Centro de Investigación y de Estudios Avanzados del Instituto Politécnico Nacional

Unidad Zacatenco

Departamento de Computación

Cryptography in small-characteristic finite fields

Tesis que presenta

Thomaz Eduardo de Figueiredo Oliveira

para obtener el Grado de

Doctor en Ciencias en Computación

Directores de tesis:

Dr. Francisco Rodríguez Henríquez

Dr. Julio López

Ciudad de México **Febrero 2016**

Acknowledgements

I would like to thank my advisor and co-advisor, professors Francisco Rodríguez Henríquez and Julio López who guided me through the amazing area of cryptography. I also thank professor Alfred Menezes for his warm reception during my stay Waterloo.

A special thanks to my thesis reviewers who provided me valuable comments and suggestions.

I also thank my friends from the cryptography lab for sharing their knowledge in all these years. A special thanks to the department staff who supported me during my Ph. D studies.

I thank the Consejo Nacional de Ciencia y Tecnologia - CONACyT (project number 180421) for their financial support during my stay in Mexico and Canada. Also, a special thanks to ABACUS-Cinvestav for providing us computing resources which were essential for concluding our projects.

A todos mis amigos de México que compartieron momentos inolvidables.

E finalmente, um agradecimento especial à minha família. Sem o seu apoio, minha experiência durante estes quatro anos seria muito mais difícil.

Abstract

Since the beginning of public-key cryptography, small-characteristic finite fields have been proposed as basic mathematical structures for implementing electronic communication protocols and standardized algorithms that achieve different information security objectives. This is because the arithmetic on these fields can be efficiently realized in the binary and trinary number systems, which are fundamental in modern computer architectures. This thesis proposes a concrete analysis of the current security and performance of different primitives based on these fields.

In the first part of this document, we introduce efficient software implementations of the point multiplication algorithm for two families of binary elliptic curves which are provided with efficiently computable endomorphisms. The first class is called Galbraith-Lin-Scott (GLS) curves. There, we present state-of-the-art implementations based on the Gallant-Lambert-Vanstone decomposition method and on the Montgomery ladder approach, in order to achieve a high-speed protected and non-protected code against timing attacks. The second family studied in this thesis is called anomalous binary curves or Koblitz curves. On these elliptic curves, we present, for the first time, a timing-attack protected scalar multiplication based on the regular recoding approach. In addition, we introduce a novel implementation of Koblitz curves defined over the extension field $\mathbb{F}_4$, which resulted in an efficient arithmetic that exploits the internal parallelism contained in the newest desktop processors. All of the previously mentioned implementations are supported by a new projective coordinate system, denoted lambda-coordinates, which provides state-of-the-art formulas for computing the basic point arithmetic operations.

In the second part, we provide a concrete analysis of the impact of the recent approaches for solving the discrete logarithm problem (DLP) in small-characteristic fields of cryptographic interest. After that, we realize practical attacks against fields proposed in the literature to realize pairing-based protocols. Finally, we study the practical implications of the Gaudry-Hess-Smart attack against the binary GLS curves. For that purpose, we analyze and implement techniques to improve the efficiency of the Enge-Gaudry algorithm for solving the DLP over hyperelliptic curves.

Resumen

Desde el inicio de la criptografía de llave pública, los campos finitos de característica chica han sido propuestos como estructuras matemáticas en implementación de protocolos de comunicación electrónica, cuyo objetivo es garantizar distintos atributos de seguridad. Estas estructuras son propuestas porque pueden ser implementadas eficientemente en sistemas numéricos binarios o ternarios, los cuales son intrínsecos de las arquitecturas computacionales modernas. En esta tesis se realiza un análisis de la seguridad y eficiencia de distintas primitivas basadas en estos campos finitos.

En la primera parte de la tesis, presentamos la implementación eficiente en *software* del algoritmo para la multiplicación escalar de puntos en dos familias de curvas elípticas binarias, las cuales cuentan con endomorfismos eficientemente computables. La primera familia es la llamada de Galbraith-Lin-Scott (GLS). En estas curvas presentamos implementaciones construidas con los métodos de Gallant-Lambert-Vanstone y la escalera de Montgomery, con la finalidade de computar una multiplicación escalar eficiente y protegida contra ataques de canal lateral. La segunda familia es la denominada como curvas binarias anómalas o curvas de Koblitz. En esta familia presentamos, de manera inédita, la implementación del algoritmo de multiplicación escalar de puntos protegida contra ataques de canal lateral, basados en tiempo, mediante la técnica de recodificación regular. Además, introducimos una novedosa implementación de las curvas de Koblitz definidas sobre la extensión de campo $\mathbb{F}_4$, lo que resultó en una aritmética eficiente que toma vantaja del paralelismo ofrecido por los procesadores de escritorio más recientes. Todas las implementaciones mencionadas fueron basadas en el nuevo sistema de coordinadas proyectivas lambda que aportan formulas al “estado del arte” para el cómputo de la aritmética de puntos.

En la segunda parte, realizamos un análisis del impacto de los avances recientes en la solución del problema del logaritmo discreto (PLD) en campos finitos de característica chica de interes criptografico. También, realizamos ataques prácticos en campos finitos usados en protocolos basados en emparejamientos. Finalmente, implementamos métodos para mejorar la eficiencia del algoritmo de Enge y Gaudry para resolver el PLD en curvas hiperelipticas.

Resumo

Desde os primórdios da criptografia de chave pública, corpos finitos de característica pequena são propostos como estruturas matemáticas para a implementação de protocolos de comunicação eletrônica que garantem diferentes atributos de segurança da informação. Estas estruturas são propostas pois podem ser instanciadas eficientemente em sistemas numéricos binários ou ternários, que são inerentes nas arquiteturas computacionais contemporâneas. Esta tese realiza uma análise dos recentes avanços em segurança e eficiência em diferentes primitivas baseadas nestes corpos finitos.

Na primeira parte desta tese, descrevemos implementações em *software* de algoritmos para a multiplicação de pontos em duas famílias de curvas elípticas binárias proporcionadas com endomorfismos eficientemente computáveis. A primeira família é chamada curvas Galbraith-Lin-Scott (GLS). Nestas curvas, apresentamos implementações baseadas no método Gallant-Lambert-Vanstone e na escada de Montgomery com a finalidade de gerar uma multiplicação escalar eficiente e protegida contra ataques de canal secundário. A segunda família denominada curvas binárias anômalas ou curvas de Koblitz. Nesta família, realizamos, de maneira inédita, implementações do algoritmo de multiplicação de pontos protegida contra ataques de canal secundário através da técnica da recodificação regular. Além disso, introduzimos implementações das curvas de Koblitz definidas sobre o corpo de extensão $\mathbb{F}_4$, o que resultou em uma aritmética eficiente e que aproveita o paralelismo interno presente nos processadores *desktop*. Todas as implementações mencionadas são construídas sobre um novo sistema de coordenadas projetivas denominadas coordenadas lambda, que fornecem fórmulas de alto nível para o cálculo da aritmética de pontos.

Na segunda parte, proporcionamos uma análise dos avanços recentes na resolução do problema do logaritmo discreto (PLD) em corpos finitos de característica pequena destinados ao uso criptográfico. Em seguida, efetuamos ataques práticos em corpos finitos usados em protocolos baseados em emparelhamentos. Finalmente, estudamos as implicações práticas do ataque Gaudry-Hess-Smart em curvas binárias GLS. Para tal propósito, implementamos técnicas para melhorar a eficiência do algoritmo de Enge e Gaudry para resolver o PLD em curvas hiperelípticas.

Résumé

Depuis les débuts de la cryptographie asymétrique, les corps finis de petite caractéristique sont proposés comme structures mathématiques pour les protocoles de communication électronique, réalisant ainsi plusieurs des objectifs de securité. Ces structures sont proposées parce qu'elles peuvent être efficacement implémentées dans les systèmes numériques binaires ou ternaires, inhérents aux architectures informatiques contemporaines. Cette thèse effectue une analyse des progrès en sécurité et en efficacité d'objets mathématiques cryprographiques basés sur ces corps finis.

