189*, respectively, on an Intel Atom N450. On an Intel Core 2 Duo E6750, the differences reduce to 25% and 32% (respect.). And on an AMD Opteron processor, the differences reduce even further to only 9% and 13% (respect.). Thus, it seems that there exists certain correlation between an architecture's "aggressiveness" for scheduling operations/exploiting ILP and the gap between the costs of $\mathbb{F}_p$ and $\mathbb{F}_{p^2}$ operations on x86-64 based processors. In general, the greater such "aggressiveness" the smaller the $\mathbb{F}_p - \mathbb{F}_{p^2}$ gap. And since working on the quadratic extension involves a considerable increase in the number of multiplications and additions, GLS loses its attractiveness if such gap is not large enough on certain platform. For the record, *ted1271gls* achieves the best cycle counts on an AMD Opteron, with an advantage of about 31% over the best previous result in the literature by [GT07b], and on an AMD Phenom II X4, with an advantage of about 29% over the closest result obtained by *gls1271-ref4* [MIR].

For extended benchmark results and comparisons with other previous works on different 64-bit processors, the reader is referred to our online database [Lon10].

Table 5.7. Cost (in cycles) of point multiplication on 64-bit architectures.

Implementation	Coordinates	Field Arithmetic	Atom N450	Core 2 Duo E6750	Opteron 252	Phenom II X4 940
ted1271gls	$\mathcal{E}/\mathcal{E}^e$	$\mathbb{F}_{p^2}$, 127-bit	**588000**	**210000**	**211000**	**181000**
jac1271gls	$\mathcal{J}$	$\mathbb{F}_{p^2}$, 127-bit	644000	228000	238000	188000
ted256189	$\mathcal{E}/\mathcal{E}^e$	$\mathbb{F}_p$, 256-bit	982000	281000	232000	213000
jac256189	$\mathcal{J}$	$\mathbb{F}_p$, 256-bit	1168000	335000	274000	252000
gls1271-ref4 [MIR]	$\mathcal{IE}$	$\mathbb{F}_{p^2}$, 127-bit	732000	295000	295000	255000
gls1271-ref3 [MIR]	$\mathcal{J}$	$\mathbb{F}_{p^2}$, 127-bit	832000	332000	341000	287000
gls1271-ref4 [GLS08]	$\mathcal{IE}$	$\mathbb{F}_{p^2}$, 127-bit	-	293000 (1)	-	-
gls1271-ref3 [GLS09]	$\mathcal{J}$	$\mathbb{F}_{p^2}$, 127-bit	-	326000 (1)	-	-
curve25519 [GT07b]	Montgomery	$\mathbb{F}_p$, 255-bit	-	386000 (2)	307000 (4)	-
Hisil et al. [HWC+09]	$\mathcal{E}/\mathcal{E}^e$	$\mathbb{F}_p$, 256-bit	-	362000 (3)	-	-
Hisil et al. [HWC+09]	$\mathcal{J}$	$\mathbb{F}_p$, 256-bit	-	468000 (3)	-	-

(1) On a 1.66GHz Intel Core 2 Duo. (2) On a 2.66GHz Intel Core 2 Duo E6700.
(3) On a 2.66GHz Intel Core 2 Duo E6550. (4) On a 2.4GHz AMD Opteron 250.

Timings using Multibase Methods

We also implemented the refined multibase algorithm using bases {2,3} and {2,3,5} proposed in Chapter 4 to assess its effectiveness on x86-64 processors. With an optimal number of precomputations (L = 7 points) and using $\mathcal{J}$ coordinates, a 256-bit scalar multiplication runs in approximately 252000 cycles using refined {2,3,5} multibase chains or wNAF on a Phenom II X4, without including the conversion cost. Thus, the small theoretical advantage of the multibase method (see §4.5) vanishes in this case because the inclusion of tripling and quintupling functions that are not used too frequently seems to degrade the cache performance and because radix-2 methods are able to exploit more advantageously additional techniques such as the merging of point operations (see §5.3.2).

For illustrative purposes, in the same implementation above we eliminated the use of precomputations. In this case, the refined {2,3,5} multibase chains and NAF allowed the computation in 261000 and 277000 cycles, respectively, on a Phenom II X4 processor. Thus, on this processor the use of multibases introduces a cost reduction of about 6%.

In all cases above, when conversion to multibase was included in the measurements the total cost of scalar multiplication became more expensive than the cases using (w)NAF.

These results confirm our analysis and recommendations in Section 4.6.1, and justify the use of radix-2 methods in the x86-64-based implementations presented in this chapter.

5.7. Conclusions

In this chapter we have proposed and evaluated different techniques and optimizations to speed up elliptic curve scalar multiplication over prime fields on the increasingly popular x86-64-based processors. We have carefully studied the architecture of these processors and optimized the arithmetic of elliptic curves at the different computational levels accordingly. Extensive tests have been carried out on at least one x86-64 processor representative from the notebook/netbook, desktop and server/workstation processor classes. Whenever relevant, we have also discussed the extension of the analysis and optimizations to other microarchitectures.

After detailing in §5.1 some previous work and the general features of x86-64 processors that are most relevant to this work, we studied the performance boost obtained when combining incomplete reduction and elimination of conditional branches with the use of a highly-efficient pseudo-Mersenne prime in §5.2. We provided explicit algorithms for performing different variants of modular addition, subtraction, multiplication by constant and division by constant with incompletely and completely reduced numbers. Our tests on the targeted platforms reveal cost reductions as high as 9% and 12% in the computation of point doubling and doubling-addition, respectively, when combining the techniques above. Overall, the cost reduction in a

256-bit scalar multiplication was estimated to be up to 11%.

In §5.3, we analyzed the effect of RAW dependencies between contiguous field operations in the performance of scalar multiplication. We demonstrated that by rescheduling or merging field operations and merging point operations the cost of point doubling may be reduced in up to 12% in the targeted processors. This gain is obtained by the compound effect of reducing the number of pipeline stalls, memory reads/writes and function calls. Overall, the cost reduction in a 256-bit scalar multiplication was estimated in up to 9%, demonstrating that some modern compilers and dynamic out-of-order schedulers inside processors are unable to fully eliminate these contiguous dependencies.

In §5.4, some optimizations exploiting the projective equivalence were proposed for point operations. Revised formulas carefully optimized with the techniques described in this chapter are explicitly stated in Appendix B1 for the case of Jacobian ($\mathcal{J}$) and mixed Twisted Edwards homogeneous/extended homogeneous ($\mathcal{E}/\mathcal{E}^e$) coordinates.

The application of the rescheduling/merging of field operations and merging of point operations over quadratic extension fields was studied in §5.5. Revised formulas carefully optimized with these techniques (and techniques exploiting the projective equivalence; §5.4) are explicitly stated in Appendix B2 for the case of $\mathcal{J}$ and $\mathcal{E}/\mathcal{E}^e$ coordinates when using the GLS method over $\mathbb{F}_{p^2}$.

In §5.6, we illustrated the significant performance improvement obtained with the techniques under analysis with high-speed implementations of variable-scalar-variable-point scalar multiplication at the 128-bit security level. Our software was extensively code-optimized and incorporates state-of-the-art ECC algorithms, including the best precomputation scheme for each setting following results from Chapter 3. We presented *four* variants using either $\mathcal{E}/\mathcal{E}^e$ or $\mathcal{J}$ coordinates and with or without exploiting the GLS method. Remarkably, we showed that a point multiplication can be computed in only 181000 cycles (~60μsec.) on an AMD Phenom II X4 when combining $\mathcal{E}/\mathcal{E}^e$ with GLS. This represents a cost reduction of about 29% over the closest competitor. In the case of Jacobian coordinates with GLS, we reported a computation in only 188000 cycles (~60μsec.) in the same platform, which represents an improvement of about 34%. For the traditional case without using GLS, our implementations using $\mathcal{E}/\mathcal{E}^e$ and $\mathcal{J}$ coordinates are 22% and 28% faster than the previously best published results using the same coordinate systems. In summary, our implementations compute scalar multiplication up to 34% faster than the best previous results on x86-64 processors. We also reported that the use of GLS enables cost reductions as high as 45% on an Intel Atom and as high as 13% on an AMD Opteron.

Similar results are expected when exploiting the proposed optimizations with other curve forms and coordinate systems or with other scenarios involving, for instance, multiple scalar multiplication and fixed-point scalar multiplication, among others.

Chapter 6

Efficient Techniques for Implementing Pairings in Software

In this chapter, we propose efficient methods and optimized explicit formulas that speed up significantly the computation of pairings on ordinary curves over prime fields. Our contributions can be summarized as follows:

- We generalize the well-known technique of lazy reduction, previously applied to quadratic extension fields only [Sco07], to the whole pairing arithmetic including towering and curve arithmetic. We show that this approach leads to the elimination of at least 32% of the total number of reductions in a state-of-the-art implementation of the optimal ate pairing over a Barreto-Naehrig (BN) curve at the 128-bit security level.
- For dealing with more costly higher-precision additions required by lazy reduction, we develop a flexible methodology that keeps intermediate values under Montgomery reduction boundaries and maximizes the use of operations without carry checks.
- Following the approach detailed in Section 5.4, formulas for point doubling and addition in Jacobian and homogeneous coordinates are carefully optimized by eliminating several commonly neglected operations that are not inexpensive on modern 64-bit platforms.
- Finally, we illustrate the significant savings obtained by the new techniques with a high-speed implementation of the optimal ate pairing over a BN curve at the 128-bit security

level. By combining our methods with other state-of-the-art techniques, we obtain an implementation that improves the best available timings in the literature by 28%-34% on several x86-64-based processors.

This chapter is organized as follows. After discussing relevant previous work in §6.1, we describe the generalization of lazy reduction to pairing-friendly tower fields in §6.2. In the same section, we discuss how to optimize the implementation of tower field arithmetic when dealing with both single- and double-precision operations, and illustrate the flexible methodology with the popular tower $\mathbb{F}_p \rightarrow \mathbb{F}_{p^2} \rightarrow \mathbb{F}_{p^6} \rightarrow \mathbb{F}_{p^{12}}$. In §6.3, we present our optimizations to the curve arithmetic in the Miller loop, including the application of lazy reduction. Then, in §6.4 we describe our high-speed implementation of an optimal ate pairing on BN curves, carry out a detailed operation count and compare our results with the previously best results in the literature. We end this chapter with some conclusions in §6.5.

6.1. Previous Work

In recent years there has been a growing interest in pairing-based cryptography with numerous efforts focused on improving the efficiency of the pairing computation. Some works have proposed optimizations inside the Miller loop [BLS03, BLS04b] including the denominator elimination technique [BKL+02], while other works have focused on minimizing the length of the Miller's algorithm [HSV06, BGO+07, ZZH08, Ver10], constructing pairing-friendly elliptic curves [BLS03b, BW05, SB06, Fre06] and devising tower extensions of finite fields $\mathbb{F}_{p^k}$ [KM05, BS10]. An important research effort involves the development and optimization of explicit formulas for the curve arithmetic; see for example [CHB+09, CLN10]. Yet another crucial study involves the efficient implementation of the tower field $\mathbb{F}_{p^k}$ and base field arithmetic in $\mathbb{F}_p$ [FVV09, NNS10, BGM+10]. In this chapter we focus on the latter two issues and propose efficient methods for speeding up computations in the towering and curve arithmetic.

In the case of efficient implementation of the towering and base field arithmetic, some research warns of the potential danger of employing a prime p with low Hamming weight (e.g., Mersenne primes), in which case a modified NFS could reduce the security level [Sch10]. Therefore, the chosen prime should ideally have a general form, in which case Montgomery reduction is the most efficient method available [Mon85]; see Section 2.2.4.1. This ultimately makes modular reduction one of the most expensive operations in the underlying field arithmetic of pairings. Some efforts focus on improving the interaction between field multiplication and reduction to minimize costs [FVV09, NNS10]. A different approach involves instead the elimination of reductions by using the so-called *lazy reduction* [Sco07]. This technique goes back

to at least Crypto'98 [WD98], and has been advantageously exploited by many works in different scenarios [LH00, Ava04, Sco07]. According to [LH00], multiplication in $\mathbb{F}_{p^k}$ can be performed with k reductions modulo p when $\mathbb{F}_{p^k}$ is seen as a direct extension over $\mathbb{F}_p$ via an irreducible binomial. This improves the traditional method that requires k^2 reductions (or $k(k+1)/2$ reductions when using Karatsuba multiplication). Lazy reduction was first employed in the context of pairing computation by [Sco07] to eliminate reductions in $\mathbb{F}_{p^2}$ multiplication. Essentially, when using Karatsuba method for multiplication in $\mathbb{F}_{p^2}$, lazy reduction allows us to lower the number of reductions from 3 to only 2. Note that these savings are at the cost of somewhat more expensive additions. If, for instance, one considers the tower $\mathbb{F}_p \rightarrow \mathbb{F}_{p^2} \rightarrow \mathbb{F}_{p^6} \rightarrow \mathbb{F}_{p^{12}}$, then this approach requires $3 \cdot 6 \cdot 3 = 54$ integer multiplications with $2 \cdot 6 \cdot 3 = 36$ reductions modulo p for performing one multiplication in $\mathbb{F}_{p^{12}}$; see [Sco07, HMS08, BGM+10].

In this work we go a step further and generalize lazy reduction to the whole pairing computation, including the tower extension and curve arithmetic. For instance, these optimizations allow us to eliminate about 32% of reductions in a state-of-the-art implementation of the optimal ate pairing using a BN curve with 128 bits of security; see Section 6.4.2 for complete details.

Recently, many authors have targeted the efficient software implementation of bilinear pairings at the 128-bit security level. Most remarkable results include the computation of the R-ate pairing in 10,000,000 cycles on one core of an Intel Core 2 Duo processor by Hankerson et al. [HMS08], and the computation of the optimal ate pairing in 4,380,000 cycles on one core of an Intel Core 2 Quad Q9550 by Naehrig et al. [NNS10] and in 2,950,000 cycles on one core of an Intel Core 2 Duo T7100 by Beuchat et al. [BGM+10]. Beuchat et al. also reports an optimal ate pairing computation in 2,330,000 cycles on one core of an Intel Core i7 860 processor.

In this work, to demonstrate the effectiveness of our optimizations, we realize a high-speed implementation of an optimal ate pairing at the 128-bit security level that additionally incorporates the latest advancements in the area, including software techniques by Beuchat et al. [BGM+10] to optimize carry handling and eliminate function call overheads in the $\mathbb{F}_{p^2}$ arithmetic, and the use of efficient compressed squarings and decompression in cyclotomic subgroups to speed up computations in the final exponentiation by Karabina (see [Kar10] and also [AKL+10, Section 5.2]). We report a pairing computation in 2,194,000 cycles on one core of an Intel Core 2 Duo E6750 and in 1,688,000 cycles on an Intel Core i5 540M. Moreover, we also report a pairing computation in only 1,562,000 cycles (~0.5msec.) on an AMD Phenom II X4 940 processor. Taking into account timings in identical platforms, our results introduce improvements between 28% and 34% in comparison with the best previous results.

6.2. Lazy Reduction for Tower Fields

In this section, we generalize the lazy reduction technique to towering-friendly fields $\mathbb{F}_{p^k}$, with $k = 2^i 3^j$, $i \geq 1$, $j \geq 0$, as defined by [BS10], which are conveniently built with irreducible binomials. We show that multiplication (and squaring) in a tower extension $\mathbb{F}_{p^k}$ only requires k reductions and still benefits from different arithmetic optimizations available in the literature to reduce the number of subfield multiplications/squarings. For instance, with our approach one now requires $3 \cdot 6 \cdot 3 = 54$ integer multiplications and $2 \cdot 3 \cdot 2 = 12$ reductions modulo p using the tower $\mathbb{F}_p \rightarrow \mathbb{F}_{p^2} \rightarrow \mathbb{F}_{p^6} \rightarrow \mathbb{F}_{p^{12}}$ to compute one multiplication in $\mathbb{F}_{p^{12}}$ (eliminating 24 reductions in comparison with the traditional approach); or 36 integer multiplications and 12 reductions modulo p to compute one squaring in $\mathbb{F}_{p^{12}}$ (eliminating 18 reductions in comparison with the traditional approach). Although wider in generality, these techniques are analyzed in detail in the context of Montgomery multiplication and Montgomery reduction [Mon85], which are commonly used in the context of pairings over ordinary curves. We explicitly state our formulas for the towering construction $\mathbb{F}_p \rightarrow \mathbb{F}_{p^2} \rightarrow \mathbb{F}_{p^6} \rightarrow \mathbb{F}_{p^{12}}$ in Section 6.2.2. In the remainder, the term *modular reduction modulo p* always refers to modular reductions of double-precision integers.

Lemma 6.1. A sum of products with the form $\sum \pm a_i \cdot b_i \bmod p$, where a_i, b_i are elements in Montgomery representation, can be reduced with only one Montgomery reduction modulo p by accumulating inner products as double-precision integers always that $\sum \pm a_i \cdot b_i < 2^N \cdot p$, where $N = n \cdot w$, n is the exact number of words required to represent p, i.e., $n = \lceil \lceil \log_2 p \rceil / w \rceil$, and w is the computer word-size.

Lemma 6.1 defines the basic lazy reduction technique in the context of Montgomery reduction. Readers should note that internal additions and subtractions with partial results r "slightly" outside the Montgomery reduction range $[0, 2^N \cdot p]$, i.e., $2^N \cdot p \leq r < 2^{N+1} \cdot p$, can be easily corrected at negligible cost by performing a subtraction with $2^N \cdot p$.

Next, we present our main result applying lazy reduction to towering-friendly fields.

Theorem 6.1. In a towering-friendly field $\mathbb{F}_{p^k}$, multiplications and squarings can be computed with k reductions.

Proof. We will proof this theorem in a wider context for generic tower extensions built with irreducible binomials. Let $\mathbb{F}_{p^k}$ be a direct extension of $\mathbb{F}_p$, where $k = \prod_{i=1}^{t} n_i$, and an element $a \in \mathbb{F}_{p^k}$ be represented in polynomial basis as $a(X) = a_0 + a_1 X + \ldots + a_{k-1} X^{k-1}$. Then one can use the following tower construction $\mathbb{F}_{p^{n_0}} = \mathbb{F}_p \rightarrow \mathbb{F}_{p^{n_1}} \rightarrow \mathbb{F}_{p^{n_1 \cdot n_2}} \rightarrow \ldots \rightarrow \mathbb{F}_{p^{n_1 \cdot n_2 \cdot \ldots \cdot n_t}} = \mathbb{F}_{p^k}$ to represent the extension field $\mathbb{F}_{p^k}$ s.t. $\mathbb{F}_{p^{n_1}} = \mathbb{F}_p[u] / (u^{n_1} - \beta)$, $\mathbb{F}_{p^{n_1 \cdot n_2}} = \mathbb{F}_{p^{n_1}}[v] / (v^{n_2} - \xi)$, …,

$\mathbb{F}_{p^k} = \mathbb{F}_{p^{n_1 \cdot n_2 \cdot \ldots \cdot n_{t-1}}}[w] \Big/ (w^{n_t} - \gamma)$. Assuming that an element $c \in \mathbb{F}_{p^{n_1}}$ is represented as $c(u) = c_0 + c_1 u + \ldots + c_{n_1-1} u^{n_1-1}$, where $c_i \in \mathbb{F}_p$, polynomial multiplication of two elements $a, b \in \mathbb{F}_{p^{n_1}}$ can be expressed as:

$$\begin{aligned} c(u) = a(u) \cdot b(u) &= \sum_{i=0}^{n_1-1} a_i u^i \cdot \sum_{i=0}^{n_1-1} b_i u^i \bmod (u^{n_1} - \beta) \\ &= \sum_{m=0}^{n_1-1} \left(\sum_{j=0}^{m} a_j b_{m-j} \bmod p + \beta \sum_{j=m+1}^{n_1-1} a_j b_{m-j+n_1} \bmod p \right) u^m \\ &= \sum_{m=0}^{n_1-1} \left(\left(\sum_{j=0}^{m} a_j b_{m-j} + \beta \sum_{j=m+1}^{n_1-1} a_j b_{m-j+n_1} \right) \bmod p \right) u^m = \sum_{m=0}^{n_1-1} \left(c_m \bmod p \right) u^m, \end{aligned} \tag{6.1}$$

where only n_1 reductions are necessary by applying Lemma 6.1.

Similarly, assuming that an element $f \in \mathbb{F}_{p^{n_1 \cdot n_2}}$ is represented as $f(v) = f_0 + f_1 v + \ldots + f_{n_2-1} v^{n_2-1}$, where $f_i \in \mathbb{F}_{p^{n_1}}$, polynomial multiplication of two elements $d, e \in \mathbb{F}_{p^{n_1 \cdot n_2}}$ can be expressed by:

$$\begin{aligned} f(v) = d(v) \cdot e(v) &= \sum_{i=0}^{n_2-1} d_i v^i \cdot \sum_{i=0}^{n_2-1} e_i v^i \bmod (v^{n_2} - \xi) \\ &= \sum_{l=0}^{n_2-1} \left(\sum_{j=0}^{l} d_j e_{l-j} + \xi \sum_{j=l+1}^{n_2-1} d_j e_{l-j+n_2} \right) v^l . \end{aligned} \tag{6.2}$$

Then, by using (6.1) to perform $\mathbb{F}_{p^{n_1}}$ multiplications $d_x \cdot e_y$ from (6.2) and applying Lemma 6.1 again, we obtain the following expression for multiplication in $\mathbb{F}_{p^{n_1 \cdot n_2}}$:

$$\begin{aligned} f(v) &= \sum_{l=0}^{n_2-1} \left(\sum_{j=0}^{l} \sum_{m=0}^{n_1-1} \left(c_{m,j,l-j} \bmod p \right) u^m + \xi \sum_{j=l+1}^{n_2-1} \sum_{m=0}^{n_1-1} \left(c_{m,j,l-j+n_2} \bmod p \right) u^m \right) v^l \\ &= \sum_{l=0}^{n_2-1} \left(\sum_{m=0}^{n_1-1} \left(\left(\sum_{j=0}^{l} c_{m,j,l-j} + \xi \sum_{j=l+1}^{n_2-1} c_{m,j,l-j+n_2} \right) \bmod p \right) u^m \right) v^l \\ &= \sum_{l=0}^{n_2-1} \left(\sum_{m=0}^{n_1-1} \left(f_{l,m} \bmod p \right) u^m \right) v^l , \end{aligned} \tag{6.3}$$

where $c_{m,x,y}$ correspond to coefficients c_m in (6.1) for each multiplication $d_x \cdot e_y$ from (6.2). Note that again reductions have been shifted and only $n_1 \cdot n_2$ reductions are required in each $\mathbb{F}_{p^{n_1 \cdot n_2}}$ multiplication. If one continues applying this procedure recursively it can be easily seen that a multiplication in $\mathbb{F}_{p^k} = \mathbb{F}_{p^{n_1 \cdot n_2 \cdot \ldots \cdot n_t}}$ requires $k = n_1 \cdot n_2 \cdot \ldots \cdot n_t$ reductions. □

It is important to note that there is no restriction in the selection of parameters for the

irreducible binomials (e.g., β and ξ in the proof above). However, for efficiency purposes one should select parameters with small values such that multiplications with them can be converted to a few additions and subtractions (see for example the chosen parameters in the illustrative tower in Section 6.2.2).

The next theorem extends our result to towering-friendly fields exploiting Karatsuba multiplication.

Theorem 6.2. Multiplications in a towering-friendly field $\mathbb{F}_{p^k}$ built with irreducible binomials, where $k = 2^d 3^e$, $d \geq 1$, $e \geq 0$, can be computed with *k* reductions and $3^d 6^e$ multiplications.

Proof. Let the tower $\mathbb{F}_{p^{k_0}} = \mathbb{F}_p \rightarrow \mathbb{F}_{p^{k_1}} = \mathbb{F}_{p^{n_1}} \rightarrow \mathbb{F}_{p^{k_2}} = \mathbb{F}_{p^{n_1 \cdot n_2}} \rightarrow \ldots \rightarrow \mathbb{F}_{p^{k_t}} = \mathbb{F}_{p^{n_1 \cdot n_2 \cdots n_t}}$ represent $\mathbb{F}_{p^{k_t}}$ s.t. $\mathbb{F}_{p^{k_i}} = \mathbb{F}_{p^{k_{i-1}}}[x] \Big/ (x^{n_i} - \beta_{k_i})$ with $n_i \in \{2,3\}$, $i > 0$ integer, and assume β_{k_i} are chosen such that multiplication by these values can be computed with a few additions or subtractions. Then, multiplication $c = a \cdot b$ of two elements $a = (a_0 + a_1 x), b = (b_0 + b_1 x) \in \mathbb{F}_{p^{k_i}}$ with $n_i = 2$ (second degree irred. binomial) and $a_i, b_i \in \mathbb{F}_{p^{k_{i-1}}}$ can be computed using Karatsuba method as follows:

$$c = (a_0 b_0 + a_1 b_1 \beta_{k_i}) + \left[(a_0 + a_1)(b_0 + b_1) - a_0 b_0 - a_1 b_1\right] x, \tag{6.4}$$

which requires 3 integer multiplications in $\mathbb{F}_{p^{k_{i-1}}}$. Similarly, multiplication $c = a \cdot b$ of two elements $a = (a_0 + a_1 x + a_2 x^2)$ and $b = (b_0 + b_1 x + b_2 x^2) \in \mathbb{F}_{p^{k_i}}$ with $n_i = 3$ (third degree irred. binomial) and $a_i, b_i \in \mathbb{F}_{p^{k_{i-1}}}$ can be computed using Karatsuba method as follows:

$$\begin{gathered} v_0 = a_0 \cdot b_0,\ v_1 = a_1 \cdot b_1,\ v_2 = a_2 \cdot b_2, \\ c_0 = v_0 + \beta_{k_i}\left[(a_1 + a_2)(b_1 + b_2) - v_1 - v_2\right],\ c_1 = (a_0 + a_1)(b_0 + b_1) - v_0 - v_1 + v_2 \beta_{k_i}, \\ c_1 = (a_0 + a_2)(b_0 + b_2) - v_0 + v_1 - v_2,\ c = c_0 + c_1 x + c_1 x^2, \end{gathered} \tag{6.5}$$

which requires 6 integer multiplications in $\mathbb{F}_{p^{k_{i-1}}}$. Therefore, a tower consisting of *d* second degree extensions (i.e., $n_i = 2$) and *e* third degree extensions (i.e., $n_i = 3$) can be executed with $3^d 6^e$ multiplications. Following Theorem 6.1, we only require *k* reductions in total since terms in both expressions (6.4) and (6.5) are simply sums of products as required. □

Theorem 6.2 shows that lazy reduction can be combined with Karatsuba multiplication for efficient computation in tower extension fields. In fact, it is straightforward to generalize lazy reduction to any formula that also involves only sums (or subtractions) of products of the form $\sum \pm a_i \cdot b_i$, with $a_i, b_i \in \mathbb{F}_{p^{k_l}}$, such as complex squaring or the asymmetric squaring formulas devised by Chung and Hasan [CH07].

For efficiency purposes, we suggest a different treatment for the highest layer in the tower

arithmetic. Proof of Theorem 6.1 implies that reductions can be completely delayed to the end of the last layer by applying lazy reduction, but in some cases (when the optimal k is already reached and no reductions can be saved) it will be more efficient to perform reductions right after multiplications or squarings. This will be illustrated later with the computation of squaring in $\mathbb{F}_{p^{12}}$ in Section 6.2.2.