Dans la première partie, nous presentons différentes implémentations logicielles efficaces de l'algorithme de multiplication de points sur deux famillies de courbes elliptiques binaires possédant des endomorphismes efficacement calculables. La première catégorie concerne les courbes de Galbraith-Lin-Scott (GLS). Nous presentons des implémentations de multiplication de points basées sur la méthode de décomposition Gallant-Lambert-Vanstone et sur l'échelle de Montgomery pour développer un code rapide, en version protegée et en version non-protegée contre les attaques par canaux auxiliaires. La deuxième catégorie est composée des courbes de Koblitz. Sur ces courbes, nous presentons pour la première fois, un algorithme de multiplication par un scalaire protegé contre les attaques par canaux auxiliaires, basé sur la méthode de la reprogrammation regulière. De plus, nous introduisons une nouvelle implémentation des courbes de Koblitz définies sur le corps fini $\mathbb{F}_4$, qui jouit d'arithmétique efficace exploitant le parallelisme interne des processeurs *desktop*. Toutes nos implémentations sont supportées par un nouveau système de coordonnées projectives, coordonnées lambda, qui fournit une représentation plus adaptée à l'arithmétique de points.

Dans la deuxième partie, nous présentons une analyse de l'impact des nouvelles méthodes pour résoudre le problème du logarithme discret (DLP) dans les corps finis considerés. En suite, nous procédons à des attaques pratiques contre des corps de base de courbes elliptiques *pairing-friendly*. Finalement, nous étudions les implications practiques de l'attaque Gaudry-Hess-Smart contre les courbes GLS. Pour cela, nous mettons en œuvre des techniques pour améliorer l'efficacité de l'algorithme de Enge-Gaudry pour résoudre le DLP dans les courbes hyper-elliptiques.

Contents

List of Figures

List of Tables

List of Algorithms

1 Introduction

Extension fields of small characteristic are quite useful for implementing cryptographic primitives. This is because their elements can be directly represented in the binary or ternary number system, which is inherent to the modern computers based on integrated circuits. As a consequence, the small-characteristic field arithmetic functions are usually more efficient when compared with large prime fields.

For instance, let us consider the basic two-word schoolbook multiplication. We want to multiply two field elements $c = a \times b$, where each of them is stored in two machine registers, namely, (a_0, a_1) and (b_0, b_1). The schoolbook multiplication operation is depicted in Figure 1.1.

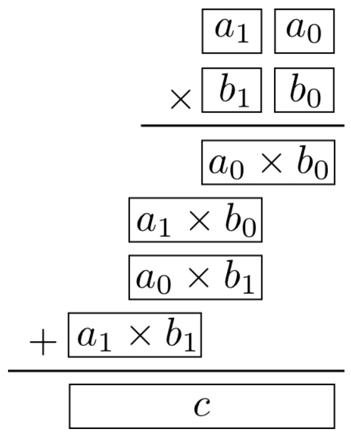

Figure 1.1: The two-word schoolbook multiplication

Given that our architecture is embedded with native multipliers with and without carry, which is the case of modern high-end desktops and smart devices, the four multiplication operations $(a_0 \times b_0)$, $(a_1 \times b_0)$, $(a_0 \times b_1)$ and $(a_1 \times b_1)$ are similar in terms of efficiency for the binary and the prime field cases[1],[2].

However, when we analyze the schoolbook addition phase, the costs differ between the large and small-characteristic fields. If we consider binary fields constructed with a polynomial basis, the addition function is realized easily with the exclusive-or logical operator, since the polynomials that represent the field elements are added coefficient-wise and reduced modulo two. As a result, it is not required to manage carries. On the other hand, in large characteristic fields, we must control the carry values that could appear during the intermediate additions, with makes the implementation more cumbersome and, consequently, less efficient.

Considering the advantage, in terms of efficiency, of the small-characteristic fields, one could ask: why aren't those fields prevalent in real-world cryptographic protocols? The reason is that, in terms of security, the structure inherent to cryptographic primitives constructed over small-characteristic fields allows a wider and more powerful range of attacks. If we consider the binary elliptic curves, different approaches for solving the discrete logarithm problem (ECDLP) were devised in the last decades [58]. In small-characteristic fields, impressive progress in solving the DLP were observed in the last five years, which culminated in a quasi-polynomial algorithm [13].

1.1 Motivation

In short, we have currently the following scenario. On the one hand, there exist different options for selecting efficient and elegant small-characteristic field primitives which are well-suited for implementation on a wide range of software and hardware architectures. On the other hand, strong and effective approaches for solving the mathematical problems beneath those structures were proposed recently and their progress seem to continue. These circunstances have brought a considerable level of suspiciousness in the community on applying cryptographic primitives based on small-characteristic fields to the real-world activities.

In this thesis, we intend to clarify the practical implications of the new advances on the security of small-characteristic field-based primitives and, at the same time, demonstrate that those primitives are highly efficient and should be considered in

[1]In current high-end desktop platforms (e.g. Intel Haswell) the 64-bit carry-less multiplier has a latency of 7 clock cycles [130], while the 64-bit multiplication with carry is available with a latency of 4 clock cycles [52].

[2]For fields of small characteristic greater than two, the multiplication is more costly in software. This is because there are no native instructions which implement the operation in such fields. One solution is to implement the multiplication via the expensive comb methods and/or to use a look-up table approach.

cryptographic libraries, standards and protocols.

1.2 Outline

This document is divided into two parts. In the first part, denoted *high-speed elliptic curve cryptography*, we focus on the constructive aspects of the small-characteristic field cryptography. More precisely, we present software implementations of the scalar multiplication algorithm on elliptic curves defined over binary fields.

In Chapter 2, we introduce a novel system of projective coordinates called *lambda coordinates*. Its formulas for point addition, doubling and doubling-and-addition are presented with their respective proofs. In addition, we compare the cost for computing the point arithmetic operations with state-of-the-art coordinate systems. This work was realized along with Diego F. Aranha, Julio López and Francisco Rodríguez-Henríquez and published in [119, 120].

Chapter 3 describes 128-bit scalar multiplication implementations on a Galbraith-Lin-Scott (GLS) curve defined over the quadratic field $\mathbb{F}_{2^{2\cdot 127}}$. After giving the details of our base and quadratic field arithmetic, we present a protected and non-protected version of the point multiplication algorithm designed with the Gallant-Lambert-Vanstone method. Finally, we propose and implement new procedures in order to compute the Montgomery ladder with the halve-and-add and double-and-add approaches. The work presented in this chapter is based on the papers [119, 120, 118], coauthored with Diego F. Aranha, Julio López and Francisco Rodríguez-Henríquez.

In Chapter 4, we devise methods for implementing timing-resistant point multiplication algorithms on Koblitz curves. At first, we give details of an adaptation of the regular recoding procedure proposed by Joye-Tunstall [91] to scalars represented in the τ-adic form. Next, we propose a new family of Koblitz curves defined over $\mathbb{F}_4$, which resulted in the fastest protected 128-bit secure point multiplication on those curves. The advances presented in this chapter are a joint work with Diego F. Aranha, Julio López and Francisco Rodríguez-Henríquez and were partially published in [118].

In the following paragraphs, we present the outline of the second part of this thesis, entitled *discrete logarithm problem*. In these chapters, we analyzed and implemented algorithms that solve the discrete logarithm problem (DLP) on small-characteristic fields of cryptographic interest and on binary GLS curves.

Chapter 5 describes the recent advances on solving the DLP on small-characteristic fields and presents implementations of those attacks against two pairing-friendly fields, specifically, $\mathbb{F}_{3^{6\cdot 137}}$ and $\mathbb{F}_{3^{6\cdot 163}}$. In addition, we analyze concretely the impact of the new approaches in other fields of cryptographic interest, namely, $\mathbb{F}_{3^{6\cdot 509}}$ and $\mathbb{F}_{3^{6\cdot 1429}}$. This work is related to different papers [2, 1, 4, 3], which were couthored with Gora Adj, Alfred Menezes and Francisco Rodríguez-Henríquez.

In Chapter 6, we present an implementation of the Gaudry-Hess-Smart attack (GHS) against a binary GLS curve defined over the field $\mathbb{F}_{2^{2\cdot 31}}$. Also, we present the practical implications of constructing a dynamic factor base, as proposed in [85], in the relations collection phase of the Enge-Gaudry algorithm for solving the DLP on hyperelliptic curves. This work was published in [36] and was performed with Jesús-Javier Chi.

Finally, in Chapter 7, we conclude the thesis by listing more specifically our main contributions, a collection of open problems and further research themes related to our main subjects of study.

Part I

High-Speed Elliptic Curve Cryptography

2 Lambda Coordinates

From the algorithmic point of view, one of the most effective approaches to accelerate the computation of the scalar multiplication is the improvement of the point arithmetic formulas. The quest for simpler formulas, along with the relatively high cost of the field inversion operation, which is required by the arithmetic of points represented in affine coordinates, motivated the development of distinct projective coordinate systems.

In the case of binary curves, one of the first proposals[1] was the homogeneous projective coordinates system [114, 5], which represents an affine point $P = (x, y)$ as the triplet (X, Y, Z), where $x = \frac{X}{Z}$ and $y = \frac{Y}{Z}$; whereas in the Jacobian coordinate system [37], a projective point $P = (X, Y, Z)$ corresponds to the affine point $(x = \frac{X}{Z^2}, y = \frac{Y}{Z^3})$. In 1998, López-Dahab (LD) coordinates [105] were introduced, representing the affine-coordinate $x = \frac{X}{Z}$ and $y = \frac{Y}{Z^2}$.

Since then, LD coordinates have become the most studied coordinate system for binary elliptic curves, with many authors [94, 101, 8, 100, 21] contributing to improve their performance. In 2007, Kim and Kim [93] presented a 4-dimensional extension of the LD coordinate system that represents P as (X, Y, Z, T^2), with $x = \frac{X}{Z}$, $y = \frac{Y}{T}$ and $T = Z^2$. In a different vein, Bernstein *et al.* introduced in [21] a set of complete formulas[2] for binary Edwards elliptic curves.

Alternatively, we have different affine representations for binary elliptic points, namely, $(x, \frac{y}{x})$ and $(x, x + \frac{y}{x})$, which were introduced in [95, 139]. In [139] the latter representation was designated λ-affine representation of points, and was used for performing the point doubling operation in [105, 106, 139], the point halving in [95, 140, 53, 11], and point compression in [107].

The efficiency of a coordinate system is measured by counting the number of field

[1]The homogeneous projective coordinates were originally proposed to accelerate integer factorization methods based on the elliptic curves [114].

[2]Given a field K, a complete system of addition laws on an elliptic curve E/K has the property that for any two points $P, Q \in E(K)$, there is an addition law in the collection that can be used to add P and Q [32].

operations required to perform the point arithmetic functions, namely, addition and doubling. Usually, only the field multiplication, squaring and inversion operations are considered, since the costs of the other functions, such as the addition, are usually negligible[3]

Also, when presenting coordinate systems costs and formulas, we frequently separate the point addition into two kinds: full or projective, and mixed. Given two points $P = (X_P, Y_P, Z_P)$ and $Q = (X_Q, Y_Q, Z_Q)$, both in projective coordinates, the point full addition is the operation

$$R = (X_R, Y_R, Z_R) = P + Q.$$

The mixed point addition is quite similar: given a point $P = (X_P, Y_P, Z_P)$ in projective coordinates and a point $Q = (x_Q, y_Q)$ in affine coordinates, the mixed addition is the operation $R = (X_R, Y_R, Z_R) = P + Q$. The motives for dividing the point addition into different categories, are twofold: first, the mixed addition is less expensive than the full addition. In the former, one has that, the coordinate Z_Q is equal to one. Consequently, a few field multiplications are saved. Second, different scalar multiplication algorithms require the computation of a distinct amount of mixed and full point multiplication functions. As a result, point multiplication estimations can be made more reliable and concrete if we consider the aforementioned operations separately.

2.1 Coordinate systems

In this section, we describe the main binary projective coordinate system formulas for computing the point doubling and full addition. The mixed addition can be derived from the full addition formula by taking the normalized version of the projective coordinate. Following the scope of this thesis, we only describe the coordinate systems related to Weierstrass binary elliptic curves:

$$E/\mathbb{F}_{2^m} : y^2 + xy = x^3 + ax^2 + b. \tag{2.1}$$

[3]This statement is only true for the binary field arithmetic implemented in high-end desktops. In the near future, it is expected that the difference between the binary field addition and multiplication costs become smaller (see Section 7.3.1).

2.1.1 Affine coordinates

Theorem 1 ([152, Section 2.8]). *Let $P = (x_P, y_P)$ be a point in a non-supersingular elliptic curve. Then the formula for computing $R = 2P = (x_R, y_R)$ is given by:*

$$\begin{aligned}\lambda &= x_P + y_P/x_P\\ x_R &= \lambda^2 + \lambda + a\\ y_R &= \lambda \cdot (x_P + x_R) + x_R + y_P.\end{aligned}$$

Therefore, one inversion, two multiplications and one squaring are required to perform point doubling in affine coordinates.

Theorem 2 ([152, Section 2.8]). *Let $P = (x_P, y_P)$ and $Q = (x_Q, y_Q)$ be points in a non-supersingular elliptic curve, with $P \neq \pm Q$. Then the formula for computing $R = P + Q = (x_R, y_R)$ is given by:*

$$\begin{aligned}\lambda &= (y_P + y_Q)/(x_P + x_Q)\\ x_R &= \lambda^2 + \lambda + x_P + x_Q + a\\ y_R &= \lambda \cdot (x_P + x_R) + x_R + y_P.\end{aligned}$$

Then, we need one inversion, two multiplications and one squaring to perform the point addition in affine coordinates.

2.1.2 Homogeneous projective coordinates

Theorem 3 ([114]). *Let $P = (X_P, Y_P, Z_P)$ be a point in a non-supersingular elliptic curve. Then the formula for computing $R = 2P = (X_R, Y_R, Z_R)$ is given by:*

$$\begin{aligned}A &= X_P \cdot Z_P\\ B &= b \cdot Z_P^4 + X_P^4\\ X_R &= A \cdot B\\ Y_R &= X_P^4 \cdot A + B \cdot (X_P^2 + Y_P \cdot Z_P + A)\\ Z_R &= A^3.\end{aligned}$$

As a result, seven multiplications and five squarings are needed to implement the point doubling in homogeneous coordinates.

Theorem 4 ([114]). *Let $P = (X_P, Y_P, Z_P)$ and $Q = (X_Q, Y_Q, Z_Q)$ be points in a non-supersingular elliptic curve, with $P \neq \pm Q$. Then the formula for computing*

$R = P + Q = (X_R, Y_R, Z_R)$ *is given by:*

$$\begin{aligned}
A &= X_Q \cdot Z_P + X_P \cdot Z_Q \\
B &= Y_Q \cdot Z_P + Y_P \cdot Z_Q \\
C &= A + B \\
D &= A^2 \cdot (A + a \cdot Z_P \cdot Z_Q) + Z_P \cdot Z_Q \cdot B \cdot C \\
X_R &= A \cdot D \\
Y_R &= C \cdot D + A^2 \cdot (B \cdot X_P + A \cdot Y_P) \\
Z_R &= A^3 \cdot Z_P \cdot Z_Q.
\end{aligned}$$

Here, we need sixteen multiplications and one squaring to implement the point full addition in homogeneous coordinates.

2.1.3 Jacobian projective coordinates

The Jacobian coordinates formulas described in this section are based on [19].

Theorem 5 ([37]). *Let $P = (X_P, Y_P, Z_P)$ be a point in a non-supersingular elliptic curve. Then the formula for computing $R = 2P = (X_R, Y_R, Z_R)$ is given by:*

$$\begin{aligned}
A &= X_P^2 \\
B &= A^2 \\
C &= Z_P^2 \\
D &= C^2 \\
X_R &= B + b \cdot D^2 \\
Z_R &= X_P \cdot C \\
Y_R &= B \cdot Z_R + (A + Y_P \cdot Z_P + Z_R) \cdot X_R.
\end{aligned}$$

As a consequence, five multiplications and five squarings are required to implement the point doubling in Jacobian coordinates.

Theorem 6 ([37]). *Let $P = (X_P, Y_P, Z_P)$ and $Q = (X_Q, Y_Q, Z_Q)$ be points in a non-supersingular elliptic curve, with $P \neq \pm Q$. Then the formula for computing*

$R = P + Q = (X_R, Y_R, Z_R)$ is given by:

$$\begin{aligned}
A &= X_P \cdot Z_Q^2 + X_Q \cdot Z_P^2 \\
B &= Y_P \cdot Z_Q^3 + Y_Q \cdot Z_P^3 \\
C &= A \cdot Z_P \\
D &= B \cdot X_Q + C \cdot Y_Q \\
Z_R &= C \cdot Z_Q \\
E &= B + Z_R \\
X_R &= a \cdot Z_R^2 + B \cdot E + A^3 \\
Y_R &= E \cdot X_R + C^2 \cdot D.
\end{aligned}$$

Consequently, fifteen multiplications and five squarings are needed to perform the point full addition in Jacobian coordinates.