In summary, the generalized lazy reduction can be applied to every computation involving operations in tower extension fields in the Miller loop and final exponentiation, including the recently proposed compressed squarings by [Kar10] (see Appendix C1).

Remarkably, in the Miller Loop reductions can also be delayed from the underlying $\mathbb{F}_{p^2}$ field during multiplication and squaring to the arithmetic layer immediately above, i.e., the point arithmetic and line evaluation. Similarly to the tower extension, reductions on this upper layer should only be delayed in the cases where this technique leads to fewer reductions. For details, see Section 6.3.

There are some penalties when delaying reductions. In particular, *single-precision* operations (with operands occupying $n = \lceil \lceil \log_2 p \rceil / w \rceil$ words, where w is the computer word-size) are replaced by *double-precision* operations (with operands occupying $2n$ words). However, this disadvantage can be minimized in terms of speed by selecting a field size smaller than the word-size boundary because this technique can be exploited more extensively for optimizing double-precision arithmetic.

6.2.1. Selecting a Field Size Smaller than the Word-Size Boundary

If the modulus p is selected such that $l = \lceil \log_2 p \rceil < N$, where again $N = n \cdot w$, $n = \lceil l/w \rceil$ and w is the computer word-size, then several consecutive additions without carry-out in the most significant word (MSW) can be performed before a multiplication with the form $c = a \cdot b$, where $a, b \in [0, 2^N - 1]$ s.t. $c < 2^{2N}$. In the case of Montgomery reduction, the restriction is given by the upper bound $c < 2^N \cdot p$. Similarly, when delaying reductions the result of a multiplication without reduction has maximum value $(p-1)^2 < 2^{2N}$ (assuming that $a, b \in [0, p]$) and several consecutive double-precision additions without carry-outs in the MSW (and, in some cases, subtractions without borrow-outs in the MSW) can be performed before reduction. When using Montgomery reduction up to $\sim \lfloor 2^N / p \rfloor$ additions can be performed without carry checks.

Furthermore, cheaper single- and double-precision operations exploiting this "extra room" can be combined for maximal performance. The challenge is to optimally balance their use in the tower arithmetic since both may interfere with each other. For instance, if intermediate values are allowed to grow up to $2p$ before multiplication (instead of p) then the maximum result would be $4p^2$. This strategy makes use of cheaper single-precision additions without carry checks but limits the number of double-precision additions that can be executed without carry checks after

multiplication with delayed reduction. As it will be evident later, to maximize the gain obtained with the proposed methodology one should take into account relative costs of operations and maximum bounds.

In the case of double-precision arithmetic, different optimizing alternatives are available. Let us analyze them in the context of Montgomery arithmetic. First, as pointed out by [BGM+10], if $c > 2^N \cdot p$, where c is the result of a double-precision addition, then c can be restored with a cheaper single-precision subtraction by $2^N \cdot p$ (note that the first half of this value consists of zeroes only). Second, different options are available to convert negative numbers to positive after double-precision subtraction. In particular, let us consider the computation $c = a + l \cdot b$, where $a, b \in [0, mp^2]$, $m \in \mathbb{Z}^+$ and $l < 0 \in \mathbb{Z}$, which is a recurrent operation (for instance, when $l = \beta$ from Section 6.2.2). For this operation, we have explored the alternatives listed in Table 6.1, which can be integrated in the tower arithmetic with different advantages.

Table 6.1. Different options to convert negative results to positive after a subtraction with the form $c = a + l \cdot b$, where $a, b \in [0, mp^2]$, $m \in \mathbb{Z}^+$ and $l < 0 \in \mathbb{Z}$ s.t. $|lmp| < 2^N$.

Option 1: $r = c + (2^N \cdot p / 2^h)$, $r \in [0, mp^2 + 2^N \cdot p / 2^h]$, where h is a small integer s.t. $\lvert lmp^2 \rvert < 2^N \cdot p / 2^h < 2^N \cdot p - mp^2$.
Option 2: if $c < 0$ then $r = c + 2^N \cdot p$, $r \in [0, 2^N \cdot p]$.
Option 3: $r = c - lmp^2$, $r \in [0, (\lvert l \rvert + 1)mp^2]$, s.t. $(\lvert l \rvert + 1)mp < 2^N$.
Option 4: if $c < 0$ then $r = c - lmp^2$, $r \in [0, \lvert lmp^2 \rvert]$.

In particular, Options 2 and 4 in Table 6.1 require conditional checks that make the corresponding operations more expensive. Nevertheless, these options may be valuable when negative values cannot be corrected with other options without violating the upper bound. Also note that Option 2 can make use of a cheaper single-precision subtraction for converting negative results to positive. Options 1 and 3 are particularly efficient because no conditional checks are required. Moreover, if l is small enough (and h maximized for Option 1) several following operations can avoid carry checks. Between both, Option 1 is generally more efficient because adding $2^N \cdot p / 2^h$ requires less than double-precision if $h \leq w$, where w is the computer word-size.

Next, we demonstrate how the different design options discussed in this section can be exploited with a clever selection of parameters and applied to different operations combining single- and double-precision arithmetic to speed up the extension field arithmetic.

6.2.2. Practical Application of the Generalized Lazy Reduction

For our illustrative analysis, we use the tower $\mathbb{F}_p \rightarrow \mathbb{F}_{p^2} \rightarrow \mathbb{F}_{p^6} \rightarrow \mathbb{F}_{p^{12}}$ constructed as follows [PSN+10]:

- $\mathbb{F}_{p^2} = \mathbb{F}_p[i]/(i^2 - \beta)$, where $\beta = -1$.
- $\mathbb{F}_{p^6} = \mathbb{F}_{p^2}[v]/(v^3 - \xi)$, where $\xi = 1 + i$.
- $\mathbb{F}_{p^{12}} = \mathbb{F}_{p^6}[w]/(w^2 - v)$.

We use a similar tower construction for our illustrative implementation of the optimal ate pairing on a BN curve (see Section 6.4.1 for complete details).

When targeting the 128-bit security level, single- and double-precision operations are defined by operands with sizes N = 256 and $2N$ = 512, respectively. For our selected prime, $\lceil \log_2 p \rceil = 254$ and $2^N \cdot p \approx 6.8 p^2$. We use the following notation [AKL+10]:

(i) $+, -, \times$ are operators not involving carry handling or modular reduction for boundary keeping;
(ii) $\oplus$, $\ominus$, $\otimes$ are operators producing reduced results through carry handling or modular reduction;
(iii) a superscript in an operator is used to denote the extension degree involved in the operation;
(iv) notation $a_{i,j}$ is used to address j-th subfield element inside extension field element a_i;
(v) variables with lower case t and upper case T represent single- and double-precision integers or extension field elements composed of single and double-precision integers, respectively.

The following notation is used for the cost of operations:

(i) M, S, A denote the cost of multiplication, squaring, addition in $\mathbb{F}_p$, respectively;
(ii) m, s, a, i denote the cost of multiplication, squaring, addition and inversion in $\mathbb{F}_{p^2}$, respectively;
(iii) M_u, S_u, R denote the cost of unreduced multiplication and squaring producing double-precision results, and modular reduction of double-precision elements in $\mathbb{F}_p$, respect.;
(iv) m_u, s_u, r denote the cost of unreduced multiplication and squaring, and modular reduction of double-precision elements in $\mathbb{F}_{p^2}$, respectively.

For the remainder of the chapter, we assume that (except when explicitly stated) double-

precision addition has the cost of 2*A* and 2*a* in $\mathbb{F}_p$ and $\mathbb{F}_{p^2}$, respectively, which approximately follows what we observe in practice.

Note that, as stated before, if $c > 2^N \cdot p$ after adding $c = a + b$ in double-precision we correct the result by computing $c - 2^N \cdot p$. Similar to subtraction (see Table 6.1), we refer to the latter as "Option 2". Remaining references to "Option *x*" are taken from Table 6.1.

We will now illustrate a selection of operations for efficient multiplication in $\mathbb{F}_{p^{12}}$, beginning with multiplication in $\mathbb{F}_{p^2}$. Let $a, b, c \in \mathbb{F}_{p^2}$ such that $a = a_0 + a_1 i$, $b = b_0 + b_1 i$, $c = a \cdot b = c_0 + c_1 i$. The required operations for computing $\mathbb{F}_{p^2}$ multiplication are detailed in Algorithm 6.1. As explained in Beuchat et al. [BGM+10, Section 5.2], when using the Karatsuba method and $a_i, b_i \in \mathbb{F}_p$, $c_1 = (a_0 + a_1)(b_0 + b_1) - a_0 b_0 - a_1 b_1 = a_0 b_1 + a_1 b_0 < 2p^2 < 2^N \cdot p$, additions are single-precision, reduction after multiplication can be delayed and hence subtractions are double-precision (steps 1-3 in Algorithm 6.1). Obviously, these operations do not require carry checks. For $c_0 = a_0 b_0 - a_1 b_1$, c_0 is in interval $[-p^2, p^2]$ and a negative result can be converted to positive using **Option 1** with $h = 2$ or **Option 2**, for which the final c_0 is in the range $[0, (2^N \cdot p/4) + p^2] \subset [0, 2^N \cdot p]$ or $[0, 2^N \cdot p]$, respectively (step 4 in Algorithm 6.1). Following Theorem 6.1, all reductions can be completely delayed to the next arithmetic layer (higher extension or curve arithmetic).

Algorithm 6.1. Multiplication in $\mathbb{F}_{p^2}$ without reduction ($\times^2$, cost of $m_u = 3M_u + 8A$)

Input: $a = (a_0 + a_1 i)$ and $b = (b_0 + b_1 i) \in \mathbb{F}_{p^2}$

Output: $c = a \cdot b = (c_0 + c_1 i) \in \mathbb{F}_{p^2}$

1: $T_0 \leftarrow a_0 \times b_0$, $T_1 \leftarrow a_1 \times b_1$, $t_0 \leftarrow a_0 + a_1$, $t_1 \leftarrow b_0 + b_1$
2: $T_2 \leftarrow t_0 \times t_1$, $T_3 \leftarrow T_0 + T_1$
3: $T_3 \leftarrow T_2 - T_3$
4: $T_4 \leftarrow T_0 \ominus T_1$ (Option 1 or 2)
5: Return $c = (T_4 + T_3 i)$

Let us now define multiplication in $\mathbb{F}_{p^6}$. Let $a, b, c \in \mathbb{F}_{p^6}$ such that $a = (a_0 + a_1 v + a_2 v^2)$, $b = (b_0 + b_1 v + b_2 v^2)$, $c = a \cdot b = (c_0 + c_1 v + c_2 v^2)$. The required operations for computing $\mathbb{F}_{p^6}$ multiplication are detailed in Algorithm 6.2. In this case, $c_0 = v_0 + \xi[(a_1 + a_2)(b_1 + b_2) - v_1 - v_2]$, $c_1 = (a_0 + a_1)(b_0 + b_1) - v_0 - v_1 + \xi v_2$ and $c_2 = (a_0 + a_2)(b_0 + b_2) - v_0 - v_2 + v_1$, where $v_0 = a_0 b_0$, $v_1 = a_1 b_1$ and $v_2 = a_2 b_2$. First, note that the pattern $s_x = (a_i + a_j)(b_i + b_j) - v_i - v_j$ repeats for each c_x, $0 \le x \le 2$. After multiplications using Algorithm 6.1 with **Option 1** (h = 2), we have $v_{i,0}, v_{j,0} \in [0, (2^N \cdot p/4) + p^2]$ and $v_{i,1}, v_{j,1} \in [0, 2p^2]$ (step 1 of Alg. 6.2). Outputs of single-precision additions of the forms $(a_i + a_j)$ and $(b_i + b_j)$ are in the range [0, 2*p*] and hence do not produce carries (steps 2, 9 and 17 of Alg. 6.2). Corresponding $\mathbb{F}_{p^2}$ multiplications $r_x = (a_i + a_j)(b_i + b_j)$

Algorithm 6.2. Multiplication in $\mathbb{F}_{p^6}$ without reduction ($\times^6$, cost of $6m_u + 28a$)

Input: $a = (a_0 + a_1 v + a_2 v^2)$ and $b = (b_0 + b_1 v + b_2 v^2) \in \mathbb{F}_{p^6}$

Output: $c = a \cdot b = (c_0 + c_1 v + c_2 v^2) \in \mathbb{F}_{p^6}$

1:	$T_0 \leftarrow a_0 \times^2 b_0,\ T_1 \leftarrow a_1 \times^2 b_1,\ T_2 \leftarrow a_2 \times^2 b_2$	(Option 1, $h = 2$)
2:	$t_0 \leftarrow a_1 +^2 a_2,\ t_1 \leftarrow b_1 +^2 b_2$	
3:	$T_3 \leftarrow t_0 \times^2 t_1$	(Option 2)
4:	$T_4 \leftarrow T_1 +^2 T_2$	
5:	$T_{3,0} \leftarrow T_{3,0} \ominus T_{4,0}$	(Option 2)
6:	$T_{3,1} \leftarrow T_{3,1} - T_{4,1}$	
7:	$T_{4,0} \leftarrow T_{3,0} \ominus T_{3,1},\ T_{4,1} \leftarrow T_{3,0} \oplus T_{3,1}\ (\equiv T_4 \leftarrow \xi \cdot T_3)$	(Option 2)
8:	$T_5 \leftarrow T_4 \oplus^2 T_0$	(Option 2)
9:	$t_0 \leftarrow a_0 +^2 a_1,\ t_1 \leftarrow b_0 +^2 b_1$	
10:	$T_3 \leftarrow t_0 \times^2 t_1$	(Option 2)
11:	$T_4 \leftarrow T_0 +^2 T_1$	
12:	$T_{3,0} \leftarrow T_{3,0} \ominus T_{4,0}$	(Option 2)
13:	$T_{3,1} \leftarrow T_{3,1} - T_{4,1}$	
14:	$T_{4,0} \leftarrow T_{2,0} \ominus T_{2,1}$	(Option 1, $h = 1$)
15:	$T_{4,1} \leftarrow T_{2,0} + T_{2,1}$ (steps 14-15 $\equiv T_4 \leftarrow \xi \cdot T_2$)	
16:	$T_6 \leftarrow T_3 \oplus^2 T_4$	(Option 2)
17:	$t_0 \leftarrow a_0 +^2 a_2,\ t_1 \leftarrow b_0 +^2 b_2$	
18:	$T_3 \leftarrow t_0 \times^2 t_1$	(Option 2)
19:	$T_4 \leftarrow T_0 +^2 T_2$	
20:	$T_{3,0} \leftarrow T_{3,0} \ominus T_{4,0}$	(Option 2)
21:	$T_{3,1} \leftarrow T_{3,1} - T_{4,1}$	
22:	$T_{7,0} \leftarrow T_{3,0} \oplus T_{1,0}$	(Option 2)
23:	$T_{7,1} \leftarrow T_{3,1} + T_{1,1}$	
24:	Return $c = (T_5 + T_6 v + T_7 v^2)$	

using Algorithm 6.1 with **Option 2** give results in the ranges $r_{x,0} \in [0, 2^N \cdot p]$ and $r_{x,1} \in [0, 8p^2]$ (steps 3, 10 and 18). Although $\max(r_{x,1}) = 8p^2 > 2^N \cdot p$, note that $8p^2 < 2^{2N}$ and $s_{x,1} = a_{i,0}b_{j,1} + a_{i,1}b_{j,0} + a_{j,0}b_{i,1} + a_{j,1}b_{i,0} \in [0, 4p^2]$ since $s_x = a_i b_j + a_j b_i$. Hence, for $0 \le x \le 2$, double-precision subtractions for computing $s_{x,1}$ using Karatsuba do not require carry checks (steps 4 and 6, 11 and 13, 19 and 21). For computing $s_{x,0} = r_{x,0} - (v_{i,0} + v_{j,0})$ addition does not require carry check (output range $[0, 2(2^N \cdot p/4 + p^2)] \subset [0, 2^N \cdot p]$) and subtraction gives result in the range $[0, 2^N \cdot p]$ when using **Option 2** (steps 5, 12 and 20). For computing c_0, multiplication by ξ, i.e., $S_0 = \xi s_0$ involves the operations $S_{0,0} = s_{0,0} - s_{0,1}$ and $S_{0,1} = s_{0,0} + s_{0,1}$,

which are computed in double-precision using **Option 2** to get the output range $[0,2^N \cdot p]$ (step 7). Similarly, final additions with v_0 require **Option 2** to get again the output range $[0,2^N \cdot p]$ (step 8). For computing c_1, $S_1 = \xi v_2$ is computed as $S_{1,0} = v_{2,0} - v_{2,1}$ and $S_{1,1} = v_{2,0} + v_{2,1}$, where the former requires a double-precision subtraction using **Option 1** (h = 1) to get a result in the range $[0,2^N \cdot p/2 + 2^N \cdot p/4 + p^2)] \subset [0,2^N \cdot p]$ (step 14) and the latter requires a double-precision addition with no carry check to get a result in the range $[0,(2^N \cdot p/4) + 3p^2] \subset [0,2^N \cdot p]$ (step 15). Then, $c_{1,0} = s_{1,0} + S_{1,0}$ and $c_{1,1} = s_{1,1} + S_{1,1}$ involve double-precision additions using **Option 2** to obtain results in the range $[0,2^N \cdot p]$ (step 16). Results $c_{2,0} = s_{2,0} + v_{1,0}$ and $c_{2,1} = s_{2,1} + v_{1,1}$ require a double-precision addition using **Option 2** (final output range $[0,2^N \cdot p]$, step 22) and a double-precision addition without carry check (final output range $[0,6p^2] \subset [0,2^N \cdot p]$, step 23), respectively. Modular reductions have been delayed again to the last layer $\mathbb{F}_{p^{12}}$.

Finally, let $a,b,c \in \mathbb{F}_{p^{12}}$ such that $a = a_0 + a_1 w$, $b = b_0 + b_1 w$, $c = a \cdot b = c_0 + c_1 w$. Algorithm 6.3 details the required operations for computing multiplication in $\mathbb{F}_{p^{12}}$. In this case, $c_1 = (a_0 + a_1)(b_0 + b_1) - a_0 b_0 - a_1 b_1$. At step 1, $\mathbb{F}_{p^6}$ multiplications $a_0 b_0$ and $a_1 b_1$ give outputs in range $\subset [0,2^N \cdot p]$ using Algorithm 6.2. Additions $a_0 + a_1$ and $b_0 + b_1$ are single-precision reduced modulo p so that multiplication $(a_0 + a_1)(b_0 + b_1)$ in step 2 gives output in range $\subset [0,2^N \cdot p]$ using Algorithm 6.2. Then, subtractions by $a_1 b_1$ and $a_0 b_0$ use double-precision operations with **Option 2** to have an output range $[0,2^N \cdot p]$ so that we can apply Montgomery reduction at step 5 to obtain the result modulo p. For $c_0 = a_0 b_0 + v a_1 b_1$, multiplication by v, i.e., $T = v \cdot v_1$, where $v_i = a_i b_i$, involves the double-precision operations $T_{0,0} = v_{2,0} - v_{2,1}$, $T_{0,1} = v_{2,0} + v_{2,1}$, $T_1 = v_0$ and $T_2 = v_1$, all performed with **Option 2** to obtain the output range $[0,2^N \cdot p]$ (steps 6-7). Final addi-

Algorithm 6.3. Multiplication in $\mathbb{F}_{p^{12}}$ ($\times^{12}$, cost of $18m_u + 6r + 110a$)

Input: $a = (a_0 + a_1 w)$ and $b = (b_0 + b_1 w) \in \mathbb{F}_{p^{12}}$

Output: $c = a \cdot b = (c_0 + c_1 w) \in \mathbb{F}_{p^{12}}$

1: $T_0 \leftarrow a_0 \times^6 b_0$, $T_1 \leftarrow a_1 \times^6 b_1$, $t_0 \leftarrow a_0 \oplus^6 a_1$, $t_1 \leftarrow b_0 \oplus^6 b_1$
2: $T_2 \leftarrow t_0 \times^6 t_1$
3: $T_3 \leftarrow T_0 \oplus^6 T_1$ (Option 2)
4: $T_2 \leftarrow T_2 \ominus^6 T_3$ (Option 2)
5: $c_1 \leftarrow T_2 \bmod^6 p$
6: $T_{2,2,0} \leftarrow T_{1,2,0} \ominus T_{1,2,1}$, $T_{2,0,1} \leftarrow T_{1,2,0} \oplus T_{1,2,1}$ (Option 2)
7: $T_{2,1} \leftarrow T_{1,0}$, $T_{2,2} \leftarrow T_{1,1}$ (steps 6-7 $\equiv T_2 \leftarrow v \cdot T_1$)
8: $T_2 \leftarrow T_0 \oplus^6 T_2$ (Option 2)
9: $c_0 \leftarrow T_2 \bmod^6 p$
10: Return $c = (c_0 + c_1 w)$

tion with a_0b_0 uses double-precision with **Option 2** again so that we can apply Montgomery reduction at step 9 to obtain the result modulo p. We remark that, by applying the lazy reduction technique using the operation sequence above, we have reduced the number of reductions in $\mathbb{F}_{p^6}$ from 3 to only 2, or the number of total modular reductions in $\mathbb{F}_p$ from 54 (or 36 if lazy reduction is employed in $\mathbb{F}_{p^2}$) to only $k = 12$.

As previously stated, there are situations when it is more efficient to perform reductions right after multiplications and squarings in the last arithmetic layer of the tower construction. We illustrate the latter with squaring in $\mathbb{F}_{p^{12}}$. As shown in Algorithm 6.4, a total of 2 reductions in $\mathbb{F}_{p^6}$ are required when performing $\mathbb{F}_{p^6}$ multiplications in step 4. If lazy reduction was applied, the number of reductions would stay at 2, and worse, the total cost would be increased because some operations would require double-precision. The reader should note that the approach suggested by [PSN+10], where the formulas in [CH07] are employed for computing squarings in internal cubic extensions of $\mathbb{F}_{p^{12}}$, saves $1m$ in comparison with Algorithm 6.4. However, we experimented such approach with several combinations of formulas and towering, and it remained consistently slower than Algorithm 6.4 due to an increase in the number of additions.

Algorithm 6.4. Squaring in $\mathbb{F}_{p^{12}}$ (cost of $12m_u + 6r + 73a$)

Input: $a = (a_0 + a_1 w) \in \mathbb{F}_{p^{12}}$

Output: $c = a^2 = (c_0 + c_1 w) \in \mathbb{F}_{p^{12}}$

1: $t_0 \leftarrow a_0 \oplus^6 a_1,\ t_{1,0,0} \leftarrow a_{1,2,0} \ominus a_{1,2,1},\ t_{1,0,1} \leftarrow a_{1,2,0} \oplus a_{1,2,1}$
2: $t_{1,1} \leftarrow a_{1,0},\ t_{1,2} \leftarrow a_{1,1}$ (steps 2-3 $\equiv t_1 \leftarrow v \cdot a_1$)
3: $t_1 \leftarrow a_0 \oplus^6 t_1$
4: $c_1 \leftarrow (a_0 \times^6 a_1) \bmod^6 p,\ t_0 \leftarrow (t_0 \times^6 t_1) \bmod^6 p$
5: $t_{1,0,0} \leftarrow c_{1,2,0} \ominus c_{1,2,1},\ t_{1,0,1} \leftarrow c_{1,2,0} \oplus c_{1,2,1}$
6: $t_{1,1} \leftarrow c_{1,0},\ t_{1,2} \leftarrow c_{1,1}$ (steps 6-7 $\equiv t_1 \leftarrow v \cdot c_1$)
7: $t_1 \leftarrow t_1 \oplus^6 c_1$
8: $c_0 \leftarrow t_0 \ominus^6 t_1,\ c_1 \leftarrow c_1 \oplus^6 c_1$
9: Return $c = (c_0 + c_1 w)$

6.3. Optimizing Curve Arithmetic in Miller Loop

In this section, we present our optimizations to the curve arithmetic. Remarkably, we show that the technique proposed in Section 6.2 for delaying reductions can also be applied to the point arithmetic over a quadratic extension field. Reductions can be delayed to the end of each $\mathbb{F}_{p^2}$ multiplication/squaring and then delayed further for those sums of products that allow reducing the number of reductions. Although not plentiful (given the nature of most curve arithmetic

formulas which have consecutive and redundant multiplications/squarings), there are a few places where this technique can be applied. To be consistent with other results in the literature, we only assume that double-precision addition has the cost of $2A$ and $2a$ in $\mathbb{F}_p$ and $\mathbb{F}_{p^2}$ when applying the lazy reduction technique. When this technique is not applied, we do not distinguish between single- and double-precision additions.

6.3.1. Jacobian Coordinates

The curve arithmetic in the Miller loop is traditionally performed using Jacobian coordinates [HMS08, BGM+10]. Let the point $T=(X_1,Y_1,Z_1)\in E'(\mathbb{F}_{p^2})$ be in Jacobian coordinates. The point doubling computation $2T=(X_3,Y_3,Z_3)$ and evaluation of the arising line function l at point $P=(x_P,y_P)\in E(\mathbb{F}_p)$ are traditionally performed with the following formulae [HMS08, Section 2]:

$$X_3=9X_1^4-8X_1Y_1^2,\ \ Y_3=3X_1^2(4X_1Y_1^2-X_3)-8Y_1^4,\ \ Z_3=2Y_1Z_1,$$
$$l=Z_3Z_1^2y_P-3X_1^2Z_1^2x_P+(3X_1^3-2Y_1^2). \tag{6.6}$$

An operation count of (6.6) reveals that this formula can be performed with $6m+5s+11a+4M$. We present the following revised formula that requires fewer $\mathbb{F}_{p^2}$ additions:

$$X_3=\frac{9X_1^4}{4}-2X_1Y_1^2,\ \ Y_3=\frac{3X_1^2}{2}(X_1Y_1^2-X_3)-Y_1^4,\ \ Z_3=Y_1Z_1,$$
$$l=Z_3Z_1^2y_P-(\frac{3X_1^2Z_1^2x_P}{2})+(\frac{3X_1^3}{2}-Y_1^2). \tag{6.7}$$

This doubling formula only requires $6m+5s+8a+4M$ if computed as follows ($\bar{x}_P=-x_P$ is precomputed):

$$A=3X_1^2/2,\ B=Y_1^2,\ C=X_1\cdot Y_1^2,\ D=2C,$$
$$X_3=A^2-D,\ E=C-X_3,\ Y_3=A\cdot E-B^2,\ Z_3=Y_1\cdot Z_1,\ F=Z_1^2,$$
$$l_{0,0}=Z_3\cdot F\cdot y_P,\ l_{1,0}=A\cdot F\cdot\bar{x}_P,\ l_{1,1}=A\cdot X_1-B.$$

Applying Lazy Reduction:

Let the point $T=(X_1,Y_1,Z_1)\in E'(\mathbb{F}_{p^2})$ be in Jacobian coordinates. Formula (6.7) combined with the lazy reduction technique to compute $2T=(X_3,Y_3,Z_3)$ and the tangent line evaluated at point

$P=(x_P, y_P)\in E(\mathbb{F}_p)$ has a total cost of $6m_u+5s_u+10r+10a+4M$ if computed as detailed in Algorithm 6.5.