2.1.4 López-Dahab projective coordinates

The López-Dahab coordinates formulas described below is based on [19].

Theorem 7 ([105])**.** *Let $P = (X_P, Y_P, Z_P)$ be a point in a non-supersingular elliptic curve. Then the formula for computing $R = 2P = (X_R, Y_R, Z_R)$ is given by:*

$$\begin{aligned}
A &= X_P \cdot Z_P \\
B &= X_P^2 \\
C &= B + Y_P \\
D &= A \cdot C \\
Z_R &= A^2 \\
X_R &= C^2 + D + a \cdot Z_R \\
Y_R &= (Z_R + D) \cdot X_R + B^2 \cdot Z_R.
\end{aligned}$$

In this coordinate system, five multiplications and four squarings are needed to perform the point doubling.

Theorem 8 ([105])**.** *Let $P = (X_P, Y_P, Z_P)$ and $Q = (X_Q, Y_Q, Z_Q)$ be points in a non-supersingular elliptic curve, with $P \neq \pm Q$. Then the formula for computing*

$R = P + Q = (X_R, Y_R, Z_R)$ is given by:

$$\begin{aligned}
A &= X_P \cdot Z_Q^2 \\
B &= X_Q \cdot Z_P^2 \\
C &= A^2 \\
D &= B^2 \\
E &= A + B \\
F &= C + D \\
G &= Y_P \cdot Z_Q^2 \\
H &= Y_Q \cdot Z_P^2 \\
I &= G + H \\
J &= I \cdot E \\
Z_R &= F \cdot Z_P \cdot Z_Q \\
X_R &= A \cdot (H + D) + B \cdot (C + G) \\
Y_R &= (A \cdot J + F \cdot G) \cdot F + (J + Z_R) \cdot X_R.
\end{aligned}$$

As a result, thirteen multiplications and four squarings are required to perform the point full addition in López-Dahab coordinates.

2.1.5 Coordinate systems summary

In Table 2.1, we summarize the costs for performing the point doubling and full addition using the previously presented coordinate systems. Here, $\hat{m}$ represents the general field multiplication. The symbols $\hat{m}_a$ and $\hat{m}_b$ mean, respectively, the field multiplication by the curve parameters a and b. This distinction is made because, in some scenarios, it is possible to choose those curve parameters with a certain degree of freedom. As a result, the developer can select the parameters in a way such that $\hat{m}_a$ and $\hat{m}_b$ are less costly than $\hat{m}$. The squaring operation is symbolized by $\hat{s}$ and the inversion by $\hat{i}$.

We conclude from the above comparison that the López-Dahab coordinate system is the most efficient projective coordinate system for short binary Weierstrass curves. The affine coordinate system would outperform it if one field inversion is less or equal than three multiplications plus three squarings for the point doubling case, and less or equal than eleven multiplications plus three squarings, for the point full addition function. In high-end desktop architectures, those scenarios seem very unlikely in the year term, since the latency and throughput of the carry-less multiplier are being reduced in the newest processors [129].

Table 2.1: Binary coordinate systems comparison: field operations

Coordinate system	**Point doubling**	**Point full addition**
Affine	$1\hat{i} + 2\hat{m} + \hat{s}$	$1\hat{i} + 2\hat{m} + 1\hat{s}$
Homogeneous	$6\hat{m} + 1\hat{m}_b + 5\hat{s}$	$15\hat{m} + 1\hat{m}_a + 1\hat{s}$
Jacobian	$4\hat{m} + 1\hat{m}_b + 5s$	$14\hat{m} + 1\hat{m}_a + 5\hat{s}$
López-Dahab	$4\hat{m} + 1\hat{m}_a + 4\hat{s}$	$13\hat{m} + 4\hat{s}$

In Table 2.2, we present a coordinate system comparative with respect to memory usage. Here, we consider the number of values that must be read or written at least once during the computation of the point operations. During the programming phase, one can optimize the code in order to reduce the amount of memory that need to be simultaneously allocated.

Table 2.2: Binary coordinate systems comparison: memory usage. The variables naming is in accordance with the formulas presented in Section 2.1.

Coordinate system	**Point doubling**	**Point full addition**
Affine	$\lambda + a + (x_R, y_R) + (x_P, y_P)$ Total: 6	$\lambda + a + (x_R, y_R) + (x_P, y_P) + (x_R, y_R)$ Total: 8
Homogeneous	$(A, B) + b + (X_R, Y_R, Z_R) + (X_P, Y_P, Z_P)$ Total: 9	$(A, B, C, D) + a + (X_R, Y_R, Z_R) + (X_P, Y_P, Z_P) + (X_Q, Y_Q, Z_Q)$ Total: 14
Jacobian	$(A, B, C, D) + b + (X_R, Y_R, Z_R) + (X_P, Y_P, Z_P)$ Total: 11	$(A, B, C, D, E) + a + (X_R, Y_R, Z_R) + (X_P, Y_P, Z_P) + (X_Q, Y_Q, Z_Q)$ Total: 15
López-Dahab	$(A, B, C, D) + a + (X_R, Y_R, Z_R) + (X_P, Y_P, Z_P)$ Total: 11	$(A, B, C, D, E, F, G, H, I, J) + (X_R, Y_R, Z_R) + (X_P, Y_P, Z_P) + (X_Q, Y_Q, Z_Q)$ Total: 19

In the next section, we will present formulas for a new coordinate system that produces more efficient formulas than the projective systems discussed hitherto.

2.2 Lambda projective coordinates

As seen in the previous section, in order to have a more efficient elliptic curve arithmetic, it is standard to use a projective version of the Weierstrass elliptic curve equation (2.1), where the points are represented in the so-called projective space. In the following, we describe the λ-projective coordinates, a coordinate system whose associated group law is introduced in this part.

Given a point $P = (x_P, y_P) \in E(\mathbb{F}_{2^m})$ with $x_P \neq 0$, the λ-affine representation of P is defined as (x_P, λ_P), where $\lambda_P = x_P + \frac{y_P}{x_P}$. The λ-projective point $P = (X_P, L_P, Z_P)$ corresponds to the λ-affine point $(\frac{X_P}{Z_P}, \frac{L_P}{Z_P})$. The λ-projective equation form of the Weierstrass equation (2.1) is,

$$(L^2 + LZ + aZ^2)X^2 = X^4 + bZ^4. \tag{2.2}$$

Notice that the condition $x_P = 0$ does not pose a limitation in practice, since the only point P with $x_P = 0$ that satisfies equation (2.1) is $(0, \sqrt{b})$, which is usually confined to a subgroup of no cryptographic interest.

2.2.1 Group law

In this section, the formulas for point doubling and addition in the λ-projective coordinate system are presented. Complementary formulas, when they exist, and complete proofs follow each given formula.

Theorem 9. *Let $P = (X_P, L_P, Z_P)$ be a point in a non-supersingular curve. Then the formula for computing $R = 2P = (X_R, L_R, Z_R)$ using the λ-projective representation is given by*

$$\begin{aligned}
A &= L_P^2 + (L_P \cdot Z_P) + a \cdot Z_P^2 \\
X_R &= A^2 \\
Z_R &= A \cdot Z_P^2 \\
L_R &= (X_P \cdot Z_P)^2 + X_R + A \cdot (L_P \cdot Z_P) + Z_R.
\end{aligned}$$

As a result, five multiplications and four squarings are required to perform the point doubling in λ-projective coordinates.

For situations where the multiplication by the b-coefficient is fast, one can replace a standard multiplication with a multiplication by the constant $(a^2 + b)$. We present below an alternative formula for calculating L_R:

$$L_R = (L_P + X_P)^2 \cdot ((L_P + X_P)^2 + A + Z_P^2) + (a^2 + b) \cdot Z_P^4 + X_R + (a + 1) \cdot Z_R.$$

Proof of Theorem 9. Let $P = (x_P, \lambda_P)$ be a point in an non-supersingular curve. Then a formula for computing $R = 2P = (x_R, \lambda_R)$ is given by

$$\begin{aligned} x_R &= \lambda_P^2 + \lambda_P + a \\ \lambda_R &= \frac{x_P^2}{x_R} + \lambda_P^2 + a + 1. \end{aligned}$$

From [78, Section 3.1.2], we have the formulas: $x_R = \lambda_P^2 + \lambda_P + a$ and $y_R = x_P^2 + \lambda_P x_R + x_R$. Then, a formula for computing λ_R can be obtained as follows:

$$\begin{aligned} \lambda_R &= \frac{y_R + x_R^2}{x_R} = \frac{(x_P^2 + \lambda_P \cdot x_R + x_R) + x_R^2}{x_R} \\ &= \frac{x_P^2}{x_R} + \lambda_P + 1 + x_R = \frac{x_P^2}{x_R} + \lambda_P + 1 + (\lambda_P^2 + \lambda_P + a) \\ &= \frac{x_P^2}{x_R} + \lambda_P^2 + a + 1. \end{aligned}$$

In affine coordinates, the doubling formula requires one division and two squarings. Given the point $P = (X_P, L_P, Z_P)$ in the λ-projective representation, an efficient projective doubling algorithm can be derived by applying the doubling formula to the affine point $(\frac{X_P}{Z_P}, \frac{L_P}{Z_P})$. For x_R we have:

$$\begin{aligned} x_R &= \frac{L_P^2}{Z_P^2} + \frac{L_P}{Z_P} + a = \frac{L_P^2 + L_P \cdot Z_P + a \cdot Z_P^2}{Z_P^2} \\ &= \frac{A}{Z_P^2} = \frac{A^2}{A \cdot Z_P^2}. \end{aligned}$$

For λ_R we have:

$$\begin{aligned} \lambda_R &= \frac{\frac{X_P^2}{Z_P^2}}{\frac{T}{Z_P^2}} + \frac{L_P^2}{Z_P^2} + a + 1 \\ &= \frac{X_P^2 \cdot Z_P^2 + A \cdot (L_P^2 + (a+1) \cdot Z_P^2)}{A \cdot Z_P^2}. \end{aligned}$$

From the λ-projective equation, we have the relation $A \cdot X_P^2 = X_P^4 + b \cdot Z_P^4$. Then the numerator w of λ_R can also be written as follows,

$$\begin{aligned}
w &= X_P^2 \cdot Z_P^2 + A \cdot (L_P^2 + (a+1) \cdot Z_P^2) \\
&= X_P^2 \cdot Z_P^2 + A \cdot L_P^2 + A^2 + A^2 + (a+1) \cdot Z_R \\
&= X_P^2 \cdot Z_P^2 + A \cdot L_P^2 + L_P^4 + L_P^2 \cdot Z_P^2 + a^2 \cdot Z_P^4 + A^2 + (a+1) \cdot Z_R \\
&= X_P^2 \cdot Z_P^2 + A \cdot (L_P^2 + X_P^2) + X_P^4 + b \cdot Z_P^4 + L_P^4 \\
&+ L_P^2 \cdot Z_P^2 + a^2 \cdot Z_P^4 + A^2 + (a+1) \cdot Z_R \\
&= (L_P^2 + X_P^2) \cdot ((L_P^2 + X_P^2) + A + Z_P^2) + A^2 \\
&+ (a^2 + b) \cdot Z_P^4 + (a+1) \cdot Z_R.
\end{aligned}$$

This completes the proof. □

Theorem 10. *Let $P = (X_P, L_P, Z_P)$ and $Q = (X_Q,\ L_Q,\ Z_Q)$ be points in a non-supersingular curve, with $P \neq \pm Q$. Then the addition $R = P + Q = (X_R, L_R, Z_R)$ can be computed by the formulas*

$$\begin{aligned}
A &= L_P \cdot Z_Q + L_Q \cdot Z_P \\
B &= (X_P \cdot Z_Q + X_Q \cdot Z_P)^2 \\
X_R &= A \cdot (X_P \cdot Z_Q) \cdot (X_Q \cdot Z_P) \cdot A \\
L_R &= (A \cdot (X_Q \cdot Z_P) + B)^2 + (A \cdot B \cdot Z_Q) \cdot (L_P + Z_P) \\
Z_R &= (A \cdot B \cdot Z_Q) \cdot Z_P.
\end{aligned}$$

Proof of Theorem 10. Let $P = (x_P, \lambda_P)$ and $Q = (x_Q, \lambda_Q)$ be elliptic curve points. Then a formula for $R = P + Q = (x_R, \lambda_R)$ is given by

$$\begin{aligned}
x_R &= \frac{x_P \cdot x_Q}{(x_P + x_Q)^2}(\lambda_P + \lambda_Q) \\
\lambda_R &= \frac{x_Q \cdot (x_R + x_P)^2}{x_R \cdot x_P} + \lambda_P + 1.
\end{aligned}$$

Since P and Q are elliptic points on a non-supersingular curve, we have the following relation: $y_P^2 + x_P \cdot y_P + x_P^3 + a \cdot x_P^2 = b = y_Q^2 + x_Q \cdot y_Q + x_Q^3 + a \cdot x_Q^2$. The known formula for computing the x-coordinate of R is given by $x_R = s^2 + s + x_P + x_Q + a$, where $s = \frac{y_P + y_Q}{x_P + x_Q}$. Then one can derive the new formula as follows,

$$\begin{aligned}
x_R &= \frac{(y_P + y_Q)^2 + (y_P + y_Q) \cdot (x_P + y_Q)}{(x_P + x_Q)^2} \\
&\quad + \frac{(x_P + x_Q)^3 + a \cdot (x_P + x_Q)^2}{(x_P + x_Q)^2} \\
&= \frac{b + b + x_Q \cdot (x_P^2 + y_P) + x_P \cdot (x_Q^2 + y_Q)}{(x_P + x_Q)^2} \\
&= \frac{x_P \cdot x_Q \cdot (\lambda_P + \lambda_Q)}{(x_P + x_Q)^2}.
\end{aligned}$$

For computing λ_R, we use the observation that the x-coordinate of $R - P$ is x_Q. We also know that for $-P$ we have $\lambda_{-P} = \lambda_P + 1$ and $x_{-P} = x_P$. By applying the formula for the x-coordinate of $R + (-P)$ we have

$$\begin{aligned}
x_Q = x_{R+(-P)} &= \frac{x_R \cdot x_{-P}}{(x_R + x_{-P})^2} \cdot (\lambda_R + \lambda_{-P}) \\
&= \frac{x_R \cdot x_P}{(x_R + x_P)^2} \cdot (\lambda_R + \lambda_P + 1).
\end{aligned}$$

Then $\lambda_R = \frac{x_Q \cdot (x_R + x_P)^2}{x_R \cdot x_P} + \lambda_P + 1$.

To obtain a λ-projective addition formula, we apply the formulas above to the affine points $(\frac{X_P}{Z_P}, \frac{L_P}{Z_P})$ and $(\frac{X_Q}{Z_Q}, \frac{L_Q}{Z_Q})$. Then, the x_R coordinate of $P + Q$ can be computed as:

$$\begin{aligned}
x_R &= \frac{\frac{X_P}{Z_P} \cdot \frac{X_Q}{Z_Q} \cdot (\frac{L_P}{Z_P} + \frac{L_Q}{Z_Q})}{(\frac{X_P}{Z_P} + \frac{X_Q}{Z_Q})^2} \\
&= \frac{X_P \cdot X_Q \cdot (L_P \cdot Z_Q + L_Q \cdot Z_P)}{(X_P \cdot Z_Q + X_Q \cdot Z_P)^2} = X_P \cdot X_Q \cdot \frac{A}{B}.
\end{aligned}$$

For the λ_R coordinate we have:

$$\begin{aligned}
\lambda_R &= \frac{\frac{X_Q}{Z_Q} \cdot (\frac{X_P \cdot X_Q \cdot A}{B} + \frac{X_P}{Z_P})^2}{\frac{X_P \cdot X_Q \cdot A}{B} \cdot \frac{X_P}{Z_P}} + \frac{L_P + Z_P}{Z_P} \\
&= \frac{(A \cdot X_Q \cdot Z_P + B)^2 + (A \cdot B \cdot Z_Q)(L_P + Z_P)}{A \cdot B \cdot Z_P \cdot Z_Q}.
\end{aligned}$$

In order that both x_R and λ_R have the same denominator, the formula for x_R can be written as

$$X_R = \frac{X_P \cdot X_Q \cdot A}{B} = \frac{A \cdot (X_P \cdot Z_Q) \cdot (X_Q \cdot Z_P) \cdot A}{A \cdot B \cdot Z_P \cdot Z_Q}.$$

Therefore, $x_R = \frac{X_R}{Z_R}$ and $\lambda_R = \frac{L_R}{Z_R}$. This completes the proof. □

Furthermore, we derived an efficient formula for computing the operation $R = 2Q+P$, with the points Q and P represented in λ-projective and λ-affine coordinates, respectively.