Algorithm 6.5. Point doubling in Jacobian coordinates (cost of $6m_u+5s_u+10r+10a+4M$)

Input: $T=(X_1,Y_1,Z_1)\in E'(\mathbb{F}_{p^2})$, $P=(x_P, y_P)\in E(\mathbb{F}_p)$ and $\overline{x}_P=-x_P$

Output: $2T=(X_3,Y_3,Z_3)\in E'(\mathbb{F}_{p^2})$ and the tangent line $l\in\mathbb{F}_{p^{12}}$

1: $t_0\leftarrow X_1\otimes^2 X_1,\ t_2\leftarrow Z_1\otimes^2 Z_1$
2: $t_1\leftarrow t_0\oplus^2 t_0,\ Z_3\leftarrow Y_1\otimes^2 Z_1$
3: $t_0\leftarrow t_0\oplus^2 t_1,\ t_3\leftarrow Y_1\otimes^2 Y_1$
4: $t_0\leftarrow t_0/^2 2$
5: $t_1\leftarrow t_0\otimes^2 t_2,\ t_4\leftarrow t_0\otimes^2 X_1$
6: $l_{1,0,0}\leftarrow t_{1,0}\otimes\overline{x}_P,\ l_{1,0,1}\leftarrow t_{1,1}\otimes\overline{x}_P,\ l_{1,1}\leftarrow t_4\ominus^2 t_3,\ t_2\leftarrow Z_3\otimes^2 t_2$
7: $t_1\leftarrow t_3\otimes^2 X_1$
8: $l_{0,0,0}\leftarrow t_{2,0}\otimes y_P,\ l_{0,0,1}\leftarrow t_{2,1}\otimes y_P,\ Y_1\leftarrow t_1\oplus^2 t_1,\ X_1\leftarrow t_0\otimes^2 t_0$
9: $X_3\leftarrow X_1\ominus^2 Y_1$
10: $t_1\leftarrow t_1\ominus^2 X_3$
11: $T_0\leftarrow t_3\times^2 t_3,\ T_1\leftarrow t_0\times^2 t_1$ (Option 1, $h=2$)
12: $T_1\leftarrow T_1\ominus^2 T_0$ (Option 2)
13: $Y_3\leftarrow T_1 \bmod^2 p$
14: Return $2T=(X_3,Y_3,Z_3)$ and $l=(l_0,l_1)$

Let the points $T=(X_1,Y_1,Z_1)$ and $R=(X_2,Y_2,Z_2)\in E'(\mathbb{F}_{p^2})$ be in Jacobian coordinates. To compute $T+R=(X_3,Y_3,Z_3)$ and the tangent line l evaluated at point $P=(x_P, y_P)\in E(\mathbb{F}_p)$ we use the following addition formula:

$$\theta=Y_2Z_1^3-Y_1,\ \lambda=X_2Z_1^2-X_1,$$
$$X_3=\theta^2-2X_1\lambda^2-\lambda^3,\ Y_3=\theta(3X_1\lambda^2-\theta^2+\lambda^3)-Y_1\lambda^3,\ Z_3=Z_1\lambda,$$
$$l=Z_3y_P+(\theta\overline{x}_P)+(\theta X_2-Y_2Z_3), \tag{6.8}$$

that costs $10m_u+3s_u+11r+10a+4M$ when exploiting lazy reduction (see Algorithm 6.6).

6.3.2. Homogeneous Coordinates

Costello et al. [CLN10, Section 5] proposed the use of homogeneous coordinates to perform the curve arithmetic entirely on the twist. Their point doubling/line evaluation formula costs $2m+7s+$

Algorithm 6.6. Point addition in Jacobian coordinates (cost of $10m_u + 3s_u + 11r + 10a + 4M$)

Input: $T = (X_1, Y_1, Z_1)$ and $R = (X_2, Y_2, Z_2) \in E'(\mathbb{F}_{p^2})$, $P = (x_P, y_P) \in E(\mathbb{F}_p)$ and $\overline{x}_P = -x_P$

Output: $T + R = (X_3, Y_3, Z_3) \in E'(\mathbb{F}_{p^2})$ and the tangent line $l \in \mathbb{F}_{p^{12}}$

1: $t_1 \leftarrow Z_1 \otimes^2 Z_1$
2: $t_3 \leftarrow X_2 \otimes^2 t_1,\ t_1 \leftarrow t_1 \otimes^2 Z_1$
3: $t_3 \leftarrow t_3 \ominus^2 X_1,\ t_4 \leftarrow t_1 \otimes^2 Y_2$
4: $Z_3 \leftarrow Z_1 \otimes^2 t_3,\ t_0 \leftarrow t_4 \ominus^2 Y_1,\ t_1 \leftarrow t_3 \otimes^2 t_3$
5: $t_4 \leftarrow t_1 \otimes^2 t_3,\ X_3 \leftarrow t_0 \otimes^2 t_0$
6: $t_1 \leftarrow t_1 \otimes^2 X_1$
7: $t_3 \leftarrow t_1 \oplus^2 t_1$
8: $X_3 \leftarrow X_3 \ominus^2 t_3$
9: $X_3 \leftarrow X_3 \ominus^2 t_4$
10: $t_1 \leftarrow t_1 \ominus^2 X_3$
11: $T_0 \leftarrow t_0 \times^2 t_1,\ T_1 \leftarrow t_4 \times^2 Y_1$ (Option 1, $h = 2$)
12: $T_0 \leftarrow T_0 \ominus^2 T_1$ (Option 2)
13: $Y_3 \leftarrow T_0 \bmod^2 p,\ l_{1,0,0} \leftarrow t_{0,0} \otimes \overline{x}_P,\ l_{1,0,1} \leftarrow t_{0,1} \otimes \overline{x}_P$
14: $T_0 \leftarrow t_0 \times^2 X_2,\ T_1 \leftarrow Z_3 \times^2 Y_2$ (Option 1, $h = 2$)
15: $T_0 \leftarrow T_0 \ominus^2 T_1$ (Option 2)
16: $l_{1,1} \leftarrow T_0 \bmod^2 p,\ l_{0,0,0} \leftarrow Z_{3,0} \otimes y_P,\ l_{0,0,1} \leftarrow Z_{3,1} \otimes y_P$
17: Return $T + R = (X_3, Y_3, Z_3)$ and $l = (l_0, l_1)$

$23a + 4M + 1M_{b'}$. The twisting of point P, given in our case by $(x_p / w^2, y_p / w^3) = (\frac{x_p}{\xi} v^2, \frac{y_p}{\xi} vw)$, is eliminated by multiplying the whole line evaluation by ξ and relying on the final exponentiation to eliminate this extra factor [CLN10]. Clearly, the main drawback of this formula is the high number of additions. We present the following revised formula:

$$X_3 = \frac{X_1 Y_1}{2}(Y_1^2 - 9b'Z_1^2),\ \ Y_3 = \left[\frac{1}{2}(Y_1^2 + 9b'Z_1^2)\right] - 27b'^2 Z_1^4,\ \ Z_3 = 2Y_1^3 Z_1,$$
$$l = (-2Y_1 Z_1 y_P)vw - (3X_1^2 x_P)v^2 + \xi(3b'Z_1^2 - Y_1^2). \tag{6.9}$$

This doubling formula gives the cost of $3m + 6s + 17a + 4M + 1M_{b'} + 1M_{\xi}$. Moreover, if the parameter b' is cleverly selected as in [PSN+10], multiplication by b' can be performed with minimal number of additions and subtractions. For instance, if one fixes $b = 2$ then $b' = 2/(1+i) = 1 - i$. Accordingly, the following execution has a cost of $3m + 6s + 19a + 4M$ (note that computations for E and $l_{0,0}$ are over $\mathbb{F}_p$ and $\overline{y}_P = -y_P$ is precomputed):

$$A = X_1 \cdot Y_1/2,\ B = Y_1^2,\ C = Z_1^2,\ D = 3C,\ E_0 = D_0 + D_1,\ E_1 = D_1 - D_0,$$

$$F = 3E,\ X_3 = A\cdot(B-F),\ G = (B+F)/2,\ Y_3 = G^2 - 3E^2,$$
$$H = (Y_1+Z_1)^2 - (B+C),\ Z_3 = B\cdot H,\ I = E-B,\ l_{0,0,0} = I_0 - I_1,$$
$$l_{0,0,1} = I_0 + I_1,\ l_{1,1} = H\cdot \overline{y}_P,\ l_{0,2} = 3X_1^2\cdot x_P. \tag{6.10}$$

We point out that in practice we have observed that $m-s \approx 3a$. Hence, it is more efficient to compute X_1Y_1 directly than using $(X_1+Y_1)^2$, B and X_1^2. If this was not the case, the formula could be computed with cost $2m+7s+23a+4M$.

Applying Lazy Reduction:

Doubling formula (6.9) requires 25 $\mathbb{F}_p$ reductions (3 per $\mathbb{F}_{p^2}$ multiplication using Karatsuba, 2 per $\mathbb{F}_{p^2}$ squaring and 1 per $\mathbb{F}_p$ multiplication). First, by delaying reductions inside $\mathbb{F}_{p^2}$ arithmetic the number of reductions per multiplication goes down to only 2, with 22 reductions in total. Moreover, reductions corresponding to G^2 and $3E^2$ in Y_3 (see execution (6.10)) can be

Algorithm 6.7. Point doubling in homogeneous coordinates (cost of $3m_u + 6s_u + 8r + 22a + 4M$)

Input: $T = (X_1, Y_1, Z_1) \in E'(\mathbb{F}_{p^2})$, $P = (x_P, y_P) \in E(\mathbb{F}_p)$ and $\overline{y}_P = -y_P$

Output: $2T = (X_3, Y_3, Z_3) \in E'(\mathbb{F}_{p^2})$ and the tangent line $l \in \mathbb{F}_{p^{12}}$

1: $t_0 \leftarrow Z_1 \otimes^2 Z_1,\ t_4 \leftarrow X_1 \otimes^2 Y_1,\ t_1 \leftarrow Y_1 \otimes^2 Y_1$
2: $t_3 \leftarrow t_0 \oplus^2 t_0,\ t_4 \leftarrow t_4 /^2 2,\ t_5 \leftarrow t_0 \oplus^2 t_1$
3: $t_0 \leftarrow t_0 \oplus^2 t_3$
4: $t_{2,0} \leftarrow t_{0,0} \oplus t_{0,1},\ t_{2,1} \leftarrow t_{0,1} \ominus t_{0,0}$ $(\equiv t_2 \leftarrow b'\cdot t_2)$
5: $t_0 \leftarrow X_1 \otimes^2 X_1,\ t_3 \leftarrow t_2 \oplus^2 t_2$
6: $t_3 \leftarrow t_2 \oplus^2 t_3,\ l_{0,2} \leftarrow t_0 +^2 t_0$
7: $X_3 \leftarrow t_1 \ominus^2 t_3,\ l_{0,2} \leftarrow l_{0,2} +^2 t_0,\ t_3 \leftarrow t_1 \oplus^2 t_3$
8: $X_3 \leftarrow t_4 \otimes^2 X_3,\ t_3 \leftarrow t_3 /^2 2$
9: $T_0 \leftarrow t_3 \times^2 t_3,\ T_1 \leftarrow t_2 \times^2 t_2$ (Option 1, $h = 2$)
10: $T_2 \leftarrow T_1 +^2 T_1,\ t_3 \leftarrow Y_1 \oplus^2 Z_1$
11: $T_2 \leftarrow T_1 +^2 T_2,\ t_3 \leftarrow t_3 \otimes^2 t_3$
12: $t_3 \leftarrow t_3 \ominus^2 t_5$
13: $T_0 \leftarrow T_0 \ominus^2 T_2$ (Option 2)
14: $Y_3 \leftarrow T_0 \bmod^2 p,\ Z_3 \leftarrow t_1 \otimes^2 t_3,\ t_2 \leftarrow t_2 \ominus^2 t_1$
15: $l_{0,0,0} \leftarrow t_{2,0} \ominus t_{2,1},\ l_{0,0,1} \leftarrow t_{2,0} \oplus t_{2,1}$ $(\equiv l_{0,0} \leftarrow \xi\cdot t_2)$
16: $l_{0,2,0} \leftarrow l_{0,2,0} \otimes x_P,\ l_{0,2,1} \leftarrow l_{0,2,1} \otimes x_P$
17: $l_{1,1,0} \leftarrow t_{3,0} \otimes \overline{y}_P,\ l_{1,1,1} \leftarrow t_{3,1} \otimes \overline{y}_P$
18: Return $2T = (X_3, Y_3, Z_3)$ and $l = (l_0, l_1)$

further delayed and merged, eliminating the need of two reductions. Thus, the number of reductions is now 20 and the total cost of formula (6.9) is $3m_u + 6s_u + 8r + 22a + 4M$. The details are shown in Algorithm 6.7.

Let $T = (X_1, Y_1, Z_1)$ and $R = (X_2, Y_2, Z_2) \in E'(\mathbb{F}_{p^2})$ be points in homogeneous coordinates. To compute $T + R = (X_3, Y_3, Z_3)$ and the tangent line l evaluated at point $P = (x_P, y_P) \in \mathbb{F}_p$ we use the following addition formula:

$$\theta = Y_1 - Y_2 Z_1,\ \lambda = X_1 - X_2 Z_1,$$
$$X_3 = \lambda(\lambda^3 + Z_1\theta^2 - 2X_1\lambda^2),\ Y_3 = \theta(3X_1\lambda^2 - \lambda^3 - Z_1\theta^2) - Y_1\lambda^3,\ Z_3 = Z_1\lambda^3,$$
$$l = \lambda\overline{y}_P - (\theta x_P)v^2 + \xi(\theta X_2 - \lambda Y_2)vw, \tag{6.11}$$

that costs $11m_u + 2s_u + 11r + 12a + 4M$ when employing lazy reduction (see Alg. 6.8 below).

Algorithm 6.8. Point addition in homogeneous coordinates (cost of $11m_u + 2s_u + 11r + 12a + 4M$)

Input: $T = (X_1, Y_1, Z_1)$ and $R = (X_2, Y_2, Z_2) \in E'(\mathbb{F}_{p^2})$, $P = (x_P, y_P) \in E(\mathbb{F}_p)$ and $\overline{y}_P = -y_P$

Output: $T + R = (X_3, Y_3, Z_3) \in E'(\mathbb{F}_{p^2})$ and the tangent line $l \in \mathbb{F}_{p^{12}}$

1: $t_1 \leftarrow Z_1 \otimes^2 X_2,\ t_2 \leftarrow Z_1 \otimes^2 Y_2$
2: $t_1 \leftarrow X_1 \ominus^2 t_1,\ t_2 \leftarrow Y_1 \ominus^2 t_2$
3: $t_3 \leftarrow t_1 \otimes^2 t_1$
4: $X_3 \leftarrow t_3 \otimes^2 X_1,\ t_4 \leftarrow t_2 \otimes^2 t_2$
5: $t_3 \leftarrow t_1 \otimes^2 t_3,\ t_4 \leftarrow t_4 \otimes^2 Z_1$
6: $t_4 \leftarrow t_3 \oplus^2 t_4$
7: $t_4 \leftarrow t_4 \ominus^2 X_3$
8: $t_4 \leftarrow t_4 \ominus^2 X_3$
9: $X_3 \leftarrow X_3 \ominus^2 t_4$
10: $T_1 \leftarrow t_2 \times^2 X_3,\ T_2 \leftarrow t_3 \times^2 Y_1$ (Option 1, $h = 2$)
11: $T_2 \leftarrow T_1 \ominus^2 T_2$ (Option 2)
12: $Y_3 \leftarrow T_2 \bmod^2 p,\ X_3 \leftarrow t_1 \otimes^2 t_4,\ Z_3 \leftarrow t_3 \otimes^2 Z_1$
13: $l_{0,2,0} \leftarrow t_{2,0} \otimes x_P,\ l_{0,2,1} \leftarrow t_{2,1} \otimes x_P$
14: $l_{0,2} \leftarrow -^2 l_{0,2}$
15: $T_1 \leftarrow t_2 \times^2 X_2,\ T_2 \leftarrow t_1 \times^2 Y_2$ (Option 1, $h = 2$)
16: $T_2 \leftarrow T_1 \ominus^2 T_2$ (Option 2)
17: $t_2 \leftarrow T_1 \bmod^2 p$
18: $l_{0,0,0} \leftarrow t_{2,0} \ominus t_{2,1},\ l_{0,0,1} \leftarrow t_{2,0} \oplus t_{2,1}$ $(\equiv l_{0,0} \leftarrow \xi \cdot t_2)$
19: $l_{1,1,0} \leftarrow t_{1,0} \otimes \overline{y}_P,\ l_{1,1,1} \leftarrow t_{1,1} \otimes \overline{y}_P$
18: Return $T + R = (X_3, Y_3, Z_3)$ and $l = (l_0, l_1)$

6.4. High-Speed Pairing Implementation

In this section, we evaluate theoretically and empirically the performance gain obtained by exploiting the lazy reduction technique and improved explicit formulas. As a side effect, we demonstrate that a careful selection of curve and parameters, efficient coding and the use of other additional optimizations allow us to realize a high-speed software implementation that surpasses the best results in the literature by significant margins.

6.4.1. Optimal Ate Pairing on BN Curves

For our analysis and tests, we use the Barreto-Naehrig (BN) curve:

$$E_{BN}: \; y^2 = x^3 + 2 \tag{6.12}$$

defined over $\mathbb{F}_p$, where $p = 36u^4 + 36u^3 + 24u^2 + 6u + 1 \equiv 3 (\mathrm{mod}\, 4)$, embedding degree k = 12, prime order $n = 36u^4 + 36u^3 + 18u^2 + 6u + 1$ and $u = -(2^{62} + 2^{55} + 1) < 0 \in \mathbb{Z}$.

To implement the arithmetic over extension fields efficiently, we follow the recommendations in [IEEE08] to represent $\mathbb{F}_{p^k}$ with a tower of extensions using irreducible binomials. Accordingly, we represent $\mathbb{F}_{p^{12}}$ using the flexible towering scheme used in [DSD07, HMS08, BGM+10, PSN+10] combined with the parameters suggested by [PSN+10]:

- $\mathbb{F}_{p^2} = \mathbb{F}_p[i]/(i^2 - \beta)$, where $\beta = -1$.
- $\mathbb{F}_{p^4} = \mathbb{F}_{p^2}[s]/(s^2 - \xi)$, where $\xi = 1 + i$.
- $\mathbb{F}_{p^6} = \mathbb{F}_{p^2}[v]/(v^3 - \xi)$, where $\xi = 1 + i$.
- $\mathbb{F}_{p^{12}} = \mathbb{F}_{p^4}[t]/(t^3 - s)$ or $\mathbb{F}_{p^6}[w]/(w^2 - v)$.

As can be seen in Algorithm 6.1, the selection of $\beta = -1$, enabled by the fact that $p \equiv 3 (\mathrm{mod}\, 4)$, accelerates $\mathbb{F}_{p^2}$ arithmetic since multiplications by β can be computed as simple subtractions [PSN+10].

Although several variants of the Tate pairing are available (e.g., R-ate, optimal ate, $\mathcal{X}$-ate), our experiments reveal that they achieve very similar performance. For testing purposes, we choose to implement the optimal ate pairing given by:

$$\begin{aligned} a_{opt}: \mathrm{G}_2 \times \mathrm{G}_1 &\rightarrow \mathrm{G}_\mathrm{T} \\ (Q, P) &\rightarrow \left(f_{r,Q}(P) \cdot l_{[r]Q, \pi_p(Q)}(P) \cdot l_{[r]Q + \pi_p(Q), -\pi_p^2(Q)}(P) \right)^{\frac{p^{12}-1}{n}}, \end{aligned} \tag{6.13}$$

where $r = 6u + 2 < 0$ since $u < 0$. To accommodate the negative r, Arahna et al. [AKL+10] modifies Algorithm 2.9 with the replacement of an expensive inversion by a simple conjugation. The details are shown in Algorithm 6.9. For complete details, the reader is referred to [AKL+10, Section 5.1].

Curve arithmetic and line evaluation in Algorithm 6.9 (lines 1, 2, 5, 6, 9) were implemented with the optimized formulas in homogeneous coordinates discussed in Section 6.3.2 (Algorithm 6.7 and Algorithm 6.8). Towering arithmetic (lines 3, 5, 6, 9, 10) was optimized with the lazy reduction technique as described in Section 6.2. Following [AKL+10], for accumulating line evaluations into the Miller variable, $\mathbb{F}_{p^{12}}$ is represented using the towering $\mathbb{F}_p \rightarrow \mathbb{F}_{p^2} \rightarrow \mathbb{F}_{p^4} \rightarrow \mathbb{F}_{p^{12}}$ and a special (dense×sparse)-multiplication (called *sparse multiplication*) costing $13m_u + 6r + 61a$ is used (steps 5 and 6 of Algorithm 6.9). Aranha et al. also points that, during the first iteration of the loop, a squaring in $\mathbb{F}_{p^{12}}$ can be eliminated since the Miller variable is initialized as 1 (step 1 in Algorithm 2.9) and a special (sparse×sparse) multiplication (called *sparser multiplication*) costing $7m_u + 5r + 30a$ is used to multiply the first two line evaluations (step 3 of Algorithm 6.9). This *sparser* multiplication is also used for multiplying the two final line evaluations in step 9 of the algorithm. Final exponentiation in step 10 was implemented with the method by Scott et al. [SBC+09], in which the power $(p^{12}-1)/n$ is factored in the exponents (p^6-1), (p^2+1) and $(p^4-p^2+1)/n$. Among them, the most expensive part is the computation with the exponent $(p^4-p^2+1)/n$. In this case, the execution can be performed in the cyclotomic subgroup $\mathbb{G}_{\phi_6}(\mathbb{F}_{p^2})$, which requires, among other operations, 3 exponentiations by $|u|$. In order to speed up these exponentiations, we use the faster compressed squarings by Karabina [Kar10].

Algorithm 6.9. Modified optimal ate pairing on BN curves (generalized for $u < 0$)

Input: $P \in \mathrm{G}_1, Q \in \mathrm{G}_2, r = |6u+2| = \sum_{i=0}^{\log_2 r} r_i 2^i$

Output: $a_{opt}(Q,P)$

1: $d \leftarrow l_{Q,Q}(P),\ T \leftarrow 2Q,\ e \leftarrow 1$
2: if $r_{\lfloor \log_2 r \rfloor - 1} = 1$ then $e \leftarrow l_{T,Q}(P), T \leftarrow T + Q$
3: $f \leftarrow d \cdot e$
4: for $i = \lfloor \log_2 r \rfloor - 2$ downto 0 do
5: $\quad f \leftarrow f^2 \cdot l_{T,T}(P), T \leftarrow 2T$
6: $\quad$ if $r_i = 1$ then $f \leftarrow f \cdot l_{T,Q}(P), T \leftarrow T + Q$
7: $Q_1 \leftarrow \pi_p(P),\ Q_2 \leftarrow \pi_p^2(Q)$
8: if $u < 0$ then $T \leftarrow -T,\ f \leftarrow f^{p^6}$
9: $d \leftarrow l_{T,Q_1}(P),\ T \leftarrow T + Q_1,\ e \leftarrow l_{T,-Q_2}(P),\ T \leftarrow T - Q_2,\ f \leftarrow f \cdot (d \cdot e)$
10: $f \leftarrow f^{(p^6-1)(p^2+1)(p^4-p^2+1)/n}$
11: Return f

Remarkably, we note that these compressed squarings can be sped up by applying the generalized lazy reduction again. In total, about 8% of reductions can be eliminated per exponentiation by $|u|$. The reader is referred to Appendix C1 for complete details.

6.4.2. Operation Count

We now consider all the described improvements and state-of-the-art techniques to carry out a detailed operation count of an optimal ate pairing over BN curves using Algorithm 6.9. We aim to determine the performance gain obtained with the use of the generalized lazy reduction technique introduced in Section 6.2.

Operation counts for arithmetic performed by the Miller's algorithm when using the *generalized lazy reduction* are detailed in Table 6.2. For reference, we also include costs when using lazy reduction for $\mathbb{F}_{p^2}$ arithmetic only (referred to as *basic lazy reduction*).

First, using the parameter selection detailed in Section 6.4.1 the Miller loop in Algorithm 6.9 requires 1 negation in $\mathbb{F}_p$ to precompute the coordinate $-y_P$; 64 point doublings with line evaluations, 6 point additions with line evaluations, 2 negations, 1 p-power Frobenius and 1 p^2-power Frobenius in $E(\mathbb{F}_{p^2})$; and 1 conjugation, 66 sparse multiplications, 63 squarings, 2 sparser multiplications and 1 multiplication in $\mathbb{F}_{p^{12}}$. Thus, the cost of the Miller loop when using the generalized lazy reduction technique (ML_{GL}) is given by:

$$\begin{aligned} ML_{GL} &= 1A + 64(3m_u + 6s_u + 8r + 22a + 4M) + 6(11m_u + 2s_u + 11r + 12a + 4M) + 5a + (2m_u + 2r + 2A) \\ &\quad + (1a + 2M) + 66(13m_u + 6r + 61a) + 63(12m_u + 6r + 73a) + 2(7m_u + 5r + 30a) + (18m_u + 6r + 110a) \\ ML_{GL} &= 1906m_u + 396s_u + 1370r + 10281a + 282M + 3A. \end{aligned} \quad (6.14)$$

And the total cost of the Miller loop when using basic lazy reduction (ML_{BL}) is given by:

$$\begin{aligned} ML_{BL} &= 1A + 64(3m + 6s + 19a + 4M) + 6(11m + 2s + 10a + 4M) + 5a + (2m + 2A) + (1a + 2M) \\ &\quad + 66(13m + 36a) + 63(12m + 51a) + 2(7m + 18a) + (18m + 67a) \\ ML_{BL} &= 1906m + 396s + 6974a + 282M + 3A. \end{aligned} \quad (6.15)$$

The final exponentiation in Algorithm 6.9 requires 1 inversion, 4 conjugations, 15 multiplications, 3 u-th powers, 4 cyclotomic squarings, 5 p-power Frobenius and 3 p^2-power Frobenius in $\mathbb{F}_{p^{12}}$. Thus, the cost of the final exponentiation when using the generalized lazy reduction technique (FE_{GL}) is given by:

Table 6.2. Operation counts for arithmetic required by Miller's algorithm when using: (i) generalized lazy reduction technique; (ii) basic lazy reduction applied to $\mathbb{F}_{p^2}$ arithmetic only.

Curve Arithmetic	Operation count (generalized lazy reduction)	Operation count (basic lazy reduction)
Point doubling/line evaluation	$3m_u + 6s_u + 8r + 22a + 4M$	$3m + 6s + 8r + 19a + 4M$
Point addition/line evaluation	$11m_u + 2s_u + 11r + 12a + 4M$	$11m + 2s + 11r + 10a + 4M$
p-power Frobenius	$2m_u + 2r + 2A$	$2m + 2A$
p^2-power Frobenius	$1a + 2M$	$1a + 2M$
Negation	$1a$	$1a$
$\mathbb{F}_{p^2}$ Arithmetic	Operation count	Operation count
Add/Sub	$1a = 2A$	$1a = 2A$
Double-precision Add/Sub	$2a$	$2a$
Multiplication by ξ	$2A$	$2A$
Double-precision Mult. by ξ	$4A$	-
Conjugation	$1A$	$1A$
Reduction	$r = 2R$	$r = 2R$
Multiplication	$m = m_u + r = 3M_u + 2R + 8A$	$m = m_u + r = 3M_u + 2R + 8A$
Squaring	$s = s_u + r = 2M_u + 2R + 3A$	$s = s_u + r = 2M_u + 2R + 3A$
Inversion	$i = 1I + 2M + 2S + 2A$	$i = 1I + 2M + 2S + 2A$
$\mathbb{F}_{p^{12}}$ Arithmetic	Operation count	Operation count
Add/Sub	$6a = 12A$	$6a = 12A$
Conjugation	$3a$	$3a$
Multiplication	$18m_u + 6r + 110a$	$18m + 67a$
Sparse Multiplication	$13m_u + 6r + 61a$	$13m + 36a$
Sparser Multiplication	$7m_u + 5r + 30a$	$7m + 18a$
Squaring	$12m_u + 6r + 73a$	$12m + 51a$
Cyclotomic Squaring	$9s_u + 6r + 46a$	$6m + 61a$
Compressed Squaring	$6s_u + 4r + 31a$	$4m + 27a$
p-power Frobenius	$5m + 6A$	$5m + 6A$
p^2-power Frobenius	$10M + 2a$	$10M + 2a$
Inversion	$1i + 25m_u + 9s_u + 24r + 112a$	$1i + 25m + 9s + 82a$

$$FE_{GL} = (1i + 25m_u + 9s_u + 24r + 112a) + 4(3a) + 15(18m_u + 6r + 110a) + 3(1i + 36m_u + 372s_u + 9m + 6s + 260r + 2164a) + 4(9s_u + 6r + 46a) + 5(5m + 6A) + 3(10M + 2a)$$

$$FE_{GL} = 4i + 394m_u + 61m + 1158s_u + 21s + 906r + 8456a + 30M + 30A. \quad (6.16)$$

And the total cost of the final exponentiation when using basic lazy reduction (FE_{BL}) is given by:

$$FE_{BL} = (1i + 25m + 9s + 82a) + 4(3a) + 15(18m + 67a) + 3(1i + 293m + 6s + 1830a) + 4(6m + 61a) + 5(5m + 6A) + 3(10M + 2a)$$

$$FE_{BL} = 4i + 1223m + 27s + 6839a + 30M + 30A. \quad (6.17)$$

After adding (6.14) with (6.16) and adding (6.15) with (6.17), we obtain:

$$ML_{GL} + FE_{GL} = 4i + 2361m_u + 1575s_u + 2358r + 18737a + 30M + 30A$$
$$= 4I + 10561M_u + 5044R + 61128A. \quad (6.18)$$

$$ML_{BL} + FE_{BL} = 4i + 3129m + 423s + 13813a + 312M + 33A$$
$$= 4I + 10561M_u + 7432R + 53968A. \quad (6.19)$$

Therefore, in the case of a state-of-the-art optimal ate pairing the generalized lazy reduction technique allows us to eliminate about 32% of reductions. For instance, if we assume that $1M_u = 0.65R$ and $1A = 0.1R$ (neglecting the cost of field inversions for simplification purposes) the expected cost reduction for the whole pairing computation is approximately 9%. Obviously, this estimate is expected to grow with the ratios *R*/*A* (reduction/addition) and R/M_u (reduction/integer multiplication).

6.4.3. Implementation Results

A software implementation was developed in collaboration with Diego F. Aranha to evaluate the performance boost obtained with the introduced techniques and improved explicit formulas. To optimize carry handling and eliminate function call overheads, we followed suggestions by [BGM+10] and implemented the $\mathbb{F}_{p^2}$ arithmetic purely in Assembly. Higher-level algorithms were implemented using the C language and compiled with GCC. To obtain our cycle counts, we ran our implementations 10^4 times, averaged and approximated the results to the nearest 1000

cycles. Table 6.3 compares the timings of our *Basic* and *Optimized* implementations: the former employs lazy reduction below $\mathbb{F}_{p^2}$ only, whereas the latter is fully optimized with the lazy reduction technique applied to the whole pairing computation. Both implementations exploit faster compressed squarings and our optimized explicit formulas using homogeneous coordinates. Therefore, Table 6.3 directly illustrates the benefits of using the generalized lazy reduction technique discussed in Section 6.2. As can be seen, this technique enables in practice cost reductions between 12% and 18% on x86-64-based processors.

Table 6.3. Performance comparison of our implementations on several x86-64-based processors: (i) Basic implementation using lazy reduction below $\mathbb{F}_{p^2}$ arithmetic; (ii) Fully optimized implementation using generalized lazy reduction for the whole pairing computation. Timings are in millions of clock cycles.

Implementation	Phenom II	Cost reduct.	Core i5	Cost reduct.	Opteron	Cost reduct.	Core 2	Cost reduct.
Basic	1.777	-	2.020	-	2.005	-	2.677	-
Optimized	1.562	12%	1.688	16%	1.710	15%	2.194	18%

Table 6.4 compares our implementation results with Beuchat et al. [BGM+10], which presented the previously fastest implementation at the 128-bit security level in the literature. We remark that the tested Core i5 exhibits a microarchitecture that is equivalent to the Core i7 processor employed by [BGM+10]. To confirm this assumption, we benchmarked software by Beuchat et al. and compared the results with the ones reported in [BGM+10]. We also note that Phenom II was not considered in [BGM+10] and that we could not find a Core 2 Duo machine producing the same timings as in [BGM+10]. Hence, timings for these two architectures were measured independently by the authors using the available software.

First, observe that the *basic* implementation in Table 6.3 consistently outperforms Beuchat et al.'s results. This is due to our careful implementation using an optimal choice of parameters combined with optimized curve arithmetic in homogeneous coordinates and faster cyclotomic formulas. When lazy reduction is enabled (*optimized* implementation), pairing computation becomes faster than the best previous result by 28%-34%.

For extended benchmark results and comparisons with other previous works on different 64-bit processors, the reader is referred to our online database [Lon10b].

Table 6.4. Performance comparison of state-of-the-art pairing implementations on several x86-64-based processors. Timings are in clock cycles.

Operation	Beuchat et al. [BGM+10]			
	Phenom II (1)	Core i7 (2)	Opteron (3)	Core 2 Duo (4)
Multiplication in $\mathbb{F}_{p^2}$	440	435	443	590
Squaring in $\mathbb{F}_{p^2}$	353	342	355	479
Miller Loop	1,338,000	1,330,000	1,360,000	1,781,000
Final Exponentiation	1,020,000	1,000,000	1,040,000	1,370,000
Optimal Ate Pairing	2,358,000	2,330,000	2,400,000	3,151,000
Operation	This work			
	Phenom II (1)	Core i5 (5)	Opteron (6)	Core 2 Duo (4)
Multiplication in $\mathbb{F}_{p^2}$	368	412	390	560
Squaring in $\mathbb{F}_{p^2}$	288	328	295	451
Miller Loop	898,000	990,000	988,000	1,275,000
Final Exponentiation	664,000	713,000	722,000	919,000
Optimal Ate Pairing	1,562,000	1,688,000	1,710,000	2,194,000
Improvement	34%	28%	29%	30%

(1) On a 3.0GHz AMD Phenom II X4 940.
(2) On a 2.8GHz Intel Core i7 860.
(3) On a 2.3GHz AMD Opteron 2376.
(4) On a 2.66GHz Intel Core 2 Duo E6750.
(5) On a 2.53GHz Intel Core i5 M540.
(6) On a 2.2GHz AMD Opteron 275.

6.5. Conclusions

In this chapter, we have proposed efficient methods and improved explicit formulas that speed up significantly the computation of pairings on ordinary curves over prime fields. Most remarkably, the introduced generalized lazy reduction technique is shown to apply to every computation involving tower field operations found in the Miller loop and final exponentiation, including the recently proposed compressed squarings by [Kar10] (see Appendix C1).

After discussing relevant previous work in §6.1, we introduced the generalized lazy reduction technique in the context of tower extension fields in §6.2. We described a methodology that relies on the careful selection of the field size to keep intermediate results under Montgomery boundaries with the objective of reducing costs of additions/subtractions and maximizing the use of operations without carry checks. Moreover, we illustrated the efficient realization of these techniques with the popular tower $\mathbb{F}_p \rightarrow \mathbb{F}_{p^2} \rightarrow \mathbb{F}_{p^6} \rightarrow \mathbb{F}_{p^{12}}$, detailing the improved explicit

formulas for multiplication and squaring in $\mathbb{F}_{p^2}$, $\mathbb{F}_{p^6}$ and $\mathbb{F}_{p^{12}}$.

In §6.3, we presented our optimizations to doubling/line evaluation and addition/line evaluation formulas using Jacobian and homogeneous coordinates. The revised formulas require fewer additions and "small" operations, which are not negligible in certain platforms. Furthermore, we also applied the generalized lazy reduction technique to the curve arithmetic and explicitly stated the new formulas with minimal number of reductions.

In §6.4, we evaluated the new techniques and explicit formulas on a state-of-the-art implementation of the optimal ate pairing on BN curves at the 128-bit security level. We carried out a detailed operation count and determined that the generalized reduction technique allows us to eliminate about 32% of reductions, which represents (under certain assumptions) an estimated cost reduction of about 9% for the whole pairing computation. This improvement strongly relies on the typically large gap between reduction and addition costs, so the cost reduction is expected to grow with the *R*/*A* ratio. This analysis was confirmed in practice with a high-speed software implementation that was intensively code optimized and includes state-of-the-art techniques such as the fast compressed squaring formulas and efficient decompression by [Kar10]. We reported improvements between 12% and 18% on different 64-bit platforms when using our method. These results surpass the expected theoretical estimate since they include our methodology to optimize carry handling and maximize the number of operations without carry checks (see Section 6.2.1). As a side effect, we reported the fastest pairing implementation on x86-64-based processors with improvements ranging between 28% and 34% in comparison with the previously best results due to Beuchat et al. [BGM+10]. In particular, we reported a pairing computation in ~0.5msec. on a 3.0GHz AMD Phenom II X4 processor.

Chapter 7

Conclusions

In the last few years, intense research has been focused on the efficient computation of elliptic curve and pairing primitives to enable their realization in the plethora of potential applications and emerging platforms of the new millennium. This thesis has focused on *devising efficient methods and formulas for enabling high-speed elliptic curve and pairing-based cryptography over fields of large prime characteristic*. These results have a practical impact in the performance of cryptographic protocols and schemes based on elliptic curves and pairings. Most remarkably, a careful selection of state-of-the-art algorithms has led to the realization of *record-breaking* implementations in software. For instance, these results may directly increase the number of secure transaction requests per second that can be processed by a Web server in an Internet-based application such as e-banking or e-commerce. This could potentially lead to savings in hardware costs for corporations, to more Web-based content being protected and to reduced waiting times during online transactions for consumers, among other benefits.

A more detailed description of the contributions of this thesis follow in §7.1. Possible future research directions are described in §7.2.

7.1. Summary of Contributions

In Chapter 2 a summary of fundamental concepts of ECC and Pairing-based Cryptography was provided. Also, some advanced research topics regarding special curves and the GLS method

were described.

Chapter 3 introduced two new schemes for precomputing points. The LM Scheme, which is intended for tables of form d_iP on standard curves using Jacobian coordinates, was adapted to the case using only one inversion (case 2) and to the case without inversions (case 1). For case 2, two variants were proposed with slightly different memory requirements and speeds, case 2a and case 2b. It was shown that the new method achieves the lowest costs in the literature when using an *optimal* number of precomputations. For instance, LM Scheme, case 2b, has a cost of $1I + (9L)M + (2L+6)S$ with L non-trivial points, which is the lowest in the literature among methods using one inversion only. The cost formulas for the different variants were derived (see proofs in Appendices A1 and A2). On the other hand, the LG Scheme, which is based on the proposed idea of *conjugate additions in projective coordinates*, was shown to apply to different curve forms and types of scalar multiplication. Conjugate addition formulas were derived for $\mathcal{J}$, $\mathcal{JQ}^e$ and $\mathcal{IE}$ coordinates (see Appendix A3). Moreover, an efficient method combining the LM and LG Schemes was proposed for the case of multiple scalar multiplication on standard curves using $\mathcal{J}$. The generic cost formulas for single and multiple scalar multiplications were derived (see proofs in Appendices A5 and A6), as well as the cost formulas of the optimized schemes for $\mathcal{J}$, $\mathcal{JQ}^e$ and $\mathcal{IE}$ coordinates. Finally, an extensive comparative analysis of different pre-computations methods for different scenarios, memory requirements and security levels was carried out to determine the most efficient scheme for each case when using $\mathcal{J}$, $\mathcal{JQ}^e$ and $\mathcal{IE}$ coordinates. In general, it was shown that for the great majority of cases the proposed schemes achieve the best performance. Refer to §3.4 for complete details. Finally, potential applications for the use of conjugate additions were described (see §3.5). The outcomes of this chapter were exploited for speeding up further scalar multiplication in Chapters 4 and 5.

Chapter 4 was about efficient multibase representations for scalar multiplication and how efficient these methods are in different scenarios. First, a taxonomy and comparative analysis of the various double- and multi-base methods for scalar multiplication were discussed. Then, the theoretical analysis of the multibase NAF (*mb*NAF) method and its windowed variant, *wmb*NAF, were developed. Our methods were modeled using Markov chains and formulas for estimating the average zero and nonzero densities for cases with bases {2,3} and {2,3,5} were derived. Then, the "fractional" windows recoding was applied to the setting of *wmb*NAF to solve the problem of restricted number of precomputations imposed by standard windows. The new method, denoted by Frac-*wmb*NAF, allows a flexible number of precomputations in the execution of scalar multiplication, which makes it ideal for applications with restricted memory. The method was also analyzed theoretically using Markov chains for the case with bases {2,3}. Furthermore, a new methodology based on the *operation cost per bit* to derive efficient multibase algorithms was introduced. The optimized algorithms were implemented in Matlab to perform an extensive comparison for computing scalar multiplication when using $\mathcal{J}$, $\mathcal{JQ}^e$ and $\mathcal{IE}$

coordinates. The cases with bases {2,3} and {2,3,5} using (Frac-*w*)*mb*NAF and the refined multibase chains were compared with the performance of standard NAF-based methods and the most efficient double-base methods in the literature. For proposed and standard NAF methods, the best precomputation scheme available for each case was applied (using results from Chapter 3). The conclusion was that, currently, the proposed refined multibase chains achieve the lowest costs found in the literature among methods without precomputations, for all curve forms under analysis. For instance, using bases {2,3,5} and {2,3} for n = 160 bits we can perform a scalar multiplication with costs of only 1451M (field multiplications) and 1351M in Jacobian and inverted Edwards coordinates, respectively. With $\mathcal{JQ}^e$, that cost can be as low as 1261M using bases {2,3,5}. These results provide cost reductions between 7%-10% in comparison with NAF. Similar results were attained by the refined multibase chains using an optimal number of precomputations, although in this case the gain was only 1%-3% in comparison with (Frac)-*w*NAF (see §4.5 for complete details). A relevant comparison with the fastest curves using standard radix-2 methods followed. In conclusion, "slower" curves that can advantageously exploit multibase chains may become competitive with the "fastest" curves using radix-2 methods when curve parameters are suitably chosen and no precomputations are allowed. Finally, a discussion of potential applications and variants of the proposed methods was included, as well as a critical look at the practical implications of double- and multi-base number systems in the computation of scalar multiplication (see §4.6). In conclusion, the use of multibases was recommended for memory-constrained devices and when the conversion step (if expensive) can be performed off-line. For non-constrained devices, it was shown that the gain may be negligible and that faster curves without exploiting multibases are available. These conclusions were confirmed by tests on real x86-64-based implementations in §5.6.4, subsection "Timings using Multibase Methods".

Chapter 5 studied and brought together most efficient algorithms for the field, point and scalar arithmetic levels with the objective of achieving high-speed implementations of ECC on x86-64 processors. Optimizations at different levels were carefully tuned for the targeted architectures. First, incomplete reduction and branchless arithmetic were optimally combined for suitably chosen pseudo-Mersenne primes for achieving efficient arithmetic in $\mathbb{F}_p$. Dependencies between consecutive field operations were found to degrade the performance on the targeted processors by stalling the pipeline. The rescheduling and merging of field operations and the merging of point operations were proposed to minimize this problem. These techniques also reduce the number of function calls and memory accesses. Explicit point formulas for the relevant cases of $\mathcal{J}$ and $\mathcal{E}/\mathcal{E}^e$ over $\mathbb{F}_p$ and $\mathbb{F}_{p^2}$ were optimized by reducing the number of "small" operations and by applying the techniques aforementioned (see Appendices B1 and B2). By combining all optimized formulas with state-of-the-art algorithms, including the use of the LM precomputation scheme (see §5.6.1 and §5.6.2 for further details), we presented *two*

traditional and *two* GLS-based implementations using $\mathcal{J}$ or $\mathcal{E}/\mathcal{E}^e$ coordinates at the 128-bit security level. The various tests throughout the chapter as well as the benchmark results for full point multiplication were discussed for at least one x86-64 processor representative from the notebook, desktop and server computing classes. Presented implementations set *new speed records* and were shown to achieve **up to 34% of cost reduction** in comparison with best previous results. For instance, we reported a point multiplication computation in about **60μsec.** on a 3.0GHz AMD Phenom II X4 processor.

Finally, Chapter 6 studied and brought together most efficient algorithms for computing pairings with the objective of enabling high-speed implementations on x86-64 processors. First, the well-known technique of lazy reduction was generalized to the whole pairing arithmetic including towering and curve arithmetic. By carrying out a detailed operation count, this technique was shown to eliminate at least 32% of the total number of reductions in a state-of-the-art implementation of the optimal ate pairing over a BN curve at the 128-bit security level. Furthermore, for dealing with more costly higher-precision additions required by lazy reduction, a flexible methodology that keeps intermediate values under Montgomery reduction boundaries maximizing the use of operations without carry checks was developed. Optimized formulas were derived for the case using the tower $\mathbb{F}_p \rightarrow \mathbb{F}_{p^2} \rightarrow \mathbb{F}_{p^6} \rightarrow \mathbb{F}_{p^{12}}$ and for the new compressed squarings by [Kar10] (see §6.2.2 and Appendix C1). Following the approach detailed in Section 5.4, formulas for point doubling and addition in Jacobian and homogeneous coordinates were carefully optimized by eliminating several commonly neglected operations that are not inexpensive on modern 64-bit platforms (see §6.3). Finally, the significant savings obtained by the new techniques were illustrated with a high-speed implementation of the optimal ate pairing over a BN curve at the 128-bit security level. By combining our techniques with other state-of-the-art methods, the presented implementation set *new speed records* and was shown to achieve **up to 34% of cost reduction** on x86-64 processors in comparison with the best results in the literature. For instance, we reported a pairing computation in about **half a millisecond** on a 3.0GHz AMD Phenom II X4 processor.

7.2. Future Work

New potential research directions have arisen from the outcomes of this dissertation. We summarize them below:

Precomputations for other special curves and settings. In particular, for the efficient Twisted Edwards curve using $\mathcal{E}/\mathcal{E}^e$ or extended Jacobi quartics using homogeneous/extended homogeneous coordinates it is still unknown if other precomputation schemes with higher efficiency than the traditional scheme using $P \rightarrow 3P \rightarrow 5P \rightarrow \ldots \rightarrow mP$ exist. Further

research could focus on the development of improved schemes for these systems. Also, in §3.5 it was observed that conjugate additions can be derived for formulas over $\mathbb{F}_{2^m}$. The application of LG-like precomputation schemes to this setting requires further analysis.

More composite formulas and efficient conversion to multibase. In §4.6.1, it was argued that the main obstacle that opposes to the use of multiple bases in a wide range of applications is the computing cost of conversion from binary to multibase. Further research is needed to improve the implementation of conversion algorithms on different platforms. This effort can be complemented by the development of efficient tripling and quintupling formulas for other coordinate systems such as $\mathcal{E}/\mathcal{E}^e$ where radix-2 methods are still more efficient.

Implementation on constrained devices. Following the results and analysis in §4.5 and §4.6.1, the use of multibase methods is more promising for devices with constrained memory resources in which the gain is maximal in terms of speed. However, these devices are usually limited in terms of power. Further investigation supported with implementations is required for assessing the practical impact of using multibase methods in these platforms with such a constraint.

Analysis on other platforms; improving ECC over binary fields, HECC. Several software techniques and optimizations were proposed for elliptic curve point multiplication over $\mathbb{F}_p$ and $\mathbb{F}_{p^2}$ in Chapter 5. The analysis and implementations targeted x86-64 processors. In many cases, the proposed techniques and optimized formulas are generic and further study could be devoted to test them on different platforms, e.g., embedded devices with 32-bit and 8-bit microarchitectures. Moreover, further research can be focused on applying similar methods to the case over $\mathbb{F}_{2^m}$. For instance, it would be interesting to analyze whether data dependencies degrade performance of field operations and if similar countermeasures also apply. In fact, further study could analyze the application of these methods to other settings such as Hyperelliptic Curve Cryptosystems.

Generalized lazy reduction on other platforms. This technique was shown to reduce significantly the computing cost of pairings on various x86-64-based processors. Practical implementation of the technique in Field Programmable Gate Arrays (FPGAs), 32-bit embedded devices or microcontrollers with 8-bit architectures would be highly valuable. In certain cases, the gain is expected to grow even further as the ratio multiplication/addition is usually larger on smaller devices in which embedded multipliers are much less powerful.

Appendix A

A1 Pseudocode of the LM Precomputation Scheme

In this section, we present the pseudocode of the LM Scheme described in Section 3.2.

Lemma A.1. Algorithm A.1, that computes the initial doubling (3.3) of *Step 1* (see Section 3.2.1), costs $1M + 5S$ and requires 6 temporary registers.

Algorithm A.1. Point doubling $2\mathcal{A} \rightarrow \mathcal{J}$, $E: y^2 = x^3 + ax + b$

Input: point $P = (x_1, y_1)$ in $E(\mathbb{F}_p)$, $T_1 \leftarrow x_1$, $T_3 \leftarrow y_1$, curve parameter a
Output: point $2P = (X_2 : Y_2 : Z_2)$ and $P^{(1)} = (X_1^{(1)}, Y_1^{(1)}, Z_1^{(1)}) \equiv (x_1, y_1, 1)$

1:	$T_2 = T_3^2$	$\{y_1^2\}$
2:	$T_4 = T_1 + T_2$	$\{x_1 + y_1^2\}$
3:	$T_4 = T_4^2$	$\{(x_1 + y_1^2)^2\}$
4:	$T_5 = T_2^2$	$\{Y_1^{(1)} = y_1^4\}$
5:	$T_1 = T_1^2$	$\{x_1^2\}$
6:	$T_4 = T_4 - T_1$	$\{(x_1 + y_1^2)^2 - x_1^2\}$
7:	$T_6 = T_4 - T_2$	$\{2\beta = (x_1 + y_1^2)^2 - x_1^2 - y_1^4\}$
8:	$T_4 = T_6 / 2$	$\{X_1^{(1)} = \beta = x_1 y_1^2\}$
9:	$T_1 = 3T_1$	$\{3x_1^2\}$
10:	$T_2 = T_1 + a$	$\{3x_1^2 + a\}$
11:	$T_2 = T_2 / 2$	$\{\alpha = (3x_1^2 + a)/2\}$
12:	$T_1 = T_2^2$	$\{\alpha^2\}$
13:	$T_1 = T_1 - T_6$	$\{X_2 = \alpha^2 - 2\beta\}$
14:	$T_6 = T_4 - T_1$	$\{\beta - X_2\}$
15:	$T_2 = T_2 \times T_6$	$\{\alpha(\beta - X_2)\}$
16:	$T_2 = T_2 - T_5$	$\{Y_2 = \alpha(\beta - X_2) - y_1^4\}$
17:	Return $(T_1, T_2, T_3, T_4, T_5) = (X_2, Y_2, Z_2, X_1^{(1)}, Y_1^{(1)})$	

Lemma A.2. Algorithm A.2, that computes the first addition $2P + P$ in sequence (3.2) using $\mathrm{ADD}_{\mathrm{Co-Z}}$, costs $5M + 2S$ and requires 6 temporary registers if the precomputed table contains only one point. Otherwise, Algorithm A.2 requires 6 temporary registers for calculations plus 2 extra registers to store the (X, Y) coordinates of $3P$. To adapt Algorithm A.2 to case 1, it should also store the Z coordinate of $3P$ in register Z_3.

Algorithm A.2. Special addition with identical Z coordinate $\mathcal{J}+\mathcal{J}\rightarrow\mathcal{J}$, $E: y^2=x^3+ax+b$

Input: points $2P=(X_2:Y_2:Z_2)$ and $P^{(1)}=(X_1^{(1)},Y_1^{(1)},Z_1^{(1)})$ in $E(\mathbb{F}_p)$,
$T_1\leftarrow X_2,\ T_2\leftarrow Y_2,\ T_3\leftarrow Z_2,\ T_4\leftarrow X_1^{(1)},\ T_5\leftarrow Y_1^{(1)}$

Output: point $3P=2P+P_1^{(1)}=(X_3:Y_3:Z_3)$

1:	$T_6=T_4-T_1$	$\{X_1^{(1)}-X_2\}$
2:	$T_3=T_3\times T_6$	$\{Z_3=Z_2(X_1^{(1)}-X_2)\}$
3:	$T_4=T_6^2$	$\{(X_1^{(1)}-X_2)^2\}$
4:	$T_6=T_4\times T_6$	$\{(X_1^{(1)}-X_2)^3\}$
5:	$T_4=T_1\times T_4$	$\{X_2^{(1)}=X_2(X_1^{(1)}-X_2)^2\}$
6:	$T_1=2T_4$	$\{2X_2(X_1^{(1)}-X_2)^2\}$
7:	$T_1=T_1+T_6$	$\{(X_1^{(1)}-X_2)^3+2X_2(X_1^{(1)}-X_2)^2\}$
8:	$T_6=T_2\times T_6$	$\{Y_2^{(1)}=Y_2(X_1^{(1)}-X_2)^3\}$
9:	$T_2=T_5-T_2$	$\{Y_1^{(1)}-Y_2\}$
10:	$T_5=T_2^2$	$\{(Y_1^{(1)}-Y_2)^2\}$
11:	$T_1=T_5-T_1$	$\{X_3=(Y_1^{(1)}-Y_2)^2-(X_1^{(1)}-X_2)^3-2X_2(X_1^{(1)}-X_2)^2\}$
12:	$T_5=T_4-T_1$	$\{X_2(X_1^{(1)}-X_2)^2-X_3\}$
13:	$T_5=T_2\times T_5$	$\{(Y_1^{(1)}-Y_2)[X_2(X_1^{(1)}-X_2)^2-X_3]\}$
14:	$T_2=T_5-T_6$	$\{Y_3=(Y_1^{(1)}-Y_2)[X_2(X_1^{(1)}-X_2)^2-X_3]-Y_2(X_1^{(1)}-X_2)^3\}$
15:	$T_5=T_6$	$\{Y_2^{(1)}\}$
16:	If $m>3$ then:	
17:	$X_3=T_1$	
18:	$Y_3=T_2$	
19:	Return $(T_1,T_2,T_3,T_4,T_5,X_3,Y_3)=(X_3,Y_3,Z_3,X_2^{(1)},Y_2^{(1)},X_3,Y_3)$	

Lemma A.3. Algorithm A.3, that computes following additions in sequence (3.2) using $\mathrm{ADD}_{\mathrm{Co-}Z}$ operations, costs $5M + 2S$ per extra point, requires 6 temporary registers for calculations and 3 (4) extra registers per each point for case 2a (case 2b) to store the values X,Y,A (X,Y,B,C). In the last iteration the memory requirement is reduced by storing values X,Y (X,Y,B) in temporary registers. To adapt Algorithm A.3 to case 1, one should execute the steps that correspond to case 2a except that, instead of values $A_{(i+3)/2}$, one should store Z coordinates of points iP.