Theorem 11. *Let $P = (x_P, \lambda_P)$ and $Q = (X_Q, L_Q, Z_Q)$ be points in a non-supersingular curve. Then the operation $R = 2Q+P = (X_R, L_R, Z_R)$ can be computed as follows:*

$$\begin{aligned}
A &= L_Q^2 + L_Q \cdot Z_Q + a \cdot Z_Q^2 \\
B &= X_Q^2 \cdot Z_Q^2 + A \cdot (L_Q^2 + (a + 1 + \lambda_P) \cdot Z_Q^2) \\
C &= (x_P \cdot Z_Q^2 + A)^2 \\
X_R &= (x_P \cdot Z_Q^2) \cdot B^2 \\
Z_R &= (B \cdot C \cdot Z_Q^2) \\
L_R &= A \cdot (B + C)^2 + (\lambda_P + 1) \cdot Z_R.
\end{aligned}$$

Proof of Theorem 11. The λ-projective formula is obtained by adding the λ-affine points $S = 2Q = (x_S, \lambda_S) = (\frac{X_S}{Z_S}, \frac{L_S}{Z_S})$ and $P = (x_P, \lambda_P)$ with the formula of Theorem 2. Then, the x coordinate of $R = S + P$ is given by

$$\begin{aligned}
x_R &= \frac{x_S \cdot x_P}{(x_S + x_P)^2}(\lambda_S + \lambda_P) \\
&= \frac{X_S \cdot x_P(L_S + \lambda_P \cdot Z_S)}{(X_S + x_P \cdot Z_S)^2} \\
&= \frac{x_P \cdot (X_Q^2 \cdot Z_Q^2 + A \cdot (L_Q^2 + (a + 1 + \lambda_P) \cdot Z_Q^2))}{(T + x_P \cdot Z_Q^2)^2} \\
&= x_P \cdot \frac{B}{C}.
\end{aligned}$$

The λ_R coordinate of $S + P$ is computed as

$$\begin{aligned}
\lambda_R &= \frac{\frac{X_S}{Z_S} \cdot (x_P \cdot \frac{B}{C} + x_P)^2}{x_P \cdot \frac{B}{C} \cdot x_P} + \lambda_P + 1 \\
&= \frac{A \cdot (B + C)^2 + (\lambda_P + 1) \cdot (B \cdot C \cdot Z_Q^2)}{B \cdot C \cdot Z_Q^2}.
\end{aligned}$$

The formula for x_R can be written with denominator Z_R as follows,

$$x_R = \frac{x_P \cdot B}{C} = \frac{x_P \cdot Z_Q^2 \cdot B^2}{B \cdot C \cdot Z_Q^2}.$$

Therefore, $x_R = \frac{X_R}{Z_R}$ and $\lambda_R = \frac{L_R}{Z_R}$. This completes the proof. □

2.2.2 Comparison

Table 2.3 summarizes the costs of the basic operations on points represented by the λ-projective coordinate system. For comparison purposes, the costs of those operations with the López-Dahab projective system are also included.

Table 2.3: A cost comparison of the elliptic curve arithmetic using López-Dahab vs. the λ-projective coordinate system

Operations	**Coordinate systems**	
	López-Dahab	Lambda
Full addition	$13\hat{m} + 4\hat{s}$	$11\hat{m} + 2\hat{s}$
Mixed addition	$8\hat{m} + \hat{m}_a + 5\hat{s}$	$8\hat{m} + 2\hat{s}$
Doubling	$3\hat{m} + \hat{m}_a + \hat{m}_b + 5\hat{s}$	$4\hat{m} + \hat{m}_a + 4\hat{s}$ or $3\hat{m} + \hat{m}_a + \hat{m}_b + 4\hat{s}$
Doubling and mixed addition	$11\hat{m} + 2\hat{m}_a + \hat{m}_b + 10\hat{s}$	$10\hat{m} + \hat{m}_a + 6\hat{s}$

The Lambda coordinate system provides a point full addition formula which is two multiplications and two squarings cheaper than the LD formula. Also, it outperforms the LD coordinates in the mixed addition operation by one multiplication by the curve parameter a and three squarings.

Regarding the point doubling, the alternative Lambda formula saves one squaring, when compared with LD coordinates. Moreover, the Lambda coordinates allow to perform the atomic doubling and mixed addition operation (i.e. given the points P and Q, compute $R = 2Q + P$) by one multiplication, one multiplication by the curve parameter a, one multiplication by the curve parameter b and four squarings faster than the LD coordinate system.

Finally, the Lambda point doubling and full addition operations require 8 variables each. This amount is smaller than the Homogeneous coordinates, which is the most efficient binary projective system in terms of memory usage (see Table 2.2).

2.3 Summary

In this chapter, we presented a survey on the projective coordinate systems for binary elliptic curves. For each representation, we gave formulas for computing the point doubling and full addition operations along with their costs in terms of field arithmetic functions.

After that, we introduced a new set of projective coordinates denominated *lambda coordinates*. Here, we presented formulas and their respecive proofs for the point doubling, mixed addition, full addition and doubling-and-mixed-addition operations. Those operations, computed in lambda coordinates, outperforms, in terms of efficiency, the state-of-the-art López-Dahab projective coordinates for binary curves.

3 | Galbraith-Lin-Scott Curves

Given a point $P \in E(\mathbb{F}_{2^m})$ of prime order r, the average cost of computing the scalar multiplication $Q = kP$ by a random n-bit scalar k using the traditional double-and-add method is about $nD + \frac{n}{2}A$, where D and A are the cost of doubling and adding a point, respectively.

In 2001, Gallant, Lambert and Vanstone (GLV) [63] presented a technique that uses efficiently computable endomorphisms, available in certain classes of elliptic curves, which allows significant speedups in the scalar multiplication computation. If the elliptic curve is equipped with a non-trivial efficiently computable endomorphism ψ such that $\psi(P) = \delta P \in \langle P \rangle$, for some $\delta \in [2, r-2]$. Then the point multiplication can be computed through the GLV method as,

$$Q = kP = k_1 P + k_2 \psi(P) = k_1 P + k_2 \cdot \delta P,$$

where the subscalars $|k_1|$, $|k_2| \approx n/2$, can be found by solving a closest vector problem in a lattice [61]. Having split the scalar k into two parts, the computation of $kP = k_1 P + k_2 \psi(P)$ can be performed by applying simultaneous multiple point multiplication techniques [78, Section 3.3.3] that translates into a saving of half of the doublings required by the execution of a single point multiplication kP.

In 2009, Galbraith, Lin and Scott (GLS) [61] constructed efficient endomorphisms for a broader class of elliptic curves defined over $\mathbb{F}_{p^2}$, where p is a prime number, showing that the GLV technique also applies to these curves. Subsequently, Hankerson, Karabina and Menezes investigated in [76] the feasibility of implementing the GLS curves over $\mathbb{F}_{2^{2m}}$.

In this chapter, we present efficient implementations of the 128-bit secure scalar multiplication over binary GLS curves on high-end desktop architectures. Our work provides an efficient quadratic finite field arithmetic and takes advantage of the GLS curve endomorphism to generate fast timing-attack resistant and non-resistant point multiplication algorithms.

3.1 Binary field arithmetic

A binary extension field $\mathbb{F}_{2^m}$ of order $q = 2^m$ can be constructed by taking an m-degree polynomial $f(x) \in \mathbb{F}_2[x]$ irreducible over $\mathbb{F}_2$. The field $\mathbb{F}_{2^m}$ is isomorphic to $\mathbb{F}_2[x]/(f(x))$ and its elements are binary polynomials of degree less than m. Quadratic extensions of a binary extension field can be built using a degree two monic polynomial $g(u) \in \mathbb{F}_2[u]$ that happens to be irreducible over $\mathbb{F}_q$. In this case, the field $\mathbb{F}_{q^2}$ is isomorphic to $\mathbb{F}_q[u]/(g(u))$ and its elements can be represented as $a + bu$, with $a, b \in \mathbb{F}_q$. In this chapter, we developed an efficient field arithmetic library for the field $\mathbb{F}_q$ and its quadratic extension $\mathbb{F}_{q^2}$, with $m = 127$, which were constructed by means of the irreducible trinomials $f(x) = x^{127} + x^{63} + 1$ and $g(u) = u^2 + u + 1$, respectively.