Algorithm A.3. Special addition with identical Z coordinate $\mathcal{J} + \mathcal{J} \rightarrow \mathcal{J}$, $E: y^2 = x^3 + ax + b$

Input: $2P^{(1)} = (X_2^{(1)}, Y_2^{(1)}, Z_2^{(1)})$ and $3P = (X_3 : Y_3 : Z_3)$,
$T_1 \leftarrow X_3$, $T_2 \leftarrow Y_3$, $T_3 \leftarrow Z_2^{(1)}$, $T_4 \leftarrow X_2^{(1)}$, $T_5 \leftarrow Y_2^{(1)}$

Output: points $iP = 2P^{((i-3)/2)} + (i-2)P = (X_{(i+3)/2}, Y_{(i+3)/2}, Z_{(i+3)/2})$, for $i = 5$ to m, i odd

	LM Scheme, case 2a:	LM Scheme, case 2b:
1:	For $i = 5$ to m do (i odd)	For $i = 5$ to m do (i odd)
2:	$A_{(i+3)/2} = T_1 - T_4$ $\{A_{(i+3)/2} = X_{(i+1)/2} - X_2^{((i-3)/2)}\}$	If $i \neq m$ then:
3:	$T_3 = A_{(i+3)/2} \times T_3$ $\{Z_{(i+3)/2}\}$	$T_1 = T_1 - T_4$ $\{X_{(i+1)/2} - X_2^{((i-3)/2)}\}$
4:	$T_1 = A_{(i+3)/2}^2$ $\{A_{(i+3)/2}^2\}$	$T_3 = T_1 \times T_3$ $\{Z_{(i+3)/2}\}$
5:	$T_4 = T_1 \times T_4$ $\{X_2^{((i-1)/2)} = X_2^{((i-3)/2)} A_{(i+3)/2}^2\}$	$B_{(i+3)/2} = T_1^2$ $\{B_{(i+3)/2} = (X_{(i+1)/2} - X_2^{((i-3)/2)})^2\}$
6:	$T_1 = T_1 \times A_{(i+3)/2}$ $\{A_{(i+3)/2}^3\}$	$C_{(i+3)/2} = T_1 \times B_{(i+3)/2}$ $\{C_{(i+3)/2} = (X_{(i+1)/2} - X_2^{((i-3)/2)})^3\}$
7:	$T_2 = T_2 - T_5$ $\{Y_{(i+1)/2} - Y_2^{((i-3)/2)}\}$	$T_4 = T_4 \times B_{(i+3)/2}$ $\{X_2^{((i-1)/2)} = X_2^{((i-3)/2)} B_{(i+3)/2}\}$
8:	$T_5 = T_1 \times T_5$ $\{Y_2^{((i-1)/2)} = Y_2^{((i-3)/2)} A_{(i+3)/2}^3\}$	$T_1 = 2T_4$ $\{2X_2^{((i-3)/2)} B_{(i+3)/2}\}$
9:	$T_6 = 2T_4$ $\{2X_2^{((i-3)/2)} A_{(i+3)/2}^2\}$	$T_1 = T_1 + C_{(i+3)/2}$ $\{C_{(i+3)/2} + 2X_2^{((i-3)/2)} B_{(i+3)/2}\}$
10:	$T_1 = T_1 + T_6$ $\{A_{(i+3)/2}^3 + 2X_2^{((i-3)/2)} A_{(i+3)/2}^2\}$	$T_2 = T_2 - T_5$ $\{Y_{(i+1)/2} - Y_2^{((i-3)/2)}\}$
11:	$T_6 = T_2^2$ $\{(Y_{(i+1)/2} - Y_2^{((i-3)/2)})^2\}$	$T_5 = T_5 \times C_{(i+3)/2}$ $\{Y_2^{((i-1)/2)} = Y_2^{((i-3)/2)} C_{(i+3)/2}\}$
12:	$T_1 = T_6 - T_1$ $\{X_{(i+3)/2}\}$	$T_6 = T_2^2$ $\{(Y_{(i+1)/2} - Y_2^{((i-3)/2)})^2\}$
13:	$T_6 = T_4 - T_1$ $\{s = X_2^{((i-3)/2)} A_{(i+3)/2}^2 - X_{(i+3)/2}\}$	$T_1 = T_6 - T_1$ $\{X_{(i+3)/2}\}$
14:	$T_2 = T_2 \times T_6$ $\{s \cdot (Y_{(i+1)/2} - Y_2^{((i-3)/2)})\}$	$T_6 = T_4 - T_1$ $\{s = X_2^{((i-3)/2)} B_{(i+3)/2} - X_{(i+3)/2}\}$
15:	$T_2 = T_2 - T_5$ $\{Y_{(i+3)/2}\}$	$T_2 = T_2 \times T_6$ $\{s \cdot (Y_{(i+1)/2} - Y_2^{((i-3)/2)})\}$
16:	If $i \neq m$ then:	$T_2 = T_2 - T_5$ $\{Y_{(i+3)/2}\}$
17:	$X_{(i+3)/2} = T_1$	$X_{(i+3)/2} = T_1$
18:	$Y_{(i+3)/2} = T_2$	$Y_{(i+3)/2} = T_2$
19:	Return $(T_1, T_2, T_3, A_{(i+3)/2}, X_{(i+3)/2}, Y_{(i+3)/2})$	Else:
20:		$T_1 = T_1 - T_4$ $\{X_{(i+1)/2} - X_2^{((i-3)/2)}\}$
21:		$T_3 = T_1 \times T_3$ $\{Z_{(i+3)/2}\}$
22:		$T_6 = T_1^2$ $\{B_{(i+3)/2} = (X_{(i+1)/2} - X_2^{((i-3)/2)})^2\}$
23:		$C_{(i+3)/2} = T_1 \times T_6$ $\{C_{(i+3)/2} = (X_{(i+1)/2} - X_2^{((i-3)/2)})^3\}$
24:		$T_2 = T_2 - T_5$ $\{Y_{(i+1)/2} - Y_2^{((i-3)/2)}\}$
25:		$T_5 = T_5 \times C_{(i+3)/2}$ $\{Y_2^{((i-3)/2)} C_{(i+3)/2}\}$
26:		$T_4 = T_4 \times T_6$ $\{X_2^{((i-3)/2)} B_{(i+3)/2}\}$
27:		$T_4 = 2T_4$ $\{2X_2^{((i-3)/2)} B_{(i+3)/2}\}$
28:		$T_1 = T_2^2$ $\{(Y_{(i+1)/2} - Y_2^{((i-3)/2)})^2\}$
29:		$T_1 = T_1 - C_{(i+3)/2}$ $\{(Y_{(i+1)/2} - Y_2^{((i-3)/2)})^2 - C_{(i+3)/2}\}$
30:		$T_1 = T_1 - T_4$ $\{X_{(i+3)/2}\}$
31:		$T_4 = T_4 / 2$ $\{X_2^{((i-3)/2)} B_{(i+3)/2}\}$
32:		$T_4 = T_4 - T_1$ $\{s = X_2^{((i-3)/2)} B_{(i+3)/2} - X_{(i+3)/2}\}$
33:		$T_2 = T_2 \times T_4$ $\{s \cdot (Y_{(i+1)/2} - Y_2^{((i-3)/2)})\}$
34:		$T_2 = T_2 - T_5$ $\{Y_{(i+3)/2}\}$
35:		Return $(T_1, T_2, T_3, T_6, B_{(i+3)/2}, C_{(i+3)/2}, X_{(i+3)/2}, Y_{(i+3)/2})$

Lemma A.4. Algorithm A.4, that computes the modified Montgomery's method corresponding to *Step 2* (see Section 3.2.1), costs $1I + (3M + 1S) + (4M + 1S)(L - 1)$ and $1I + (3M + 1S) + 4(L - 1)M$ for cases 2a and 2b, respectively, and requires 4 temporary registers for calculations and storage for the affine coordinates (x, y) of $(L - 1)$ precomputed points. In addition, case 2a requires $(L - 1)$ registers for values A_j, and case 2b requires $2(L - 1)$ registers for values (B_j, C_j). This step is not executed in case 1.

Algorithm A.4. Modified Montgomery' simultaneous inversion method, $E: y^2 = x^3 + ax + b$

Input: $T_1, T_2, T_3, T_6 = B_{(m+3)/2}, A_{(j+3)/2}, B_{(j+3)/2}, C_{(j+3)/2}$, for $j = 5$ to m, j odd
$X_{(i+3)/2}, Y_{(i+3)/2}$, for $i = 3$ to m, i odd

Output: points $iP = (x_{(i+3)/2}, y_{(i+3)/2})$ for $i = 3$ to m, i odd

	LM Scheme, case 2a:		LM Scheme, case 2b:	
1:	$T_3 = T_3^{-1}$	$\{Z_{(m+3)/2}^{-1}\}$	$B_{(m+3)/2} = T_6$	
2:	$T_4 = T_3^2$	$\{Z_{(m+3)/2}^{-2}\}$	$T_3 = T_3^{-1}$	$\{Z_{(m+3)/2}^{-1}\}$
3:	$X_{(m+3)/2} = T_1 \times T_4$	$\{x_{(m+3)/2}\}$	$T_4 = T_3^2$	$\{Z_{(m+3)/2}^{-2}\}$
4:	$T_4 = T_3 \times T_4$	$\{Z_{(m+3)/2}^{-3}\}$	$X_{(m+3)/2} = T_1 \times T_4$	$\{x_{(m+3)/2}\}$
5:	$Y_{(m+3)/2} = T_2 \times T_4$	$\{y_{(m+3)/2}\}$	$T_3 = T_3 \times T_4$	$\{Z_{(m+3)/2}^{-3}\}$
6:	For $i = (m-2)$ downto 3 do (i odd)		$Y_{(m+3)/2} = T_2 \times T_3$	$\{y_{(m+3)/2}\}$
7:	$T_3 = T_3 \times A_{(i+5)/2}$	$\{Z_{(i+3)/2}^{-1}\}$	For $i = (m-2)$ downto 3 do (i odd)	
8:	$T_4 = T_3^2$	$\{Z_{(i+3)/2}^{-2}\}$	$T_4 = T_4 \times B_{(i+5)/2}$	$\{Z_{(i+3)/2}^{-2}\}$
9:	$X_{(i+3)/2} = X_{(i+3)/2} \times T_4$	$\{x_{(i+3)/2}\}$	$T_3 = T_3 \times C_{(i+5)/2}$	$\{Z_{(i+3)/2}^{-3}\}$
10:	$T_4 = T_3 \times T_4$	$\{Z_{(i+3)/2}^{-3}\}$	$X_{(i+3)/2} = X_{(i+3)/2} \times T_4$	$\{x_{(i+3)/2}\}$
11:	$Y_{(i+3)/2} = Y_{(i+3)/2} \times T_4$	$\{y_{(i+3)/2}\}$	$Y_{(i+3)/2} = Y_{(i+3)/2} \times T_3$	$\{y_{(i+3)/2}\}$
12:	Return $(X_{(i+3)/2}, Y_{(i+3)/2})$		Return $(X_{(i+3)/2}, Y_{(i+3)/2})$	

A2 Cost Analysis of the LM Precomputation Scheme

Theorem A.1. The LM Scheme, case 1, has the following cost:

$$\text{Cost}_{\text{LM Scheme, case 1}} = (6L+1)M + (3L+5)S,$$

and requires $(3L + 6)$ registers, where L is the number of non-trivial points in the precomputed table d_iP. The requirement increases to $(5L + 6)$ if values Z_i^2 and Z_i^3 are also stored in order to use the addition (or doubling-addition) with stored values during evaluation.

Proof: Following Lemmas A.1-A.3, Algorithms A.1, A.2 and A.3 cost $1M + 5S$, $5M + 2S$ and $(5M + 2S)(L-1)$, respectively. Also, precomputing values Z_i^2, Z_i^3 (to enable the use of ADD or DBLADD with store values during the evaluation stage) costs $(1M + 1S)L$. By adding these values we obtain the cost of the LM Scheme, case 1, above. In terms of memory, this method only requires 6 temporary registers during the execution of Algorithms A.1, A.2 and A.3 plus 3 registers to store the $(X:Y:Z)$ coordinates of each precomputed point. That makes a total requirement of $3L + 6$ registers. If the pair Z_i^2 / Z_i^3 is also stored per point, the total requirement increases to $5L + 6$. □

Theorem A.2. The LM Scheme, case 2a, has the following cost:

$$\text{Cost}_{\text{LM Scheme, case 2a}} = 1I + (9L)M + (3L+5)S,$$

and requires $(3L + 3)$ registers.

Proof: Following Lemmas A.1-A.3, Algorithms A.1, A.2 and A.3 cost $1M + 5S$, $5M + 2S$ and $(5M + 2S)(L - 1)$, respectively. According to Lemma A.4, Algorithm A.4 costs $1I + (3M + 1S) + (4M + 1S)(L - 1)$. By adding these values, we obtain the cost of the LM Scheme, case 2a, above. Regarding memory requirements, Algorithm A.1 needs 6 temporary registers $T_1,\ldots,T_6$. The same registers can be reused by Algorithm A.2 for calculations. Additionally, it needs 2 extra registers to store (X,Y) coordinates corresponding to $3P$, making a total of $6 + 2 = 8$ registers (see Lemma A.2). Algorithm A.3 also reuses temporary registers $T_1,\ldots,T_6$, and requires 3 registers per point, excepting the last one, to store (X, Y, A) values. For the last iteration, we only require registers $T_1,\ldots,T_6$ and 1 extra register to store A since the last (X,Y) coordinates are stored in T_1 and T_2 (see Lemma A.3). That makes an accumulated requirement of $6 + 2 + 3(L - 2) + 1 = 3L + 3$ at the end of Algorithm A.3, for $L \geq 2$. If $L = 1$, we do not compute Algorithm A.3, and the requirement is fixed by Algorithm A.2 at only 6 registers (note that in this case (X, Y) coordinates

are stored in T_1 and T_2). Algorithm A.4 requires 4 temporary registers for calculations (where T_1 and T_2 can store the (x, y) coordinates of the last point mP), $2(L-1)-2$ registers for (x, y) coordinates of the remaining $(L-1)$ points (assuming that T_3 and T_4 can store the (x, y) coordinates of $3P$) and $(L-1)$ registers for values A_j for $4 \le j \le (m+3)/2$, $m > 3$ odd, making a total requirement of $3L-1$. In conclusion, LM Scheme, case 2a, requires $3L+3$ registers. □

Theorem A.3. The LM Scheme, case 2b, has the following cost:

$$\text{Cost}_{\text{LM Scheme, case 2b}} = 1I + (9L)M + (2L+6)S,$$

and requires $(4L+1)$ registers.

Proof: Following Lemmas A.1-A.3, Algorithms A.1, A.2 and A.3 have the same costs as cases 1 and 2a, and Algorithm A.4 costs $1I + (3M + 1S) + (4M)(L-1)$. Adding these costs we obtain the value indicated for the LM Scheme, case 2b. Regarding memory requirements, Algorithm A.1 needs 6 registers $T_1,\ldots,T_6$, which can be reused by Algorithm A.2 for temporary calculations. Additionally, Algorithm A.2 needs 2 extra registers to store (X,Y) coordinates corresponding to $3P$, making a total of 6 + 2 = 8 registers (see Lemma A.2). Algorithm A.3 also reuses registers $T_1,\ldots,T_6$, and requires 4 registers per point, excepting the last one, to store (X, Y, B, C) values. For the last iteration, we only require registers $T_1,\ldots,T_6$ and 1 extra register to store C since the last (X,Y) coordinates are stored in T_1 and T_2, and T_6 stores B (see Lemma A.3). That makes an accumulated requirement of $6 + 2 + 4(L-2) + 1 = 4L + 1$ at the end of Algorithm A.3, for $L \ge 2$. If $L = 1$, we do not compute Algorithm A.3, and the requirement is fixed by Algorithm A.2 at only 6 registers as pointed out in the analysis for case 2a. Algorithm A.4 requires 4 registers for calculations (where T_1 and T_2 can store the (x, y) coordinates of the last point mP), $2(L-1)-2$ registers for (x, y) coordinates of the remaining $(L-1)$ points (assuming that T_3 and T_4 can store the (x, y) coordinates of $3P$) and $2(L-1)$ registers for values B_j, C_j for $4 \le j \le (m+3)/2$, $m > 3$ odd, making a total requirement of $4L-2$ registers. In conclusion, case 2b requires $4L+1$ registers. □

A3 Conjugate Addition Formulas

Conjugate (Mixed) Addition in Jacobian Coordinates

Let $P=(X_1:Y_1:Z_1)$ and $Q=(X_2:Y_2:Z_2)$ be two points in Jacobian coordinates on an elliptic curve E_W over $\mathbb{F}_p$. If the general addition $P+Q$ is performed using [LM08, formula (15)] and the partial values $(4\beta^3+8Z_2^2X_1\beta^2)$, $Z_2^2X_1\beta^2$, $-Z_2^3Y_1\beta^3$, Z_3, $Z_1^3Y_2$ and $Z_2^3Y_1$ are temporarily stored, the conjugate addition $P-Q=P+(-Q)=(X_1:Y_1:Z_1)+(X_2:-Y_2:Z_2)=(X_4:Y_4:Z_4)$ can be performed with the following:

$$X_4=\gamma^2-(4\beta^3+8Z_2^2X_1\beta^2),\ Y_4=\gamma(Z_2^2X_1\beta^2-X_4)-Z_2^3Y_1\beta^3,\ Z_4=Z_3, \tag{A.1}$$

where $\gamma=-2(Z_1^3Y_2+Z_2^3Y_1)$. This formula only requires $1M+1S+4A+1(\times 2)$.

In the case of mixed addition, let $P=(X_1:Y_1:Z_1)$ and $Q=(x_2,y_2)$ be two points on an elliptic curve E_W over $\mathbb{F}_p$. If the mixed addition $P+Q$ is performed using [LM08, formula (16)] and the partial values $(4\beta^3+8X_1\beta^2)$, $4X_1\beta^2$, $-8Y_1\beta^3$, Z_3 and $Z_1^3y_2$ are temporarily stored, the conjugate mixed addition $P-Q=P+(-Q)=(X_1:Y_1:Z_1)+(x_2:-y_2)=(X_4:Y_4:Z_4)$ can be performed as follows:

$$X_4=\gamma^2-(4\beta^3+8X_1\beta^2),\ Y_4=\gamma(4X_1\beta^2-X_4)-8Y_1\beta^3,\ Z_4=Z_3, \tag{A.2}$$

where $\gamma=-2(Z_1^3y_2+Y_1)$. This formula only costs $1M+1S+4A+1(\times 2)$.

To obtain the costs of the different addition/conjugate addition variants from Table 3.2, one needs to add the costs from Table 2.2 to costs of formulas (A.1) or (A.2). For instance, an addition/conjugate addition pair using $\text{ADD}_{[2,2]}$ has a cost of $(10M+2S)+(1M+1S)=11M+3S$, or $(9M+3S)+(1M+1S)=10M+4S$ if applying one *S*-*M* trading.

Conjugate (Mixed) Addition in $\mathcal{JQ}^e$ Coordinates

Let $P=(X_1:Y_1:Z_1:X_1^2:Z_1^2)$ and $Q=(X_2:Y_2:Z_2:X_2^2:Z_2^2)$ be two points in $\mathcal{JQ}^e$ coordinates on an extended Jacobi quartic curve $E_{JQ}/\mathbb{F}_p$ with $d=1$ in (2.11). If the addition $P+Q$ is performed using the following formula due to [HWC+07, HWC+08b]:

$$X_3=(\alpha+2Y_1)(\beta+2Y_2)-\alpha\beta-4Y_1Y_2,\ Z_3=4Z_1^2Z_2^2-4X_1^2X_2^2,\ X_3^2=(X_3)^2,\ Z_3^2=(Z_3)^2,$$
$$Y_3=(4X_1^2X_2^2+4Z_1^2Z_2^2+2\alpha\beta)[4(X_1^2+Z_1^2)(X_2^2+Z_2^2)+a\alpha\beta+4Y_1Y_2]-16(X_3^2+Z_3^2), \tag{A.3}$$

where $\alpha=(X_1+Z_1)^2-(X_1^2+Z_1^2)$, $\beta=(X_2+Z_2)^2-(X_2^2+Z_2^2)$, and the partial values β,

$(\alpha+2Y_1)$, $2Y_2$, $\alpha\beta$, $-4Y_1Y_2$, $(4X_1^2X_2^2+4Z_1^2Z_2^2)$, $2\alpha\beta$, $4(X_1^2+Z_1^2)(X_2^2+Z_2^2)+4Y_1Y_2$, $a\alpha\beta$, Z_3 and Z_3^2 are temporarily stored, then the conjugate addition $P-Q=P+(-Q)=(X_1,Y_1,Z_1,X_1^2,Z_1^2)+(-X_2,Y_2,Z_2,X_2^2,Z_2^2)=(X_4,Y_4,Z_4)$ can be performed with only $2M + 1S + 7A + 1\,(\times 16)$ as follows:

$$X_4=(\alpha+2Y_1)(-\beta+2Y_2)+\alpha\beta-4Y_1Y_2,\ Z_4=4Z_1^2Z_2^2-4X_1^2X_2^2=Z_3,\ X_4^2=(X_4)^2,\ Z_4^2=Z_3^2,$$
$$Y_4=(4X_1^2X_2^2+4Z_1^2Z_2^2-2\alpha\beta)[4(X_1^2+Z_1^2)(X_2^2+Z_2^2)-a\alpha\beta+4Y_1Y_2]-16(X_4^2+Z_4^2), \qquad (A.4)$$

Thus, the cost of an addition/conjugate addition pair is of $(7M + 4S) + (2M + 1S) = 9M + 5S$ if using an ADD operation or $(7M + 3S) + (2M + 1S) = 9M + 4S$, if using an $\text{ADD}_{[0,1]}$ operation. See Tables 2.4 and 3.2.

In the case of mixed addition, let $P=(X_1:Y_1:Z_1:X_1^2:Z_1^2)$ and $Q=(x_2,y_2,x_2^2)$ be two points in $\mathcal{JQ}^e$ and $\mathcal{A}$ coordinates, respectively, on an extended Jacobi quartic curve $E_{JQ}/\mathbb{F}_p$ with $d=1$ in (2.11). If the mixed addition $P+Q$ is performed using the following formula due to [HWC+07, HWC+08b]:

$$X_3=(\alpha+2Y_1)(x_2+y_2)-\alpha x_2-2Y_1y_2,\ Z_3=2(Z_1^2-X_1^2x_2^2),\ X_3^2=(X_3)^2,\ Z_3^2=(Z_3)^2,$$
$$Y_3=2((X_1^2x_2^2+Z_1^2+\alpha x_2)[2(X_1^2+Z_1^2)(x_2^2+1)+a\alpha x_2+2Y_1y_2]-2(X_3^2+Z_3^2)), \qquad (A.5)$$

where $\alpha=(X_1+Z_1)^2-(X_1^2+Z_1^2)$, and the partial values $(\alpha+2Y_1)$, αx_2, $-2Y_1y_2$, $(X_1^2x_2^2+Z_1^2)$, $[2(X_1^2+Z_1^2)(x_2^2+1)+2Y_1y_2]$, $a\alpha x_2$, Z_3 and Z_3^2 are temporarily stored, then the conjugate mixed addition $P-Q=P+(-Q)=(X_1:Y_1:Z_1:X_1^2:Z_1^2)+(-x_2,y_2,x_2^2)=(X_4:Y_4:Z_4:X_4^2:Z_4^2)$ can be performed with $2M + 1S + 7A + 2\,(\times 2)$ as follows:

$$X_4=(\alpha+2Y_1)(-x_2+y_2)+\alpha x_2-2Y_1y_2,\ Z_4=2(Z_1^2-X_1^2x_2^2)=Z_3,\ X_4^2=(X_4)^2,\ Z_4^2=Z_3^2,$$
$$Y_4=2((X_1^2x_2^2+Z_1^2-\alpha x_2)[2(X_1^2+Z_1^2)(x_2^2+1)-a\alpha x_2+2Y_1y_2]-2(X_4^2+Z_4^2)). \qquad (A.6)$$

Thus, the cost of a mixed addition/conjugate mixed addition pair is of $(6M + 3S) + (2M + 1S) = 8M + 4S$. See Tables 2.4 and 3.2.

Conjugate (Mixed) Addition in $\mathcal{IE}$ Coordinates

Let $P=(X_1:Y_1:Z_1)$ and $Q=(X_2:Y_2:Z_2)$ be two points in $\mathcal{IE}$ coordinates on a Twisted Edwards curve $E_{TE}/\mathbb{F}_p$ with $a=1$ in (2.12). If the general addition $P+Q$ is performed using the following formula due to [BL07b] (note that some terms have been rearranged to save a few

field additions):

$$X_3 = [X_1X_2Y_1Y_2 + d(Z_1Z_2)^2](X_1X_2 - Y_1Y_2),\ \ Y_3 = [X_1X_2Y_1Y_2 - d(Z_1Z_2)^2](X_1Y_2 + X_2Y_1),$$
$$Z_3 = Z_1Z_2(X_1X_2 - Y_1Y_2)(X_1Y_2 + X_2Y_1), \qquad \text{(A.7)}$$

and the partial values $[X_1X_2Y_1Y_2 + d(Z_1Z_2)^2]$, X_1X_2, Y_1Y_2, $[X_1X_2Y_1Y_2 - d(Z_1Z_2)^2]$, X_1Y_2, X_2Y_1 and Z_1Z_2 are temporarily stored, then the conjugate addition $P - Q = P + (-Q) = (X_1 : Y_1 : Z_1) + (-X_2 : Y_2 : Z_2) = (X_4 : Y_4 : Z_4)$ can be performed with the following (with a cost of only $4M + 2A$):

$$X_4 = [X_1X_2Y_1Y_2 - d(Z_1Z_2)^2](X_1X_2 + Y_1Y_2),\ \ Y_4 = -[X_1X_2Y_1Y_2 + d(Z_1Z_2)^2](X_1Y_2 - X_2Y_1),$$
$$Z_4 = -Z_1Z_2(X_1X_2 + Y_1Y_2)(X_1Y_2 - X_2Y_1), \qquad \text{(A.8)}$$

Thus, the cost of an addition/conjugate addition pair is of $(10M + 1S) + 4M = 14M + 1S$.