The following discussion assumes $m = 127$, but all techniques can be easily adapted to other field extensions.

3.1.1 Field multiplication over $\mathbb{F}_q$

Given two field elements $a, b \in \mathbb{F}_q$, the field multiplication can be performed by polynomial multiplication followed by modular reduction as, $c = a \cdot b \bmod f(x)$. Since the binary coefficients of the base field elements $\mathbb{F}_q$ can be packed as vectors of two 64-bit words, the standard Karatsuba method allows us to compute the polynomial multiplication step at a cost of three 64-bit products (equivalent to three invocations of the carry-less multiplication instruction [148]), plus some additions. Due to the very special form of $f(x)$, modular reduction is especially elegant as it can be accomplished using essentially additions and shifts.

3.1.2 Field squaring, square root and multi-squaring over $\mathbb{F}_q$

Due to the action of the Frobenius operator, field squaring and square-root are linear operations in any binary field [136]. These two operations can be implemented at a very low cost provided that the base field $\mathbb{F}_q$ is defined by a square-root friendly trinomial or pentanomial[1]. Furthermore, vectorized implementations with simultaneous table look-ups through byte shuffling instructions, as presented in [10], kept square and square-root efficient relative to multiplication even with the acceleration of field multiplication brought by the native carry-less multiplier.

[1]The continuing decrease of the carry-less multiplier costs will probably make this requirement obsolete.

Multi-squaring, or exponentiation to 2^k, with $k > 5$ is performed via look-up of per-field constant tables of field elements, as proposed in [7, 30]. For a fixed k, a table T of $2^4 \cdot \lceil \frac{m}{4} \rceil$ field elements can be precomputed such that

$$T[j, i_0 + 2i_1 + 4i_2 + 8i_3] = (i_0 z^{4j} + i_1 z^{4j+1} + i_2 z^{4j+2} + i_3 z^{4j+3})^{2^k}$$

and $a^{2^k} = \sum_{j=0}^{\lceil \frac{m}{4} \rceil} T[j, \lfloor a/2^{4j} \rfloor \bmod 2^4]$.

3.1.3 Field inversion over $\mathbb{F}_q$

Field inversion in the base field is carried out using the Itoh-Tsujii algorithm [84], by computing $a^{-1} = a^{(2^{m-1}-1)2}$. The exponentiation is computed through the terms $(a^{2^i-1})^{2^k} \cdot a^{2^k-1}$, with $0 \leq i \leq k \leq m-1$. The overall cost of the method is $m-1$ squarings and 9 multiplications given by the length of the following addition chain for $m - 1 = 126$,

$$1 \rightarrow 2 \rightarrow 3 \rightarrow 6 \rightarrow 12 \rightarrow 24 \rightarrow 48 \rightarrow 96 \rightarrow 120 \rightarrow 126.$$

The cost of squarings can be reduced by computing each required 2^k-power as a multi-squaring whenever $k > 5$. This value was determined experimentally.

3.1.4 Modular reduction

Table 3.1 provides the notation of the vector instructions that were used for performing the modular reduction algorithms to be presented in this section. This notation is closely based on [10], but notice that here, we are invoking the three-operand AVX instructions corresponding to 128-bit SSE instructions. Bitwise logical instructions operate across two entire vector registers and produce the result in a third vector register. Bitwise shifts perform parallel shifts in the 64-bit integers packed in a vector register, not propagating bits between contiguous data objects and requiring additional instructions to implement 128-bit shifts. Bytewise shifts are different in both the shift amount, which must be a multiple of 8; and the propagation of shifted out bytes between the two operands. Byte interleaving instructions take bytes alternately from the lower or higher halves of two vector register operands to produce a third output register.

For our irreducible trinomial $f(x) = x^{127} + x^{63} + 1$ choice, we use the procedure shown in Algorithm 1, which requires ten vector instructions to perform a reduction in the base field $\mathbb{F}_q$. This modular reduction algorithm can be improved when performing field squaring. In this case, the 253-bit polynomial a^2, with $a \in \mathbb{F}_q$, is

Table 3.1: Vector instructions used for the binary field arithmetic implementation

Symbol	**Description**	**AVX**
$\oplus$, $\wedge$, $\vee$	Bitwise `XOR`, `AND`, `OR`	`VPXOR`, `VPAND`, `VPOR`
$\ll_{64}, \gg_{64}$	Bitwise shift of packed 64-bit integers	`VPSLLQ, VPSRLQ`
$\rhd$	Bytewise multi-precision shift	`VPALIGNR`
$intlo_{64}$, $intlhi_{64}$	Byte interleaving of packed 64-bit integers	`VPUNPCKLBW`, `VPUNPCKHBW`

represented using two 128-bit registers $r_1||r_0$. By observing that the 63-th bit of the register r_1 is zero, the optimized modular reduction algorithm uses just six vector instructions, as shown in Algorithm 2.

Algorithm 1 Modular reduction by trinomial $f(x) = x^{127} + x^{63} + 1$

Input: 253-bit polynomial d stored into two 128-bit registers $r_1||r_0$.
Output: $\mathbb{F}_q$ element $d \mod f(x)$ stored into a 128-bit register r_0.

```
1: $t_0 \leftarrow (r_1, r_0) \rhd 64$              2: $t_0 \leftarrow t_0 \oplus r_1$
3: $r_1 \leftarrow r_1 \ll_{64} 1$                  4: $r_0 \leftarrow r_0 \oplus r_1$
5: $r_1 \leftarrow inthi_{64}(r_1, t_0)$            6: $r_0 \leftarrow r_0 \oplus r_1$
7: $t_0 \leftarrow t_0 \gg_{64} 63$                 8: $r_0 \leftarrow r_0 \oplus t_0$
9: $r_1 \leftarrow intlo_{64}(t_0, t_0)$            10: $r_0 \leftarrow r_0 \oplus (r_1 \ll_{64} 63)$
11: **return** $r_0$
```

3.1.5 Half-trace over $\mathbb{F}_q$

The trace function on $\mathbb{F}_{2^m}$ is the function $Tr : \mathbb{F}_{2^m} \rightarrow \mathbb{F}_2$ defined as $Tr(c) = \sum_{i=0}^{m-1} c^{2^i}$. The solutions of quadratic equations $x^2 + x = c$ over $\mathbb{F}_q$, with $Tr(c) = 0$, can be found by means of the half-trace function $H : \mathbb{F}_{2^m} \rightarrow \mathbb{F}_{2^m}$, which is defined as $H(c) = \sum_{i=0}^{(m-1)/2} c^{2^{2i}}$. A fast computation of this function can be achieved by exploiting its linear property,

$$H(c) = H(\sum_{i=0}^{m-1} c_i x^i) = \sum_{i=0}^{m-1} c_i H(x^i),$$

Algorithm 2 Modular reduction by $f(x) = x^{127} + x^{63} + 1$ for the squaring operation

```
Input: 253-bit polynomial $a^2$ stored into two 128-bit registers $r_1||r_0$.
Output: $\mathbb{F}_q$ element $a^2 \mod f(x)$ stored into a 128-bit register $r_0$.
  1: $t_0 \leftarrow (r_1, r_0) \rhd 64$              2: $t_0 \leftarrow t_0 \oplus r_1$
  3: $r_1 \leftarrow r_1 \ll_{64} 1$                 4: $r_0 \leftarrow r_0 \oplus r_1$
  5: $t_0 \leftarrow inthi_{64}(r_1, t_0)$           6: $r_0 \leftarrow r_0 \oplus t_0$
  7: return $r_0$
```

and by using an 8-bit index look-up table T of size $2^8 \cdot \lceil \frac{m}{8} \rceil$ field elements such that,

$$H(c) = \sum_{j=0}^{\lceil \frac{m}{8} \rceil} T[j, \lfloor \frac{c}{2^{8j}} \rfloor \bmod 2^8].$$

3.1.6 Field arithmetic over $\mathbb{F}_{q^2}$

Recall that the quadratic extension $\mathbb{F}_{q^2}$ of the base field $\mathbb{F}_q$ is built using the monic trinomial $g(u) = u^2 + u + 1 \in \mathbb{F}_2[u]$ irreducible over $\mathbb{F}_q$. An arbitrary field element $a \in \mathbb{F}_{q^2}$ is represented as $a = a_0 + a_1 u$, with $a_0, a_1 \in \mathbb{F}_q$. Operations in the quadratic extension are performed coefficient-wise. For instance, the multiplication of two elements $a, b \in \mathbb{F}_{q^2}$ is computed as,

$$\begin{aligned} a \cdot b &= (a_0 + a_1 u) \cdot (b_0 + b_1 u) \\ &= (a_0 b_0 + a_1 b_1) + (a_0 b_0 + (a_0 + a_1) \cdot (b_0 + b_1))u, \end{aligned}$$

with $a_0, a_1, b_0, b_1 \in \mathbb{F}_q$.