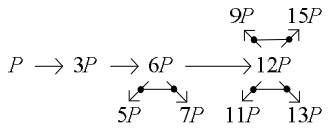
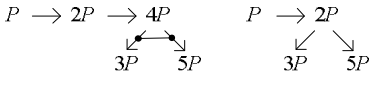
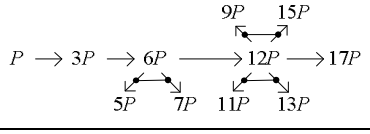
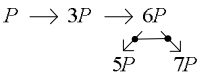
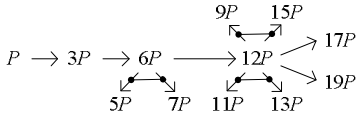
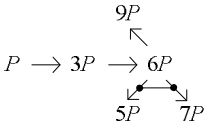
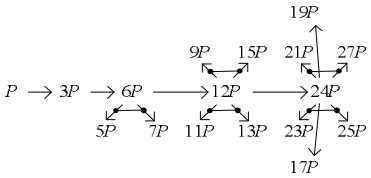
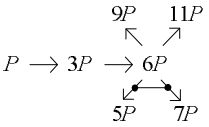
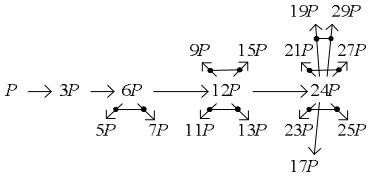
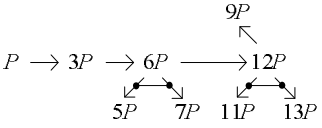
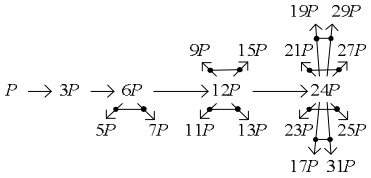
The formula for mixed addition can be obtained by setting $Z_2 = 1$ in formula (A.7) and has a cost of $9M + 1S + 4A$. Then, if the partial values $(X_1x_2Y_1y_2 + dZ_1^2)$, $(X_1x_2Y_1y_2 - dZ_1^2)$, X_1x_2, Y_1y_2, X_1y_2 and x_2Y_1 are temporarily cached, the conjugate mixed addition $P - Q = P + (-Q) = (X_1 : Y_1 : Z_1) + (x_2 : -y_2) = (X_4 : Y_4 : Z_4)$ can be performed by:

$$X_4 = [X_1x_2Y_1y_2 - dZ_1^2](X_1x_2 + Y_1y_2),\ \ Y_4 = -[X_1x_2Y_1y_2 + dZ_1^2](X_1y_2 - x_2Y_1),$$
$$Z_4 = -Z_1(X_1x_2 + Y_1y_2)(X_1y_2 - x_2Y_1), \qquad \text{(A.9)}$$

which only costs $4M + 2A$. Therefore, the cost of a mixed addition/conjugate mixed addition pair is of $(9M + 1S) + 4M = 13M + 1S$.

A4 Calculation of Precomputed Points for the LG Scheme

The following table shows the proposed sequences for computing a table with the form $d_i P$, where $d_i \in D^+ \setminus \{0,1\} = \{3,5,...,m\}$ with m odd. For m = 5, the first sequence corresponds to $\mathcal{J}$ and $\mathcal{JQ}^e$, and the second one to $\mathcal{IE}$ coordinates. Tied arrows denote an addition/conjugate addition pair (or mixed addition/conjugate mixed addition pair if addition is performed with affine point P).

m	Precomputation Scheme	m	Precomputation Scheme
3		15	
5		17	
7		19	
9		27	
11		29	
13		31	

A5 Cost Analysis of the LG Scheme, Table d_iP

Theorem A.4. Given an elliptic curve E of arbitrary form, the cost of using the LG Scheme for computing a precomputed table with the form d_iP, where $d_i \in D^+ \setminus \{0,1\} = \{3,5,...,m\}$ with m odd and the base point $P \in E(\mathbb{F}_p)$, is given by:

$$\text{Cost}_{\text{case 1/3 (2)}} = 1\text{TPL} + (\omega - 2)\text{DBL} + (2\varepsilon - L + 1)\text{ADD} + (L - \varepsilon - 1)\text{ADD-ADD}' \; (+\text{Cost}_{\mathcal{P}\to\mathcal{A}}),$$

where $m \geq 3$, $L = (m-1)/2$, $r_{\max} = 3 \times 2^{\omega-2}$ is the value of the highest "strategic" point, $\varepsilon = \left\lfloor (6L + 2r_{\max} - 3)/(6r_{\max} - 3) \right\rfloor \left(L + 1 - 2r_{\max}/3\right) + (r_{\max}/3) - 1$ and $\text{Cost}_{\mathcal{P}\to\mathcal{A}}$ denotes the cost of converting points from projective to affine coordinates in case 2.

Proof: first, note that $m \geq 3$. If $r_{\max}$ is defined as the value of the highest "strategic" point, then it holds that $r_{\max} = 3 \times 2^{\omega-2}$ for some integer $\omega \geq 2$ since "strategic" points have the form $P_{i+1} = 2P_i$, for integers $i \geq 0$ with $P_0 = 3P$. It easily follows that calculating all "strategic" points up to $r_{\max}P = (3 \times 2^{\omega-2})P$ requires one tripling and $(\omega - 2)$ doublings. Then, additions are required to compute each point in the table except $3P$, which is already calculated. Since there are L non-trivial points in the table, we require $(L-1)$ additions in total. Let us now estimate the number of regular additions required for computing points below $r_{\max}P$, and then above $r_{\max}P$. First, up to $r_{\max}P$ there are $r_{\max}/2$ odd points, from which $(r_{\max}/6) - 1$ are computed with a conjugate addition. If P and $3P$ are discarded we require $(r_{\max}/2) - (r_{\max}/6) + 1 - 2 = (r_{\max}/3) - 1$ regular additions up to $r_{\max}P$. Above $r_{\max}P$ there is a range for which points are computed with conjugate additions. Then we need to establish the value $r_{\max} < k < m$ s.t. points kP, $(k+2)P$, … , mP are calculated with regular additions. Following Appendix A4, it is straightforward to note that $k = 9$ if $r_{\max} = 6$, $k = 17$ if $r_{\max} = 12$, $k = 33$ if $r_{\max} = 24$, and so on. Thus, $k = (4r_{\max} + 3)/3$ and, hence, $\frac{m - (4r_{\max}+3)/3}{2} + 1 = L + 1 - 2r_{\max}/3$ regular additions are required above $r_{\max}P$. However, an exception happens when $m < k$, for which case the number of additions above $r_{\max}P$ should be zero. The latter can be accomplished by simply multiplying $\left\lfloor (2r_{\max} - 1 + m - (4r_{\max} + 3)/3)/(2r_{\max} - 1) \right\rfloor = \left\lfloor (6L + 2r_{\max} - 3)/(6r_{\max} - 3) \right\rfloor$ with $L + 1 - 2r_{\max}/3$. Therefore, the total number of regular additions is given by the expression $\varepsilon = \left\lfloor (6L + 2r_{\max} - 3)/(6r_{\max} - 3) \right\rfloor \left(L + 1 - 2r_{\max}/3\right) + (r_{\max}/3) - 1$. Since it was established that there are $(L-1)$ additions in total, then $(L - 1 - \varepsilon)$ are addition/conjugate addition pairs and $\varepsilon - (L - 1 - \varepsilon) = 2\varepsilon - L + 1$ are individual additions. By definition, case 2 requires the addition of the cost of converting projective points to affine. □

Corollary A.1. In the setting of Theorem A.4, the cost of the LG Scheme when using mixed coordinates is given by:

$$\mathrm{Cost}_{\mathrm{case\,1(2)}} = 1\mathrm{mTPL} + (\omega-2)\mathrm{DBL} + (\omega-2)\mathrm{mADD\text{-}mADD'} + \left(L-\varepsilon-\omega+1\right)\mathrm{ADD\text{-}ADD'} + \ldots$$

$$\ldots(2\varepsilon-L+1)\mathrm{ADD}\ \ (+\mathrm{Cost}_{\mathcal{P}\to\mathcal{A}})\,.$$

Proof: assuming that the base point P is given in affine coordinates, then $P_0 = 3P$ can be computed using a mixed tripling with the form $3\mathcal{A}\to\mathcal{P}$. Since $(\omega-2)$ doublings are required, there are also $(\omega-2)$ "strategic" points. By definition, $m > r_{\max}$, so for each "strategic" point P_j there is always a pair of points with the form $P_j \pm P$. Then, there are $(\omega-2)$ points that can be calculated with an addition/conjugate addition pair using mixed *Projective*-affine coordinates, that is, computing $\mathcal{P}\pm\mathcal{A}\to\mathcal{P}$. According to Theorem A.4, there are $(L-\varepsilon-1)$ addition/ conjugate addition pairs in total. Hence, $(L-\varepsilon-1)-(\omega+2) = L-\varepsilon-\omega+1$ are addition/ conjugate addition pairs using Jacobian coordinates, that is, computing $\mathcal{P}\pm\mathcal{P}\to\mathcal{P}$. □

A6 Cost Analysis of the LG Scheme, Table $c_iP \pm d_iQ$

Theorem A.5. Given an elliptic curve E of arbitrary form, the cost of using the LG Scheme for computing a precomputed table with the form $c_iP \pm d_iQ$, where $c_i, d_i \in D^+ = \{0,1,3,5,...,m\}$, $c_i > 1$ if $d_i = 0$, $d_i > 1$ if $c_i = 0$, m odd and P, Q are points in $E(\mathbb{F}_p)$, is given by:

$$\text{Cost}_{\text{cases 1/3(2)}} = (m-1)\text{ADD} + \frac{(m+1)^2}{4}(\text{ADD}-\text{ADD}') + 2\left\lceil \frac{m-1}{m} \right\rceil \text{DBL} \ (+\text{Cost}_{\mathcal{P}\to\mathcal{A}}),$$

where $L = (m^2 + 4m - 1)/2 > 1$ is the number of non-trivial points in the table and $\text{Cost}_{\mathcal{P}\to\mathcal{A}}$ denotes the cost of converting points from projective to affine coordinates in case 2.

Proof: first, let us establish the value L. There are $(m+1)$ points with the form c_iP or d_iQ, which can be combined in $(m+1)^2/2$ ways to get points of the form $c_iP \pm d_iQ$ with $c_id_i \neq 0$. By discarding points P and Q, we obtain the total number of non-trivial points as $L = \frac{(m+1)^2}{2} + (m+1) - 2 = \frac{m(m+4)-1}{2}$. As it always holds that $m \geq 1$, then $L > 1$. The points c_iP or d_iQ with $c_i \geq 3$ and $d_i \geq 3$ can be computed with two sequences with the form $P \to P+2P = 3P \to 3P+2P = 5P \to \ldots \to (m-2)P+2P = mP$. This requires in total *two* doublings and $(m-1)$ additions. Note that when $m = 1$, there are no calculations required for this part. Hence, for $m \geq 1$ the number of required doublings can be expressed by $2\lceil (m-1)/m \rceil$. Finally, the computation of the $(m+1)^2/2$ points $c_iP \pm d_iQ$ with $c_id_i \neq 0$ involves $(m+1)^2/4$ addition/conjugate addition pairs. By definition, case 2 requires in addition the cost of converting points from projective to affine coordinates. □

Theorem A.6. In the setting of Theorem A.5 and assuming that $m \geq 5$, the cost of the LG Scheme when using Jacobian coordinates is given by:

$$\text{Cost}_{\text{cases 1(2)}} = 2\text{DBL} + (m-1)\text{ADD}_{\text{Co-}Z} + \frac{(m+1)^2}{4}(\text{ADD}-\text{ADD}') \ (+\text{Cost2}_{\mathcal{J}\to\mathcal{A}}),$$

where $\text{Cost2}_{\mathcal{J}\to\mathcal{A}} = [2m(m+4)-1]M + [(m+1)^2/4 + 2]S$ for case 2.

Proof: according to Theorem A.1, if $m \geq 3$ points with the form d_iP, where $d_i \in D^+ \setminus \{0,1\} = \{3,5,\ldots,m\}$ can be computed with the sequence $P \to P+2P = 3P \to 3P+2P = 5P \to \ldots \to (m-2)P+2P = mP$ using one (mixed) doubling and $(m-1)/2$ additions with identical Z coordinate. Then, points c_iP and d_iQ with $c_i \geq 3$ and $d_i \geq 3$ can be computed with two doublings and $(m-1)$ additions with identical Z coordinate. The restriction $m \geq 5$ is because when $m = 3$ it is more efficient to compute $3P$ directly with a (mixed) tripling operation. Following Theorem A.5, the computation of the $(m+1)^2/2$ points $c_iP \pm d_iQ$ with $c_id_i \neq 0$

involves $(m+1)^2/4$ addition/conjugate addition pairs. Let us now proof $\text{Cost2}_{\mathcal{J}\to\mathcal{A}}$. Following the LM Scheme, case 2b, sequences for c_iP and d_iQ using additions with identical Z coordinate yield the *two* Z-coordinates Z_{mP} and Z_{mQ}. Since conjugate additions share the same Z coordinate, the $(m+1)^2/4$ addition/conjugate addition pairs $c_iP \pm d_iQ$ with $c_i d_i \neq 0$ yield $t=(m+1)^2/4$ Z-coordinates. In total, there are $(t+2)$ distinct Z-coordinates. Applying Montgomery's method for simultaneous inversion, the latter first requires one inversion and $3(t+1)$ multiplications to invert all Z coordinates combined and then recover each of them. Second, recovering $(X:Y:Z)$ coordinates of the $2t=(m+1)^2/2$ points $c_iP \pm d_iQ$ involves $(3M+1S)t$ and $(2M)t$ for points obtained by addition and conjugate addition, respectively; and recovering $(X:Y:Z)$ coordinates of the points c_iP and d_iQ by applying LM Scheme, case 2b, to coordinates Z_{mP} and Z_{mQ} costs $2\left[(2m-3)M+1S\right]$. In total, the cost of conversion to affine is $1I+3(t+1)M+(3M+1S)t+(2M)t+2(2m-3)M+2S = 1I+(8t+4m-3)M+(t+2)S$. □

A7 Comparison of LG and LM Schemes using Jacobian Coordinates

The tables below compare the performance of LM and LG Schemes with the DOS method for n = 256 and 512. For each method, we show the cost of performing an n-bit scalar multiplication and the optimal number of precomputed points L when considering that a maximum of $(2L_{ES} + R)$ registers are available for the evaluation stage (i.e., $L \leq L_{ES}$). For our analysis, R = 7. Also, to compare the performance of schemes for cases 1 and 2, we include costs of the most efficient scheme for case 1 (i.e., LM Scheme, case 1) and show at the bottom of each table the *I/M* range for which LM Scheme, case 1, would achieve the lowest cost.

Table A.1. Performance comparison of LG and LM Schemes with the DOS method in 256-bit scalar multiplication for different memory constraints on a standard curve ($1M = 0.8S$).

# Registers (L_{ES})	11 (2)		13 (3)		15 (4)		17 (5)		19 (6)	
Method	L	Cost	L	Cost	L	Cost	L	Cost	L	Cost
LM, case 2b	2	**$1I + 2396M$**	3	**$1I + 2349M$**	3	$1I + 2349M$	4	$1I + 2335M$	4	$1I + 2335M$
LM, case 2a	2	$1I + 2397M$	3	$1I + 2350M$	4	**$1I + 2337M$**	4	$1I + 2337M$	5	**$1I + 2326M$**
LG, case 2	2	$1I + 2401M$	3	$1I + 2354M$	4	$1I + 2347M$	5	$1I + 2345M$	6	$1I + 2331M$
DOS [DOS07]	2	$1I + 2399M$	3	$1I + 2354M$	4	$1I + 2342M$	5	**$1I + 2333M$**	6	**$1I + 2326M$**
LM, case 1	1	$2548M$	1	$2548M$	1	$2548M$	2	$2505M$	2	$2505M$
I/M range (LM, case1)	$I > 152M$		$I > 199M$		$I > 211M$		$I > 172M$		$I > 179M$	

# Registers (L_{ES})	23 (8)		27 (10)		29 (11)		35 (14)		≥ 61 (27)	
Method	L	Cost	L	Cost	L	Cost	L	Cost	L	Cost
LM, case 2b	5	$1I + 2323M$	6	$1I + 2313M$	6	**$1I + 2305M$**	7	**$1I + 2305M$**	7	**$1I + 2305M$**
LM, case 2a	6	$1I + 2317M$	7	**$1I + 2309M$**	7	$1I + 2309M$	7	$1I + 2309M$	7	$1I + 2309M$
LG, case 2	7	**$1I + 2316M$**	7	$1I + 2316M$	7	$1I + 2316M$	7	$1I + 2316M$	7	$1I + 2316M$
DOS [DOS07]	7	$1I + 2320M$	7	$1I + 2320M$	7	$1I + 2320M$	7	$1I + 2320M$	7	$1I + 2320M$
LM, case 1	3	$2457M$	4	$2443M$	4	$2443M$	5	$2414M$	6	$2397M$
I/M range (LM, case1)	$I > 141M$		$I > 134M$		$I > 138M$		$I > 109M$		$I > 92M$	

Table A.2. Performance comparison of LG and LM Schemes with the DOS method in 512-bit scalar multiplication for different memory constraints on a standard curve ($1M = 0.8S$).

# Registers (L_{ES})	11 (2)		13 (3)		15 (4)		17 (5)		19 (6)	
Method	L	Cost	L	Cost	L	Cost	L	Cost	L	Cost
LM, case 2b	2	**$1I + 4768M$**	3	**$1I + 4663M$**	3	$1I + 4663M$	4	$1I + 4624M$	4	$1I + 4624M$
LM, case 2a	2	$1I + 4769M$	3	$1I + 4665M$	4	**$1I + 4626M$**	4	$1I + 4626M$	5	$1I + 4593M$
LG, case 2	2	$1I + 4773M$	3	$1I + 4668M$	4	$1I + 4636M$	5	$1I + 4611M$	6	$1I + 4577M$
DOS [DOS07]	2	$1I + 4771M$	3	$1I + 4668M$	4	$1I + 4632M$	5	**$1I + 4600M$**	6	**$1I + 4572M$**
LM, case 1	1	$5089M$	1	$5089M$	1	$5089M$	2	$4991M$	2	$4991M$
I/M range (LM, case1)	$I > 321M$		$I > 426M$		$I > 463M$		$I > 391M$		$I > 419M$	

# Registers (L_{ES})	21 (7)		23 (8)		25 (9)		27 (10)		29 (11)	
Method	L	Cost	L	Cost	L	Cost	L	Cost	L	Cost
LM, case 2b	5	$1I + 4589M$	5	$1I + 4589M$	6	$1I + 4559M$	6	$1I + 4559M$	7	$1I + 4532M$
LM, case 2a	6	$1I + 4563M$	6	$1I + 4563M$	7	**$1I + 4537M$**	8	**$1I + 4530M$**	8	**$1I + 4530M$**
LG, case 2	7	**$1I + 4543M$**	7	**$1I + 4543M$**	7	$1I + 4543M$	7	$1I + 4543M$	7	$1I + 4543M$
DOS [DOS07]	7	$1I + 4547M$	8	**$1I + 4543M$**	9	$1I + 4539M$	10	$1I + 4536M$	11	$1I + 4533M$
LM, case 1	3	$4887M$	3	$4887M$	3	$4887M$	4	$4849M$	4	$4849M$
I/M range (LM, case1)	$I > 344M$		$I > 344M$		$I > 350M$		$I > 319M$		$I > 319M$	

# Registers (L_{ES})	31 (12)		33 (13)		35 (14)		37 (15)		39 (16)	
Method	L	Cost	L	Cost	L	Cost	L	Cost	L	Cost
LM, case 2b	7	$1I + 4532M$	8	$1I + 4525M$	8	$1I + 4525M$	9	$1I + 4518M$	9	$1I + 4518M$
LM, case 2a	9	**$1I + 4525M$**	10	**$1I + 4520M$**	10	**$1I + 4520M$**	11	**$1I + 4515M$**	12	**$1I + 4512M$**
LG, case 2	7	$1I + 4543M$	13	$1I + 4536M$	14	$1I + 4525M$	15	$1I + 4516M$	15	$1I + 4516M$
DOS [DOS07]	12	$1I + 4531M$	13	$1I + 4530M$	14	$1I + 4529M$	14	$1I + 4529M$	14	$1I + 4529M$
LM, case 1	5	$4811M$	5	$4811M$	5	$4811M$	6	$4774M$	6	$4774M$
I/M range (LM, case1)	$I > 286M$		$I > 291M$		$I > 291M$		$I > 259M$		$I > 262M$	

# Registers (L_{ES})	41 (17)		43 (18)		47 (20)		51 (22)		55 (24)	
Method	L	Cost	L	Cost	L	Cost	L	Cost	L	Cost
LM, case 2b	10	**$1I + 4512M$**	10	$1I + 4512M$	11	$1I + 4507M$	12	**$1I + 4503M$**	13	**$1I + 4499M$**
LM, case 2a	12	**$1I + 4512M$**	13	**$1I + 4508M$**	14	**$1I + 4506M$**	15	$1I + 4504M$	15	$1I + 4504M$
LG, case 2	15	$1I + 4516M$	15	$1I + 4516M$	15	$1I + 4516M$	15	$1I + 4516M$	15	$1I + 4516M$
DOS [DOS07]	14	$1I + 4529M$	14	$1I + 4529M$	14	$1I + 4529M$	14	$1I + 4529M$	14	$1I + 4529M$
LM, case 1	7	$4740M$	7	$4740M$	7	$4740M$	8	$4730M$	9	$4719M$
I/M range (LM, case1)	$I > 228M$		$I > 232M$		$I > 234M$		$I > 227M$		$I > 220M$	

# Registers (L_{ES})	59 (26)		61 (27)		≥ 81 (37)	
Method	L	Cost	L	Cost	L	Cost
LM, case 2b	14	**$1I + 4495M$**	15	**$1I + 4492M$**	15	**$1I + 4492M$**
LM, case 2a	15	$1I + 4504M$	15	$1I + 4504M$	15	$1I + 4504M$
LG, case 2	15	$1I + 4516M$	15	$1I + 4516M$	15	$1I + 4516M$
DOS [DOS07]	14	$1I + 4529M$	14	$1I + 4529M$	14	$1I + 4529M$
LM, case 1	10	$4709M$	11	$4699M$	15	$4665M$
I/M range (LM, case1)	$I > 214M$		$I > 207M$		$I > 173M$	

Appendix B

The following Maple scripts detail the improved explicit formulas for the case of Jacobian ($\mathcal{J}$) and mixed Twisted Edwards homogeneous/extended homogeneous ($\mathcal{E}/\mathcal{E}^e$) coordinates exploiting the techniques discussed in Chapter 5, namely incomplete reduction, merging and scheduling of field operations and merging of point operations.

B1 Explicit Formulas for "Traditional" Implementations

These formulas have been used for the "traditional" implementations discussed in Section 5.6.1. Temporary registers are denoted by t_i and Mul = multiplication, Sqr = squaring, Add = addition, Sub = subtraction, Mul$x$ = multiplication by x, Div$x$ = division by x, Neg = negation. DblSub represents the computation $a-2b \pmod p$ and SubDblSub represents the merging of $a-b \pmod p$ and $(a-b)-2c \pmod p$. Underlined field operations are merged and $operation_{IR}$ represents a field operation using incomplete reduction. In practice, input registers are reused to store the result of an operation.

Explicit Formulas using Jacobian Coordinates

```
# Weierstrass curve (for verification):
x1:=X1/Z1^2; y1:=Y1/Z1^3; x2:=X2/Z2^2; y2:=Y2/Z2^3; ZZ2:=Z2^2; ZZZ2:=Z2^3; a:=-3;
x3:=((3*x1^2+a)/(2*y1))^2-2*x1; y3:=((3*x1^2+a)/(2*y1))*(x1-x3)-y1;
x4:=((y1-y2)/(x1-x2))^2-x2-x1; y4:=((y1-y2)/(x1-x2))*(x2-x4)-y2;
x5:=((y1-y4)/(x1-x4))^2-x4-x1; y5:=((y1-y4)/(x1-x4))*(x4-x5)-y4;
```

DBL, $2\mathcal{J} \to \mathcal{J}$: $2(X_1 : Y_1 : Z_1) \to (X_{out} : Y_{out} : Z_{out})$. Cost = 4Mul+4Sqr+3Sub+1DblSub+ 1Add_{IR}+1Mul3_{IR}+1Div2_{IR}; 5 contiguous data dependencies

```
# In practice, Xout,Yout,Zout reuse the registers X1,Y1,Z1 for all cases below.
t4:=Z1^2; t3:=Y1^2; t1:=X1+t4; t4:=X1-t4; t0:=3*t4; t5:=X1*t3; t4:=t1*t0; t0:=t3^2;
t1:=t4/2; t3:=t1^2; Zout:=Y1*Z1; Xout:=t3-2*t5; t3:=t5-Xout; t5:=t1*t3; Yout:=t5-t0;
simplify([x3-Xout/Zout^2]), simplify([y3-Yout/Zout^3]); # Check
```

4DBL, $8\mathcal{J} \to \mathcal{J}$: $8(X_1 : Y_1 : Z_1) \to (X_{out} : Y_{out} : Z_{out})$. Cost = 4*(4Mul+4Sqr+3Sub+1DblSub+ 1Add_{IR}+1Mul3_{IR}+1Div2_{IR}); 1.25 contiguous data dependencies/doubling

```
t4:=Z1^2; t3:=Y1^2; t1:=X1+t4; t4:=X1-t4; t2:=3*t4; t5:=X1*t3; t4:=t1*t2; t0:=t3^2;
t1:=t4/2; Zout:=Y1*Z1; t3:=t1^2; t4:=Z1^2; Xout:=t3-2*t5; t3:=t5-Xout; t2:=Xout+t4;
t5:=t1*t3; t4:=Xout-t4; Yout:=t5-t0; t1:=3*t4; t3:=Yout^2; t4:=t1*t2; t5:=Xout*t3;
t1:=t4/2; t0:=t3^2; t3:=t1^2; Zout:=Yout*Zout; Xout:=t3-2*t5; t4:=Zout^2; t3:=t5-Xout;
t2:=Xout+t4; t5:=t1*t3; t4:=Xout-t4; Yout:=t5-t0; t1:=3*t4; t3:=Yout^2; t4:=t1*t2;
t5:=Xout*t3; t1:=t4/2; t0:=t3^2; t3:=t1^2; Zout:=Yout*Zout; Xout:=t3-2*t5; t4:=Zout^2;
t3:=t5-Xout; t2:=Xout+t4; t5:=t1*t3; t4:=Xout-t4; Yout:=t5-t0; t1:=3*t4; t3:=Yout^2;
t4:=t1*t2; t5:=Xout*t3; t1:=t4/2; t0:=t3^2; t3:=t1^2; Zout:=Yout*Zout; Xout:=t3-2*t5;
t3:=t5-Xout; t5:=t1*t3; Yout:=t5-t0;
```

mDBLADD, $2\mathcal{J} + \mathcal{A} \to \mathcal{J}$: $2(X_1 : Y_1 : Z_1) + (x_2, y_2) \to (X_{out} : Y_{out} : Z_{out})$. Cost = 13Mul+5Sqr+ 7Sub+2DblSub+1Add_{IR}+1Mul2_{IR}; 5 contiguous data dependencies

```
t5:=Z1^2; t6:=Z1*t5; t4:=x2*t5; t5:=y2*t6; t1:=t4-X1; t2:=t5-Y1; t4:=t2^2; t6:=t1^2;
t5:=t6*X1; t0:=t1*t6; t3:=t4-2*t5; t4:=Z1*t1; t3:=t3-t5; t6:=t0*Y1; t3:=t3-t0; t1:=2*t6;
Zout:=t4*t3; t4:=t2*t3; t0:=t3^2; t1:=t1+t4; t4:=t0*t5; t7:=t1^2; t5:=t0*t3; Xout:=t7-
2*t4; Xout:=Xout-t5; t3:=Xout-t4; t0:=t5*t6; t4:=t1*t3; Yout:=t4-t0;
simplify([x5-Xout/Zout^2]), simplify([y5-Yout/Zout^3]); # Check
```

DBLADD, $2\mathcal{J} + \mathcal{J} \to \mathcal{J}$: $2(X_1 : Y_1 : Z_1) + (X_2 : Y_2 : Z_2 : Z_2^2 : Z_2^3) \to (X_{out} : Y_{out} : Z_{out})$. Cost = 16Mul+ 5Sqr+7Sub+2DblSub+1Add_{IR}+1Mul2_{IR}; 3 contiguous data dependencies

```
t0:=X1*ZZ2; t5:=Z1^2; t7:=Y1*ZZZ2; t4:=X2*t5; t6:=t5*Z1; t1:=t4-t0; t5:=Y2*t6; t6:=t1^2;
t2:=t5-t7; t4:=t2^2; t5:=t6*t0; t0:=t1*t6; t3:=t4-2*t5; t6:=Z1*t1; t3:=t3-t5; t4:=Z2*t6;
t3:=t3-t0; t6:=t7*t0; Zout:=t4*t3; t4:=t2*t3; t1:=2*t6; t0:=t3^2; t1:=t1+t4; t4:=t0*t5;
t7:=t1^2; t5:=t0*t3; Xout:=t7-2*t4; Xout:=Xout-t5; t3:=Xout-t4; t0:=t5*t6; t4:=t1*t3;
Yout:=t4-t0;
simplify([x5-Xout/Zout^2]), simplify([y5-Yout/Zout^3]); # Check
```

Explicit Formulas using $\mathcal{E}/\mathcal{E}^e$ Coordinates

```
# Twisted Edwards curve (for verification):
x1:=X1/Z1; y1:=Y1/Z1; x2:=X2/Z2; y2:=Y2/Z2; T2:=X2*Y2/Z2; a:=-1;
x3:=(2*x1*y1)/(y1^2+a*x1^2); y3:=(y1^2-a*x1^2)/(2-y1^2-a*x1^2);
x4:=(x3*y3+x2*y2)/(y3*y2+a*x3*x2); y4:=(x3*y3-x2*y2)/(x3*y2-y3*x2);
```

DBL, $2\mathcal{E} \to \mathcal{E} : 2(X_1 : Y_1 : Z_1) \to (X_{out} : Y_{out} : Z_{out})$. Cost = 4Mul+3Sqr+1SubDblSub+1Add$_{IR}$+1Mul2$_{IR}$+1Neg; no contiguous data dependencies

```
t1:=2*X1; t2:=X1^2; t4:=Y1^2; t3:=Z1^2; Xout:=t2+t4; t4:=t4-t2; t3:=t4-2*t3; t2:=t1*Y1;
Yout:=-t4; Zout:=t4*t3; Yout:=Yout*Xout; Xout:=t3*t2;
simplify([x3-Xout/Zout]), simplify([y3-Yout/Zout]); # Check
# Iterate this code n times to implement nDBL with cost n(4M+3S+1SubDblSub+1AddIR+
1Mul2IR+1Neg)
```

Merged DBL–ADD, $(2\mathcal{E})^e + \mathcal{E}^e \to \mathcal{E} : 2(X_1 : Y_1 : Z_1) + ((X_2 + Y_2) : (X_2 - Y_2) : 2Z_2 : 2T_2) \to (X_{out} : Y_{out} : Z_{out})$. Cost = 12Mul+3Sqr+3Sub+1SubDblSub+4Add$_{IR}$+1Mul2$_{IR}$; no contiguous data dependencies

```
# If Z2=1 (Merged DBL-mADD), t5:=(2*Z2)*t6 is replaced by t5:=2*t6 and the number of
multiplies reduces to 11M at the expense of one extra Mul2
t1:=2*X1; t5:=X1^2; t7:=Y1^2; t6:=Z1^2; Xout:=t5+t7; t7:=t7-t5; t6:=t7-2*t6; t5:=t1*Y1;
t8:=t7*Xout; t0:=t7*t6; t7:=t6*t5; t6:=Xout*t5; Xout:=t7+t8; t1:=t7-t8; t7:=(2*T2)*t0;
t5:=(2*Z2)*t6; t0:=(X2-Y2)*t1; t1:=t5+t7; t6:=(X2+Y2)*Xout; Xout:=t5-t7; t7:=t0-t6;
t0:=t0+t6; Xout:=Xout*t7; Yout:=t1*t0; Zout:=t0*t7;
simplify([x4-Xout/Zout]), simplify([y4-Yout/Zout]); # Check
```

B2 Explicit Formulas for GLS-Based Implementations

These formulas have been used for the GLS-based implementations discussed in Section 5.6.2. Temporary registers are denoted by t_i and Mul = multiplication, Sqr = squaring, Add = addition, Sub = subtraction, Mul$x$ = multiplication by x, Div$x$ = division by x, Neg = negation. DblSub represents the operation $a-2b \pmod p$ or $a-b-c \pmod p$, Mul3Div2 represents the operation $(a+a+a)/2 \pmod p$, AddSub represents the merging of $a+b \pmod p$ and $a-b \pmod p$, AddSub2 represents $a+b-c \pmod p$, SubSub represents the merging of $a-b \pmod p$ and $c-d \pmod p$, and Mul2Mul3 represents the merging of $a+a \pmod p$ and $a+a+a \pmod p$. Underlined field operations are merged and $operation_{\mathrm{IR}}$ represents a field operation using incomplete reduction. In practice, input registers are reused to store the result of an operation.

Explicit Formulas using Jacobian Coordinates

```
# Weierstrass curve (for verification):
x1:=X1/Z1^2; y1:=Y1/Z1^3; a:=-3;
x3:=((3*x1^2+u^2*a)/(2*y1))^2-2*x1; y3:=((3*x1^2+u^2*a)/(2*y1))*(x1-x3)-y1;
x4:=((y1-y2)/(x1-x2))^2-x2-x1; y4:=((y1-y2)/(x1-x2))*(x2-x4)-y2;
x5:=((y1-y4)/(x1-x4))^2-x4-x1; y5:=((y1-y4)/(x1-x4))*(x4-x5)-y4;
```

DBL, $2\mathcal{J} \rightarrow \mathcal{J}$: $2(X_1 : Y_1 : Z_1) \rightarrow (X_{out} : Y_{out} : Z_{out})$. Cost = 4Mul+4Sqr+2Sub+1DblSub+1Mul3Div2+1AddSub+1Mulμ; no contiguous data dependencies

```
# In practice, Xout,Yout,Zout reuse the registers X1,Y1,Z1 for all cases below.
t2:=Z1^2; t3:=Y1^2; t1:=u*t2; t2:=X1+t1; t1:=X1-t1; t1:=3*t1/2; t4:=t3*X1; t1:=t2*t1;
t3:=t3^2; Xout:=t1^2; Zout:=Y1*Z1; Xout:=Xout-2*t4; t2:=t4-Xout; t1:=t1*t2; Yout:=t1-t3;
simplify([x3-Xout/Zout^2]), simplify([y3-Yout/Zout^3]); # Check
```

mADD, $\mathcal{J}+\mathcal{A} \rightarrow \mathcal{J}$: $(X_1 : Y_1 : Z_1)+(x_2, y_2) \rightarrow (X_{out} : Y_{out} : Z_{out})$. Cost = 8Mul+3Sqr+5Sub+1DblSub; no contiguous data dependencies

```
t2:=Z1^2; t1:=Z1*t2; t2:=t2*x2; t1:=t1*y2; t2:=t2-X1; t1:=t1-Y1; t3:=t2^2; t4:=t1^2;
Zout:=Z1*t2; t2:=t2*t3; t3:=t3*X1; Xout:=t4-t2; Xout:=Xout-2*t3; t3:=t3-Xout; t1:=t1*t3;
Yout:=t2*Y1; Yout:=t1-Yout;
simplify([x4-Xout/Zout^2]), simplify([y4-Yout/Zout^3]); # Check
```

mDBLADD, $2\mathcal{J}+\mathcal{A} \rightarrow \mathcal{J}$: $2(X_1 : Y_1 : Z_1)+(x_2, y_2) \rightarrow (X_{out} : Y_{out} : Z_{out})$. Cost = 13Mul+5Sqr+2Sub+2DblSub+1SubSub+1Add+1Mul2+1Mul2Mul3+1Div2; no contiguous data depend.

```
t2:=Z1^2; t1:=Z1*t2; t3:=x2*t2; t1:=y2*t1; t2:=t3-X1; t1:=t1-Y1; t3:=t2^2; t5:=t1^2;
t4:=X1*t3; t3:=t2*t3; Xout:=2*t4; t4:=3*t4; Zout:=Z1*t2; t5:=t5-t3-t4; Yout:=t3*Y1; t1:=
t1*t5; t2:=2*Yout; t3:=t5^2; t1:=t1+t2; t2:=Xout*t3; Xout:=t1^2; t3:=t5*t3; Xout:=Xout-
t2-t3; t2:=t2/2; Zout:=Zout*t5; Yout:=Yout*t3; t2:=Xout-t2; t1:=t1*t2; Yout:=t1-Yout;
simplify([x5-Xout/Zout^2]), simplify([y5-Yout/Zout^3]); # Check
```

Explicit Formulas using $\mathcal{E}/\mathcal{E}^e$ Coordinates

```
# Twisted Edwards curve (for verification):
x1:=X1/Z1; y1:=Y1/Z1; a:=-1;
x2:=X2/Z2; y2:=Y2/Z2; T2:=X2*Y2/Z2; x5:=X5/Z5; y5:=Y5/Z5; T5:=X5*Y5/Z5;
x3:=(2*x1*y1)/(y1^2+u*a*x1^2); y3:=(y1^2-u*a*x1^2)/(2-y1^2-u*a*x1^2);
x4:=(x3*y3+x2*y2)/(y3*y2+u*a*x3*x2); y4:=(x3*y3-x2*y2)/(x3*y2-y3*x2);
x6:=(x4*y4+x5*y5)/(y4*y5+u*a*x4*x5); y6:=(x4*y4-x5*y5)/(x4*y5-y4*x5);
```

DBL, $2\mathcal{E} \to \mathcal{E}$: $2(X_1 : Y_1 : Z_1) \to (X_{out} : Y_{out} : Z_{out})$. Cost = 4Mul+3Sqr+1Sub+1AddSub+2Mul2+1Mulμ; no contiguous data dependencies

```
Zout:=Z1^2; t1:=2*X1; t2:=X1^2; t1:=t1*Y1; Xout:=u*t2; Yout:=Y1^2; Zout:=2*Zout; t2:=Yout-Xout; Yout:=Yout+Xout; Zout:=Zout-t2; Yout:=t2*Yout; Xout:=t1*Zout; Zout:= t2*Zout;
simplify([x3-Xout/Zout]), simplify([y3-Yout/Zout]); # Check
```

Merged DBL–ADD, $(2\mathcal{E})^e + \mathcal{E}^e \to \mathcal{E}$: $2(X_1 : Y_1 : Z_1) + (X_2 : Y_2 : Z_2 : T_2) \to (X_{out} : Y_{out} : Z_{out})$. Cost = 13Mul+3Sqr+3Sub+1Add+2AddSub+1AddSub2+2Mul2+2Mulμ; no contiguous dependencies

```
# If Z2=1 (Merged DBL-mADD), T1:=T1*Z2 is not needed and the number of multiplies reduces to 12M
Zout:=Z1^2; t1:=2*X1; t2:=X1^2; t1:=t1*Y1; Xout:=u*t2; Yout:=Y1^2; Zout:=2*Zout; t2:=Yout-Xout; Yout:=Xout+Yout; Zout:=Zout-t2; T1:=t1*Yout; Yout:=t2*Yout; Xout:=t1*Zout; Zout:=t2*Zout; t1:=Xout*X2; T1:=T1*Z2; Zout:=Zout*T2; t2:=u*t1; t3:=T1+Zout; Zout:=T1-Zout; T1:=Yout*Y2; Xout:=Xout-Yout; Yout:=X2+Y2; t2:=T1-t2; Xout:=Xout*Yout; Yout:=Zout*t2; t1:=Xout+T1-t1; Zout:=t1*t2; Xout:=t1*t3;
simplify([x4-Xout/Zout]), simplify([y4-Yout/Zout]); # Check
```

Merged DBL–ADDADD, $(2\mathcal{E})^e + \mathcal{E}^e + \mathcal{E}^e \to \mathcal{E}$: $2(X_1 : Y_1 : Z_1) + (X_2 : Y_2 : Z_2 : T_2) + (X_3 : Y_3 : Z_3 : T_3) \to (X_{out} : Y_{out} : Z_{out})$. Cost = 22Mul+3Sqr+5Sub+2Add+3AddSub+2AddSub2+2Mul2+3Mulμ; no contiguous data dependencies

```
# If Z2=1, T1:=T1*Z2 is not needed and the number of multiplies reduces in 1M
# If Z5=1, T1:=T1*Z5 is not needed and the number of multiplies reduces in 1M
Zout:=Z1^2; t1:=2*X1; t2:=X1^2; t1:=t1*Y1; Xout:=u*t2; Yout:=Y1^2; Zout:=2*Zout; t2:=Yout-Xout; Yout:=Xout+Yout; Zout:=Zout-t2; T1:=t1*Yout; Yout:=t2*Yout; Xout:=t1*Zout; Zout:=t2*Zout; t1:=Xout*X2; T1:=T1*Z2; Zout:=Zout*T2; t2:=u*t1; t3:=T1+Zout; Zout:=T1-Zout; T1:=Yout*Y2; Xout:=Xout-Yout; Yout:=X2+Y2; t2:=T1-t2; Xout:=Xout*Yout; Yout:=Zout*t2; Xout:=Xout+T1-t1; T1:=Zout*t3; Zout:=Xout*t2; Xout:=Xout*t3; t1:=Xout*X5; T1:=T1*Z5; Zout:=Zout*T5; t2:=u*t1; t3:=T1+Zout; Zout:=T1-Zout; T1:=Yout*Y5; Xout:=Xout-Yout; Yout:=X5+Y5; t2:=T1-t2; Xout:=Xout*Yout; Yout:=Zout*t2; Xout:=Xout+T1-t1; Zout:=Xout*t2; Xout:=Xout*t3;
simplify([x6-Xout/Zout]), simplify([y6-Yout/Zout]); # Check
```

Appendix C

C1 Optimizing Compressed Squarings

Karabina [Kar10] introduced a new method for computing an exponentiation $g^{|u|}$ in cyclotomic subgroups $\mathbb{G}_{\phi_6}(\mathbb{F}_{p^2})$ using efficient compressed squarings.

Let $g = \sum_{i=0}^{2}(g_{2i}+g_{2i+1}s)t^i \in \mathbb{G}_{\phi_6}(\mathbb{F}_{p^2})$ and $g^2 = \sum_{i=0}^{2}(h_{2i}+h_{2i+1}s)t^i$ where $g_i, h_i \in \mathbb{F}_{p^2}$. Karabina showed that g and g^2 can be compressed to $C(g) = [g_2, g_3, g_4, g_5]$ and $C(g^2) = [h_2, h_3, h_4, h_5]$, respectively, where:

$$h_2 = 2(g_2 + 3\xi B_{4,5}), \quad h_3 = 3(A_{4,5} - (\xi+1)B_{4,5}) - 2g_3,$$
$$h_4 = 3(A_{2,3} - (\xi+1)B_{2,3}) - 2g_4, \quad h_5 = 2(g_5 + 3B_{2,3}), \tag{C.1}$$

with $A_{i,j} = (g_i + g_j)(g_i + \xi g_j)$ and $B_{i,j} = g_i g_j$.

The formulae above have a cost of 4 multiplications and 4 reductions in $\mathbb{F}_{p^2}$. The following improved version was proposed in [AKL+10]:

$$h_2 = 2g_2 + 3\xi(S_{4,5} - S_4 - S_5), \quad h_3 = 3(S_4 + \xi S_5) - 2g_3,$$
$$h_4 = 3(S_2 + \xi S_3) - 2g_4, \quad h_2 = 2g_5 + 3(S_{2,3} - S_2 - S_3), \tag{C.2}$$

with $S_{i,j} = (g_i + g_j)^2$ and $S_i = g_i^2$.

It is straightforward to see that the formulae above have a cost of 6 integer squarings and only 4 reductions in $\mathbb{F}_{p^2}$ by applying lazy reduction.

In total, the computation of an exponentiation $g^{|u|}$ involving compression and decompression in the cyclotomic subgroup $\mathbb{G}_{\phi_6}(\mathbb{F}_{p^2})$ requires 62 compressed squarings (C.2) during compression, $1i + 9m + 6s + 22a$ for decompression and 2 $\mathbb{F}_{p^{12}}$ multiplications to obtain the final result. Then, the total cost when applying the generalized lazy reduction technique is given by (see [AKL+10, Section 5.2] for complete details):

$$Exp = 62(6s_u + 4r + 31a) + (1i + 9m + 6s + 22a) + 2(18m_u + 6r + 110a)$$
$$= 1i + 36m_u + 372s_u + 9m + 6s + 260r + 2164a\ .$$

In contrast, the traditional computation would cost (using lazy reduction below $\mathbb{F}_{p^2}$ only):

$$Exp' = 62(4m + 27a) + (1i + 9m + 6s + 22a) + 2(18m + 67a)$$
$$= 1i + 293m + 6s + 1830a\ ,$$

Hence, our technique reduces the number of reductions in $\mathbb{F}_{p^2}$ in about 8% (from 299 to 275) in one exponentiation $g^{|u|}$ computed with the new compressed squarings.

PERMISSIONS

Partial results that have been included and extended in this Dissertation appear in [LM08b, LG09, LG09b, LG10, AKL+10]. Complete references to original publications have been included in the Bibliography in compliance with Springer's copyright, which states: "The Author retains the right to use his/her Contribution for his/her further scientific career by including the final published paper in his/her dissertation or doctoral thesis provided acknowledgement is given to the original source of publication."

Bibliography

[ACD+05] R. Avanzi, H. Cohen, D. Doche, G. Frey, T. Lange, K. Nguyen and F. Vercauteren, "Handbook of Elliptic and Hyperelliptic Curve Cryptography," CRC Press, 2005.

[Adi10] J. Adikari, "Efficient Algorithms for Elliptic Curve Cryptography," *PhD. Thesis,* University of Calgary, 2010. Available online at: https://ece.uwaterloo.ca/~jadikari/

[ADI10] J. Adikari, V. Dimitrov and L. Imbert, "Hybrid Binary-Ternary Number System for Elliptic Curve Cryptosystems," *IEEE Transactions on Computers,* Vol. 60(2), pp. 254-265, 2010.

[AHP08] R. Avanzi, C. Heuberger and H. Prodinger, "Redundant τ-adic Expansions I: Non-Adjacent Digit Sets and their Applications to Scalar Multiplication," *Cryptology ePrint Archive,* Report 2008/148, 2008.

[AKL+10] D.F. Aranha, K. Karabina, P. Longa, C. Gebotys and J. Lopez, "Faster Explicit Formulas for Computing Pairings over Ordinary Curves," *Advances in Cryptology - Eurocrypt 2011,*Springer, 2011 (to appear).

[AMD] Advanced Micro Devices, "AMD64 Architecture Programmer's Manual, Volume 1: Application Programming," 2009. Available online at: http://developer.amd.com/DOCUMENTATION/GUIDES/Pages/default.aspx

[Ava04] R. Avanzi, "Aspects of Hyperelliptic Curves over Large Prime Fields in Software Implementations," *International Workshop on Cryptographic Hardware and Embedded Systems (CHES 2004),* LNCS Vol. 3156, pp. 148-162, Springer, 2004.

[Ava05] R. Avanzi, "A Note on the Signed Sliding Window Integer Recoding and a Left-to-Right Analogue," *International Workshop on Selected Areas in Cryptography (SAC 2004),* LNCS Vol. 3357, pp. 130-143, Springer, 2005.

[BBL+07] D. Bernstein, P. Birkner, T. Lange and C. Peters, "Optimizing Double-Base Elliptic-Curve Single-Scalar Multiplication," *International Conference on Cryptolology - Indocrypt 2007,* LNCS Vol. 4859, pp. 167-182, Springer, 2007.

[BBT+08] D. Bernstein, P. Birkner, T. Lange, C. Peters and M. Joye, "Twisted Edwards Curves," *Progress in Cryptology - Africacrypt 2008,* LNCS Vol. 5023, pp. 389-405, Springer, 2008.

[BCH+00] M. Brown, D. Cheung, D. Hankerson, J. Lopez, M. Kirkup and A. Menezes, "PGP in Constrained Wireless Devices," *Usenix Security Symposium,* pp. 247-261, 2000.

[Ber06] D. Bernstein, "Curve25519: New Diffie-Hellman Speed Records," *International Conference on Practice and Theory in Public Key Cryptography (PKC 2006),* LNCS Vol. 3958, pp. 207-228, Springer, 2006.

[BF01] D. Boneh and M. Franklin, "Identity-Based Encryption from the Weil Pairing," *Advances in Cryptology - Crypto 2001,* LNCS Vol. 2139, pp. 213-229, Springer, 2001.

[BG04] D. Brown and R. Gallant, "The Static Diffie-Hellman Problem," *Cryptology ePrint Archive,* Report 2004/306, 2004.

[BGM+10] J. Beuchat, J.E. González-Díaz, S. Mitsunari, E. Okamoto, F. Rodríguez-Henríquez and T. Teruya, "High-Speed Software Implementation of the Optimal Ate Pairing over Barreto-Naehrig Curves," *International Conference on Pairing-Based Cryptography (Pairing 2010),* LNCS Vol. 6487, pp. 21-39, Springer, 2010.

[BGO+07] P.S.L.M. Barreto, S. Galbraith, C. O'hEigeartaigh and M. Scott, "Efficient Pairing Computation on Supersingular Abelian Varieties," *Designs, Codes and Cryptography,* Vol. 42, pp. 239-271, 2007.

[BHL+01] M. Brown, D. Hankerson, J. Lopez and A. Menezes, "Software Implementation of the NIST Elliptic Curves over Prime Fields," *Topics in Cryptology - CT-RSA 2001,* LNCS Vol. 2020, pp. 250-265, Springer, 2001.

[BJ03] O. Billet and M. Joye, "Fast Point Multiplication on Elliptic Curves through Isogenies," *Applied Algebra, Algebraic Algorithms, and Error Correcting Codes Symposium (AAECC-15),* LNCS Vol. 2643, pp. 43-50, Springer, 2003.

[BJ03b] O. Billet and M. Joye, "The Jacobi Model of an Elliptic Curve and Side-Channel Analysis," *International Conference on Applied Algebra, Algebraic Algorithms and Error-Correcting Codes (AAECC '03),* LNCS Vol. 2643, pp. 34-42, Springer, 2003.

[BKL+02] P.S.L.M. Barreto, H.Y. Kim, B. Lynn and M. Scott, "Efficient Algorithms for Pairing-Based Cryptosystems," *Advances in Cryptology - Crypto 2002,* LNCS Vol. 2442, pp. 354-368, Springer, 2002.

[BL07] D. Bernstein and T. Lange, "Faster Addition and Doubling on Elliptic Curves," *Advances in Cryptology - Asiacrypt 2007,* LNCS Vol. 4833, pp. 29-50, Springer, 2007.

[BL07b] D. Bernstein and T. Lange, "Inverted Edwards Coordinates," *Applied Algebra, Algebraic Algorithms, and Error Correcting Codes Symposium (AAECC-17),* LNCS Vol. 4851, pp. 20-27, Springer, 2007.

[BL08] D. Bernstein and T. Lange, "Analysis and Optimization of Elliptic-Curve Single-Scalar Multiplication," *Finite Fields and Applications: Proceedings of Fq8,* Vol. 461, pp. 1-18, 2008.

[BLS03] P.S.L.M. Barreto, B. Lynn and M. Scott, "On the Selection of Pairing-Friendly Groups," *Int. Workshop on Selected Areas in Cryptography (SAC 2003),* LNCS Vol. 3006, pp. 17-25, Springer, 2003.

[BLS03b] P.S.L.M. Barreto, B. Lynn and M. Scott, "Constructing Elliptic Curves with Prescribed Embedding Degrees," *Security in Communication Networks,* LNCS Vol. 2576, pp. 257-267, Springer, 2003.

[BLS04] D. Boneh, B. Lynn and H. Shacham, "Short signatures from the Weil pairing," *Journal of Cryptology,* Vol. 17, pp. 297-319, 2004.

[BLS04b] P.S.L.M. Barreto, B. Lynn and M. Scott, "Efficient Implementation of Pairing-Based Cryptosystems," *Journal of Cryptology,* Vol. 17, pp. 321-334, 2004.

[BN05] P.S.L.M. Barreto and M. Naehrig, "Pairing-Friendly Elliptic Curves of Prime Order," *International Workshop on Selected Areas in Cryptography (SAC 2005),* LNCS Vol. 3897, pp. 319-331, Springer, 2005.

[BPP07] R. Barua, S.K. Pandey and R. Pankaj, "Efficient Window-Based Scalar Multiplication on Elliptic Curves using Double-Base Number System," *International Conference on Cryptolology - Indocrypt 2007,* LNCS Vol. 4859, pp. 351-360, Springer, 2007.

[BS10] N. Benger and M. Scott, "Constructing Tower Extensions of Finite Fields for Implementation of Pairing-Based Cryptography," *International Workshop on Arithmetic of Finite Fields (WAIFI 2010),* LNCS Vol. 6087, pp. 180-189, Springer, 2010.

[BW05] F. Brezing and Z. Weng, "Elliptic Curves Suitable for Pairing Based Cryptography," *Designs, Codes and Cryptography,* Vol. 37(1), pp. 133-141, 2005.

[CC86] D.V. Chudnovsky and G.V. Chudnovsky, "Sequences of Numbers Generated by Addition in Formal Groups and New Primality and Factorization Tests," *Advances in Applied Mathematics,* Vol. 7(4), pp. 385-434, 1986.

[CCY96] C.-Y. Chen, C.-C. Chang and W.-P. Yang, "Hybrid Method for Modular Exponentiation with Precomputation," *Electronics Letters,* Vol. 32(6), pp. 540-541, 1996.

[CH07] J. Chung and M.A. Hasan, "Asymmetric Squaring Formulae," *IEEE Symposium on Computer Arithmetic (ARITH-18 2007),* pp. 113-122, 2007.

[CHB+09] C. Costello, H. Hisil, C. Boyd, J. Gonzalez Nieto and K.K. Wong, "Faster Pairings on Special Weierstrass Curves," *International Conference on Pairing-Based Cryptography (Pairing 2009),* LNCS Vol. 5671, pp. 89-101, Springer, 2009.

[CJL+06] M. Ciet, M. Joye, K. Lauter and P.L. Montgomery, "Trading Inversions for Multiplications in Elliptic Curve Cryptography," *Designs, Codes and Cryptography,* Vol. 39(2), pp. 189-206, 2006.

[CLN10] C. Costello, T. Lange and M. Naehrig, "Faster Pairing Computations on Curves with High-Degree Twists," *International Conference on Practice and Theory in Public Key Cryptography (PKC 2010),* LNCS Vol. 6056, pp. 224-242, Springer, 2010.

[CMO98] H. Cohen, A. Miyaji and T. Ono, "Efficient Elliptic Curve Exponentiation using Mixed Coordinates," *Advances in Cryptology - Asiacrypt '98,* LNCS Vol. 1514, pp. 51-65, Springer, 1998.

[Com90] P.G. Comba, "Exponentiation Cryptosystems on the IBM PC," *IBM Systems Journal,* Vol. 29, pp. 526-538, 1990.

[CS09] N. Costigan and P. Schwabe, "Fast Elliptic-Curve Cryptography on the Cell Broadband Engine," *Progress in Cryptology - Africacrypt 2009,* LNCS Vol. 5580, pp. 368-385, Springer, 2009.

[DC95] V. Dimitrov and T. Cooklev, "Two Algorithms for Modular Exponentiation based on Nonstandard Arithmetics," *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Science (Special Issue on Cryptography and Information Security)*, Vol. E78-A, pp. 82−87, 1995.

[DH76] W. Diffie and M. Hellman, "New Directions in Cryptography," *IEEE Transactions on Information Theory,* Vol. 22(6), pp. 644-654, 1976.

[DH08] C. Doche and L. Habsieger, "A Tree-Based Approach for Computing Double-Base Chains," *Australasian Conference on Information Security and Privacy (ACISP 2008),* LNCS Vol. 5107, pp. 433-446, Springer, 2008.

[DI06] C. Doche and L. Imbert, "Extended Double-Base Number System with Applications to Elliptic Curve Cryptography," *Progress in Cryptology - Indocrypt 2006,* LNCS Vol. 4329, pp. 335-348, Springer, 2006.

[DIK06] C. Doche, T. Icart and D.R. Kohel, "Efficient Scalar Multiplication by Isogeny Decompositions," *International Conference on Practice and Theory of Public Key Cryptography (PKC 2006),* LNCS Vol. 3958, pp. 191-206, Springer, 2006.

[DIM05] V. Dimitrov, L. Imbert and P.K. Mishra, "Efficient and Secure Elliptic Curve Point Multiplication using Double-Base Chains," *Advances in Cryptology - Asiacrypt 2005,* LNCS Vol. 3788, pp. 59-78, Springer, 2005.

[DJM98] V. Dimitrov, G. Jullien and W. Miller, "An Algorithm for Modular Exponentiation," *Information Processing Letters*, Vol. 66. pp. 155-159, 1998.

[DKS09] C. Doche, D.R. Kohel and F. Sica, "Double-Base Number System for Multi-Scalar Multiplications," *Advances in Cryptology - Eurocrypt 2009,* LNCS Vol. 5479, pp. 502-517, Springer, 2009.

[DOS07] E. Dahmen, K. Okeya and D. Schepers, "Affine Precomputation with Sole Inversion in Elliptic Curve Cryptography," *Australasian Conference on Information Security and Privacy (ACISP 2007),* LNCS Vol. 4586, pp. 245-258, Springer, 2007.

[DOT07] E. Dahmen, K. Okeya and T. Takagi, "A New Upper Bound for the Minimal Density of Joint Representations in Elliptic Curve Cryptosystems," *IEICE Transactions on Fundamentals of Electronics,* Vol. E90-A(5), pp. 952-959, 2007.

[DSD07] A.J. Devegili, M. Scott and R. Dahab, "Implementing Cryptographic Pairings over Barreto-Naehrig Curves," *International Conference on Pairing-Based Cryptography (Pairing 2007),* LNCS Vol. 4575, pp. 197-207, Springer, 2007.

[Edw07] H. Edwards, "A Normal Form for Elliptic Curves," *Bulletin of the American Mathematical Society,* Vol. 44(3), pp. 393-422, 2007.

[ElG84] T. ElGamal, "A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms," Advances in *Cryptology – Crypto '84,* LNCS Vol. 196, pp. 10-18, Springer, 1985.

[Elm06] L. Elmegaard-Fessel, "Efficient Scalar Multiplication and Security against Power Analysis in Cryptosystems based on the NIST Elliptic Curves over Prime Fields," *Master's Thesis,* University of Copenhagen, 2006.

[EYK09] S.S. Erdem, T. Yanik and C.K. Koç, "Fast Finite Field Multiplication," *Cryptographic Engineering, Chapter 5,* Springer, 2009.

[Fog1] A. Fog, "Instruction Tables: Lists of Instruction Latencies, Throughputs and Micro-operation Breakdowns for Intel, AMD and VIA CPUs," 2009. Available online at: http://www.agner.org/optimize/#manuals, accessed on January 2010.

[Fog2] A. Fog, "The Microarchitecture of Intel, AMD and VIA CPUs," 2009. Available online at: http://www.agner.org/optimize/#manuals, accessed on January 2010.

[Fre06] D. Freeman, "Constructing Pairing-Friendly Elliptic Curves with Embedding Degree 10," *International Symposium on Algorithmic Number Theory (ANTS-VII),* LNCS Vol. 4076, pp. 452-465, Springer, 2006.

[FVV09] J. Fan, F. Vercauteren and I. Verbauwhede, "Faster Fp-Arithmetic for Cryptographic Pairings on Barreto-Naehrig Curves," *International Workshop on Cryptographic Hardware and Embedded Systems (CHES 2009),* LNCS Vol. 5747, pp. 240-253, Springer, 2009.

[GAS+05] J. Großschädl, R. Avanzi, E. Savaş and S. Tillich, "Energy-Efficient Software Implementation of Long Integer Modular Arithmetic," *International Workshop on Cryptographic Hardware and Embedded Systems (CHES 2005),* LNCS Vol. 3659, pp. 75-90, Springer, 2005.

[Gau09] P. Gaudry, "Index Calculus for Abelian Varieties of Small Dimension and the Elliptic Curve Discrete Logarithm Problem," *Journal of Symbolic Computation,* Vol. 44, pp. 1690-1702, 2009.

[GGC02] V. Gupta, S. Gupta and S. Chang, "Performance Analysis of Elliptic Curve Cryptography for SSL," *ACM Workshop on Wireless Security (WiSe)*, Mobicom 2002, 2002.

[GLS08] S. Galbraith, X. Lin and M. Scott, "Endomorphisms for Faster Elliptic Curve Cryptography on a Large Class of Curves," *Cryptology ePrint Archive,* Report 2008/194, 2008.

[GLS09] S. Galbraith, X. Lin and M. Scott, "Endomorphisms for Faster Elliptic Curve Cryptography on a Large Class of Curves," *Advances in Cryptology - Eurocrypt 2009,* LNCS Vol. 5479, pp. 518-535, Springer, 2009.

[GLV01] R. Gallant, R. Lambert and S. Vanstone, "Faster Point Multiplication on Elliptic Curves with Efficient Endomorphisms," *Advances in Cryptology - Crypto 2001,* LNCS Vol. 2139, pp. 190-200, Springer, 2001.

[GMJ10] R.R. Goundar, A. Miyaji and M. Joye, "Co-Z Addition Formulæ and Binary Ladders on Elliptic Curves," *International Workshop on Cryptographic Hardware and Embedded Systems (CHES 2010),* LNCS Vol. 6225, pp. 65-79, Springer, 2010.

[Gor93] D. Gordon, "Discrete Logarithms in GF(p) using the Number Field Sieve," *SIAM Journal on Discrete Mathematics,* Vol. 6, pp. 124-138, 1993.

[GPW+04] N. Gura, A. Patel, A. Wander, H. Eberle and S.C. Shantz, "Comparing Elliptic Curve Cryptography and RSA on 8-bit CPUs," *International Workshop on Cryptographic Hardware and Embedded Systems (CHES 2004),* LNCS Vol. 3156, pp. 119-132, Springer, 2004.

[Gra10] R. Granger, "On the Static Diffie-Hellman Problem on Elliptic Curves over Extension Fields," *Advances in Cryptology - Asiacrypt 2010,* LNCS Vol. 6477, pp. 283-302, Springer, 2010.

[GSF04] V. Gupta, D. Stebila and S. Fung, "Speeding up Secure Web Transactions using Elliptic Curve Cryptography," *Annual Network and Distributed System Security (NDSS) Symposium*, 2004.

[GT07b] P. Gaudry and E. Thomé, "The mpFq Library and Implementing Curve-Based Key Exchanges," *SPEED 2007,* pp. 49-64, 2007.

[His10] H. Hisil, "Elliptic Curves, Group Law, and Efficient Computation," *PhD. Thesis,* Queensland University of Technology, 2010. Available online at: http://eprints.qut.edu.au/33233/

[HMS08] D. Hankerson, A. Menezes and M. Scott, "Software Implementation of Pairings," *Identity-Based Cryptography, Chapter 12,* pp. 188-206, IOS Press, 2008.

[HMV04] D. Hankerson, A. Menezes and S. Vanstone, "Guide to Elliptic Curve Cryptography," Springer, 2004.

[HSV06] F. Hess, N. Smart and F. Vercauteren, "The Eta Pairing Revisited," *IEEE Transactions on Information Theory,* Vol. 52(10), pp. 4595-4602, 2006.

[HT00] A. Higuchi and N. Takagi, "A Fast Addition Algorithm for Elliptic Curve Arithmetic in GF(2^n) using Projective Coordinates," *Information Processing Letters,* Vol. 76(3), pp. 101-103, 2000.

[HWC+07] H. Hisil, K. Wong, G. Carter and E. Dawson, "Faster Group Operations on Elliptic Curves," *Cryptology ePrint Archive,* Report 2007/441, 2007.

[HWC+08] H. Hisil, K. Wong, G. Carter and E. Dawson, "Twisted Edwards Curves Revisited," *Advances in Cryptology - Asiacrypt 2008,* LNCS Vol. 5350, pp. 326-343, Springer, 2008.

[HWC+08b] H. Hisil, K. Wong, G. Carter and E. Dawson, "An Intersection Form for Jacobi-Quartic Curves," *Personal Communication,* 2008.

[HWC+09] H. Hisil, K. Wong, G. Carter and E. Dawson, "Jacobi Quartic Curves Revisited," *Australasian Conference on Information Security and Privacy (ACISP 2009),* LNCS Vol. 5594, pp. 452-468, Springer, 2009.

[IEEE00] The Institute of Electrical and Electronics Engineers (IEEE), "IEEE Standard Specifications for Public-Key Cryptography," IEEE Std 1363-2000, 2000.

[IEEE08] The Institute of Electrical and Electronics Engineers (IEEE), "IEEE Draft Standard for Identity-based Public-key Cryptography Using Pairings," IEEE P1636.3/D1, 2008.

[Intel] Intel Corporation, "Intel64 and IA-32 Architectures Software Developer's Manual, Volume 1: Basic Architecture," 2009. Available online at: http://www.intel.com/products/processor/manuals/

[JFS07] K. Jarvinen, J. Forsten and J. Skytta, "FPGA Design of Self-Certified Signature Verification on Koblitz Curves," *International Workshop on Cryptographic Hardware and Embedded Systems (CHES 2007),* LNCS Vol. 4727, pp. 256-271, Springer, 2007.

[Jou00] A. Joux, "A One Round Protocol for Tripartite Diffie-Hellman," *Algorithmic Number Theory Symposium IV,* LNCS Vol. 1838, pp. 385-394, Springer, 2000.

[Kar95] A.A. Karatsuba, "The Complexity of Computations," *Proceedings of the Steklov Institute of Mathematics,* Vol. 211, pp. 169-183, 1995.

[Kar10] K. Karabina, "Squaring in Cyclotomic Subgroups," *Cryptology ePrint Archive,* Report 2010/542, 2010.

[KM05] N. Koblitz and A. Menezes, "Pairing-Based Cryptography at High Security Levels," *International Conference on Cryptography and Coding,* LNCS Vol. 3796, pp. 13-36, Springer, 2005.

[Kob87] N. Koblitz, "Elliptic Curve Cryptosystems," *Mathematics of Computation,* Vol. 48, pp. 203-209, 1987.

[KZZ04] B. Kuang, Y. Zhu and Y. Zhang, "An Improved Algorithm for uP+vQ using JSF_3," *International Conference on Applied Cryptography and Network Security (ACNS 2004),* LNCS Vol. 3089, pp. 467-478, Springer, 2004.

[Lau04] K. Lauter, "The Advantages of Elliptic Cryptography for Wireless Security," *IEEE Wireless Communications,* Vol. 11(1), pp. 62-67, 2004.

[LD99] J. Lopez and R. Dahab, "Improved Algorithms for Elliptic Curve Arithmetic in GF(2^n)," *International Workshop on Selected Areas in Cryptography (SAC '98),* LNCS Vol. 1556, pp. 201-212, Springer, 1999.

[Len87] H. Lenstra, "Factoring Integers with Elliptic Curves," *Annals of Mathematics,* Vol. 126, pp. 649-673, 1987.

[LG08] P. Longa and C. Gebotys, "Setting Speed Records with the (Fractional) Multibase Non-Adjacent Form Method for Efficient Elliptic Curve Scalar Multiplication," *CACR Technical Report,* CACR 2008-06, 2008.

[LG09] P. Longa and C. Gebotys, "Fast Multibase Methods and Other Several Optimizations for Elliptic Curve Scalar Multiplication," *International Conference on Practice and Theory in Public Key Cryptography (PKC 2009),* LNCS Vol. 5443, pp. 443-462, Springer, 2009.

[LG09b] P. Longa and C. Gebotys, "Novel Precomputation Schemes for Elliptic Curve Cryptosystems," *International Conference on Applied Cryptography and Network Security (ACNS 2009),* LNCS Vol. 5536, pp. 71-88, Springer, 2009.

[LG10] P. Longa and C. Gebotys, "Efficient Techniques for High-Speed Elliptic Curve Cryptography," *Workshop on Cryptographic Hardware and Embedded Systems (CHES 2010),* LNCS Vol. 6225, pp. 80-94, Springer, 2010.

[LH00] C.H. Lim and H.S. Hwang, "Fast Implementation of Elliptic Curve Arithmetic in GF(p^m)," *International Conference on Practice and Theory in Public Key Cryptography (PKC 2000),* LNCS Vol. 1751, pp. 405-421, Springer, 2000.

[LLM+93] A. Lenstra, H. Lenstra, M. Manasse and J. Pollard, "The Number Field Sieve," *The Development of the Number Field Sieve,* LNCS Vol. 1554, pp. 11-42, Springer, 1993.

[LLP09] E. Lee, H.-S. Lee and C.-M. Park, "Efficient and Generalized Pairing Computation on Abelian Varieties," *IEEE Transactions on Information Theory,* Vol. 55(4), pp. 1793-1803, 2009.

[LM08] P. Longa and A. Miri, "Fast and Flexible Elliptic Curve Point Arithmetic over Prime Fields," *IEEE Transactions on Computers,* Vol. 57(3), pp. 289-302, 2008.

[LM08b] P. Longa and A. Miri, "New Composite Operations and Precomputation Scheme for Elliptic Curve Cryptosystems over Prime Fields," *International Conference on Practice and Theory in Public Key Cryptography (PKC 2008),* LNCS Vol. 4939, pp. 229-247, Springer, 2008.

[LM08c] P. Longa and A. Miri, "New Multibase Non-Adjacent Form Scalar Multiplication and its Application to Elliptic Curve Cryptosystems (extended version)," *Cryptology ePrint Archive,* Report 2008/052, 2008.

[Lon07] P. Longa, "Accelerating the Scalar Multiplication on Elliptic Curve Cryptosystems over Prime Fields," *Master's Thesis,* University of Ottawa, 2007. Available online at: http://patricklonga.bravehost.com.proxy.lib.uwaterloo.ca/publications.html

[Lon08] P. Longa, "ECC Point Arithmetic Formulae (EPAF) Database," 2008. Available online at: http://patricklonga.bravehost.com/jacobian.html#jac

[Lon10] P. Longa, "Speed Benchmarks for Elliptic Curve Scalar Multiplication," 2010. Available online at: http://www.patricklonga.bravehost.com/speed_ecc.html#speed

[Lon10b] P. Longa, "Speed Benchmarks for Pairings over Ordinary Curves," 2010. Available online at: http://patricklonga.bravehost.com/speed_pairing.html#speed

[MD07] P.K. Mishra and V. Dimitrov, "Efficient Quintuple Formulas for Elliptic Curves and Efficient Scalar Multiplication using Multibase Number Representation," *Information Security Conference (ISC 2007),* LNCS Vol. 4779, pp. 390-406, Springer, 2007.

[Mel07] N. Meloni, "New Point Addition Formulae for ECC Applications," *International Workshop on Arithmetic of Finite Fields (WAIFI 2007),* LNCS Vol. 4547, pp. 189-201, Springer, 2007.

[Men09] A. Menezes, "An Introduction to Pairing-Based Cryptography," *Recent Trends in Cryptography*, Vol. 477 of Contemporary Mathematics, pp. 47-65, AMS-RSME, 2009.

[MH09] N. Meloni and A. Hasan, "Elliptic Curve Scalar Multiplication Combining Yao's Algorithm and Double Bases," *International Workshop on Cryptographic Hardware and Embedded Systems (CHES 2009),* Lecture Notes in Computer Science Vol. 5747, pp. 304-316, Springer, 2009.

[Mil86] V. Miller, "Use of Elliptic-Curves in Cryptography," *Advances in Cryptology - Crypto '85,* LNCS Vol. 218, pp. 417-426, Springer, 1986.

[Mil86b] V. Miller, "Short Programs for Functions on Curves," 1986. Available online at: http://crypto.stanford.edu/miller

[Mil04] V. Miller, "The Weil Pairing, and its Efficient Calculation," *Journal of Cryptology,* Vol. 17, pp. 235-261, 2004.

[MIR] M. Scott, "Multiprecision Integer and Rational Arithmetic C/C++ Library (MIRACL)." Available online at: http://www.shamus.ie/

[Möl01] B. Möller, "Algorithms for Multi-Exponentiation," *Selected Areas in Cryptography (SAC 2001),* LNCS Vol. 2259, pp. 165-180, Springer, 2001.

[Möl03] B. Möller, "Improved Techniques for Fast Exponentiation," *International Conference on Information Security and Cryptology (ICISC 2002),* LNCS Vol. 2587, pp. 298-312, Springer, 2003.

[Möl05] B. Möller, "Fractional Windows Revisited: Improved Signed-Digit Representations for Efficient Exponentiation," *International Conference on Information Security and Cryptology (ICISC 2004),* LNCS Vol. 3506, pp. 137-153, Springer, 2005.

[Mon85] P.L. Montgomery, "Modular Multiplication without Trial Division," *Mathematics of Computation,* Vol. 44, pp. 519-521, 1985.

[Mon87] P.L. Montgomery, "Speeding the Pollard and Elliptic Curve Methods of Factorization," *Mathematics of Computation,* Vol. 48, pp. 243-264, 1987.

[Mor90] F. Morain and J. Olivos, "Speeding up the Computations on an Elliptic Curve using Addition-Subtraction Chains," *Theoretical Informatics and Applications,* Vol. 24(6), pp. 531-544, 1990.

[MOV93] A. Menezes, T. Okamoto and S. Vanstone, "Reducing Elliptic Curve Logarithms to Logarithms in a Finite Field," *IEEE Transactions on Information Theory,* Vol. 39, pp. 1639-1646, 1993.

[mpFq] P. Gaudry and E. Thomé, "mpFq – A Finite Field Library." Available online at: http://mpfq.gforge.inria.fr/mpfq-1.0-rc2.tar.gz

[NIST00] National Institute of Standards and Technology (NIST), "Digital Signature Standard (DSS)," FIPS PUB 186-2, 2000. Available online at: http://csrc.nist.gov.proxy.lib.uwaterloo.ca/publications/PubsFIPS.html

[NIST07] National Institute of Standards and Technology (NIST), "Recommendation for Key Management - Part 1: General (Revised)," NIST Special Publication 800-57, 2007. Available online at: http://csrc.nist.gov/publications/PubsSPs.html

[NIST09] National Institute of Standards and Technology (NIST), "Digital Signature Standard (DSS)," FIPS PUB 186-3, 2009. Available online at: http://csrc.nist.gov.proxy.lib.uwaterloo.ca/publications/PubsFIPS.html

[NNS10] M. Naehrig, R. Niederhagen and P. Schwabe, "New Software Speed Records for Cryptographic Pairings," *Progress in Cryptology - Latincrypt 2010,* LNCS Vol. 6212, pp. 109-123, Springer, 2010.

[NSA09] U.S. National Security Agency (NSA), "NSA Suite B Cryptography," Fact Sheet NSA Suite B Cryptography, 2009. Available online at: http://www.nsa.gov.proxy.lib.uwaterloo.ca/ia/programs/suiteb_cryptography/index.shtml

[OKN10] K. Okeya, H. Kato and Y. Nogami, "Width-3 Joint Sparse Form," *International Conference on Information Security, Practice and Experience (ISPEC 2010),* LNCS Vol. 6047, pp. 67-84, Springer, 2010.

[OTV05] k. Okeya, T. Takagi and C. Vuillaume, "Efficient Representations on Koblitz Curves with Resistance to Side Channel Attacks," *Australasian Conference on Information Security and Privacy (ACISP 2005),* LNCS Vol. 3574, pp. 218-229, Springer, 2005.

[Pol78] J. Pollard, "Monte Carlo Methods for Index Computation mod p," *Mathematics of Computation,* Vol. 32, pp. 918-924, 1978.

[Pro03] J. Proos, "Joint Sparse Forms and Generating Zero Columns when Combing," *Technical Report CORR 2003-23,* University of Waterloo, 2003.

[PSN+10] G. Pereira, M. Simplicio Jr, M. Naehrig and P.S.L.M. Barreto, "A Family of Implementation-Friendly BN Elliptic Curves," *Cryptology ePrint Archive,* Report 2010/429, 2010.

[Rei60] G.W. Reitwiesner, "Binary Arithmetic," *Advances in Computers,* Vol. 1, pp. 231-308, 1960.

[RSA78] R. Rivest, A. Shamir and L. Adleman, "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems," *Communications of the ACM*, Vol. 21(2), pp. 120-126, 1978.

[SB06] M. Scott and P.S.L.M. Barreto, "Generating more MNT Elliptic Curves," *Designs, Codes and Cryptography,* Vol. 38(2), pp. 209-217, 2006.

[SBC+09] M. Scott, N. Benger, M. Charlemagne, L. Dominguez Perez and E. Kachisa, "On the Final Exponentiation for Calculating Pairings on Ordinary Elliptic Curves," *International Conference on Pairing-Based Cryptography (Pairing 2009),* LNCS Vol. 5671, pp. 78-88, Springer, 2009.

[Sch10] O. Schirokauer, "The Number Field Sieve for Integers of Low Weight," *Mathematics of Computation,* Vol. 79(269), pp. 583-602, 2010.

[Sco07] M. Scott, "Implementing Cryptographic Pairings," *International Conference on Pairing-Based Cryptography (Pairing 2007),* LNCS Vol. 4575, pp. 177-196, Springer, 2007.

[Sco08] M. Scott, "A Faster Way to Do ECC," Talk at *the 12th Workshop on Elliptic Curve Cryptography (ECC 2008),* 2008. Available online at: http://www.hyperelliptic.org/tanja/conf/ECC08/

[SEI10] V. Suppakitpaisarn, M. Edahiro and H. Imai, "Optimal Average Joint Hamming Weight and Minimal Weight Conversion of d Integers," *Cryptology ePrint Archive,* Report 2010/300, 2010.

[SEI11] V. Suppakitpaisarn, M. Edahiro and H. Imai, "Fast Elliptic Curve Cryptography using Optimal Double-Base Chains," *Cryptology ePrint Archive,* Report 2011/030, 2011.

[SG08] R. Szerwinski and T. Güneysu, "Exploiting the Power of GPUs for Asymmetric Cryptography," *International Workshop on Cryptographic Hardware and Embedded Systems (CHES 2008),* LNCS Vol. 5154, pp. 79-99, Springer, 2008.

[Sma99] N.P. Smart, "The Discrete Logarithm Problem on Elliptic Curves of Trace One," *Journal of Cryptology,* Vol. 12, pp. 193-196, 1999.

[Sma01] N.P. Smart, "The Hessian Form of an Elliptic Curve," *International Workshop on Cryptographic Hardware and Embedded Systems (CHES 2001),* LNCS Vol. 2162, pp. 118-125, Springer, 2001.

[SOK00] R. Sakai, K. Ohgishi and M. Kasahara, "Cryptosystems Based on Pairings," *The 2000 Symposium on Cryptography and Information Security,* 2000.

[Sol00] J. Solinas, "Efficient Arithmetic on Koblitz Curves," *Designs, Codes and Cryptography,* Vol. 19(2-3), pp. 195-249, 2000.

[Sol01] J. Solinas, "Low-Weight Binary Representations for Pairs of Integers," *Technical Report CORR 2001-41,* University of Waterloo, 2001.

[UWL+07] O. Ugus, D. Westhoff, R. Laue, A. Shoufan and S.A. Huss, "Optimized Implementation of Elliptic Curve Based Additive Homomorphic Encryption for Wireless Sensor Networks," *Workshop on Embedded Systems Security (WESS 2007),* 2007.

[Ver01] E. Verheul, "Self-Blindable Credential Certificates from the Weil Pairing," *Advances in Cryptology - Asiacrypt 2001,* LNCS Vol. 2248, pp. 533-551, Springer, 2002.

[Ver10] F. Vercauteren, "Optimal Pairings," *IEEE Transactions on Information Theory,* Vol. 56(1), pp. 455-461, 2010.

[Wal98] C.D. Walter, "Exponentiation using Division Chains," *IEEE Transactions on Computers,* Vol. 47(7), pp. 757-765, 1998.

[Wal02] C.D. Walter, "MIST: an Efficient, Randomized Exponentiation Algorithm for Resisting Power Analysis," *Topics in Cryptology – CT-RSA 2002*, Vol. 2271, pp. 142-174, 2002.

[Wal11] C.D. Walter, "Fast Scalar Multiplication for ECC over GF(p) Using Division Chains," *International Workshop on Information Security Applications (WISA 2010),* LNCS Vol. 6513, pp. 61-75, Springer, 2011.

[WD98] D. Weber and T. Denny, "The Solution of McCurley's Discrete Log Challenge," *Advances in Cryptology - Crypto '98,* LNCS Vol. 1462, pp. 458-471, Springer, 1998.

[XB01] S.-B. Xu and L. Batina, "Efficient Implementation of Elliptic Curve Cryptosystems on an ARM7 with Hardware Accelerator," *International Conference on Information and Communications Security (ICICS '01),* LNCS Vol. 2200, pp. 11-16, Springer, 2001.

[YSK02] T. Yanik, E. Savaş and C.K. Koç, "Incomplete Reduction in Modular Arithmetic," *IEE Proceedings of Computers and Digital Techniques,* Vol. 149(2), pp. 46-52, 2002.

[ZZH08] C.-A. Zhao, F. Zhang and J. Huang, "A Note on the Ate Pairing," *International Journal of Information Security,* Vol. 7(6), pp. 379-382, 2008.