The square and square-root of a field element a is accomplished using the identities,

$$\begin{aligned} a^2 = (a_0 + a_1 u)^2 &= a_0^2 + a_1^2 + a_1^2 u, \\ \sqrt{a} = \sqrt{a_0 + a_1 u} &= \sqrt{a_0 + a_1} + \sqrt{a_1} u, \end{aligned}$$

respectively. The multiplicative inverse c of a field element a is found by solving the equation $a \cdot c = (a_0 + a_1 u)(c_0 + c_1 u) = 1$, which yields the unique solution, $c_0 = (a_0 + a_1)t^{-1}$ and $c_1 = a_1 t^{-1}$, where $t = a_0 a_1 + {a_0}^2 + {a_1}^2$.

Solving quadratic equations over $\mathbb{F}_{q^2}$ of the form $x^2 + x = c$ with $Tr(c) = 0$, reduces to the solution of two quadratic equations over $\mathbb{F}_q$, as discussed next. For an element $a = a_0 + a_1 u \in \mathbb{F}_{q^2}$, a solution $x = x_0 + x_1 u \in \mathbb{F}_{q^2}$ to the quadratic equation $x^2 + x = a$, can be found by solving the base field quadratic equations,

$$\begin{aligned} x_0^2 + x_1^2 + x_0 &= a_0 \\ x_1^2 + x_1 &= a_1. \end{aligned}$$

Notice that, since $Tr(a_1) = 0$, the solution to the second equation above can be found as $x_1 = H(a_1)$. Then x_0 is determined from $x_0^2 + x_0 = x_1 + a_1 + a_0 + Tr(x_1 + a_1 + a_0)$. The solution is $x = x_0 + (x_1 + Tr(x_1 + a_1 + a_0))\, u$ [76].

The costs of the quadratic extension arithmetic in terms of its base field operations and C language implementation are presented in Table 3.2. Throughout this chapter, we denote $(\tilde{a}, \tilde{m}, \tilde{q}, \tilde{s}, \tilde{i}, \tilde{h}, \tilde{t})$ and $(\hat{a}, \hat{m}, \hat{q}, \hat{s}, \hat{i}, \hat{h}, \hat{t})$ the computational effort associated with the addition, multiplication, square-root, squaring, inversion, half-trace and trace operations over the base field $\mathbb{F}_q$ and its quadratic extension $\mathbb{F}_{q^2}$, respectively.

Table 3.2: Cost of the field $\mathbb{F}_{q^2} \cong \mathbb{F}_q[u]/(u^2 + u + 1)$ arithmetic with respect to the base field $\mathbb{F}_q$ and its C language implementation

Arithmetic over $\mathbb{F}_{q^2}$	**Cost in terms of $\mathbb{F}_q$ arithmetic operations**	**Number of instructions invoked**
Multiplication ($\hat{m}$)	$3\tilde{m} + 4\tilde{a}$	9 PCLMULQDQ + 62 AVX instr.
Square-root ($\hat{q}$)	$2\tilde{q} + \tilde{a}$	37 AVX instr.
Squaring ($\hat{s}$)	$2\tilde{s} + \tilde{a}$	33 AVX instr.
Inversion ($\hat{i}$)	$\tilde{i} + 3\tilde{m} + 3\tilde{a}$	36 PCLMULQDQ + 386 AVX instr. 160 tbl lkup.
Half-trace ($\hat{h}$)	$2\tilde{h} + \tilde{t} + 2\tilde{a}$	19 AVX instr. + 32 tbl lkup.

'PCLMULQDQ', 'AVX instr.' and 'tbl lkup.' stand for carry-less multiplication, 128-bit SSE/AVX vector instruction and table look-up, respectively.

3.2 GLS binary curves

Let $q = 2^m$ and let $E/\mathbb{F}_q : y^2 + xy = x^3 + ax^2 + b$, with $a, b \in \mathbb{F}_q$, be a binary elliptic curve. Also, pick a field element $a' \in \mathbb{F}_{q^2}$ such that $Tr(a') = 1$, where Tr is the trace function from $\mathbb{F}_{q^2}$ to $\mathbb{F}_2$ (see Section 3.1.5). Given $\#E(\mathbb{F}_q) = q+1-t$, it follows that $\#E(\mathbb{F}_{q^2}) = (q+1)^2 - t^2$. Let us define

$$\tilde{E}/\mathbb{F}_{q^2} : y^2 + xy = x^3 + a'x^2 + b, \tag{3.1}$$

with $\#\tilde{E}(\mathbb{F}_{q^2}) = (q-1)^2 + t^2$. It is known that $\tilde{E}$ is the quadratic twist of E, which means that both curves are isomorphic over $\mathbb{F}_{q^4}$ under the endomorphism [76]

$$\phi : E \to \tilde{E},$$

$$(x, y) \mapsto (x, y + sx),$$

with $s \in \mathbb{F}_{q^4} \backslash \mathbb{F}_{q^2}$ satisfying $s^2 + s = a + a'$.

It is also known that the map ϕ is an involution, *i.e.*, $\phi = \phi^{-1}$. Let $\pi : E \to E$ be the Frobenius map defined as $(x, y) \mapsto (x^{2^m}, y^{2^m})$, and let ψ be the composite endomorphism $\psi = \phi\pi\phi^{-1}$ given as,

$$\psi : \tilde{E} \to \tilde{E},$$
$$(x, y) \mapsto (x^{2^m}, y^{2^m} + s^{2^m} x^{2^m} + s x^{2^m}).$$

In this work, the binary elliptic curve $\tilde{E}_{a',b}(\mathbb{F}_{q^2})$ was defined with the parameters $a' = u$ and $b \in \mathbb{F}_q$, where b was carefully chosen to ensure that $\#\tilde{E}_{a',b}(\mathbb{F}_{q^2}) = hr$, with $h = 2$ and where r is a prime of size $2m - 1$ bits. Moreover, $s^{2^m} + s = u$, which implies that the endomorphism ψ acting over the λ-affine point

$$P = (x_0 + x_1 u, \lambda_0 + \lambda_1 u) \in \tilde{E}_{a',b}(\mathbb{F}_{q^2}),$$

can be computed with only three additions in $\mathbb{F}_q$ as

$$\psi(P) \mapsto ((x_0 + x_1) + x_1 u, (\lambda_0 + \lambda_1) + (\lambda_1 + 1)u).$$

3.2.1 Security

Given a point $Q \in \langle P \rangle$, the elliptic curve discrete logarithm problem (ECDLP) consists of finding the unique integer $k \in [0, r-1]$ such that $Q = kP$. To the best of our knowledge, the most powerful attack for solving the ECDLP on binary elliptic curves was presented in [125] (see also [82, 143]), with an associated computational complexity of $O(2^{c \cdot m^{2/3} \log m})$, where $c < 2$, and where m is a prime number. This is worse than generic algorithms with time complexity $O(2^{m/2})$ for all prime field extensions m less than $N = 2000$, a bound that is well above the range used for performing elliptic curve cryptography [125]. On the other hand, since a GLS elliptic curve is defined over a quadratic extension of the field $\mathbb{F}_q$, the generalized Gaudry-Hess-Smart (gGHS) attack [65, 80] to solve the ECDLP on the curve $\tilde{E}$, applies. To prevent this attack, it suffices to verify that the constant b of $\tilde{E}_{a',b}(\mathbb{F}_{q^2})$ is not weak. Nevertheless, the probability that a randomly selected $b \in \mathbb{F}_q^*$ is a weak parameter, is negligibly small [76].

3.3 GLV scalar multiplication

Let $\langle P \rangle$ be an additively written subgroup of prime order r defined over a GLS curve $\tilde{E}(\mathbb{F}_{q^2})$ (see Equation (3.1)). Let k be a positive integer such that $k \in [0, r-1]$. Then, the scalar multiplication operation, denoted by $Q = kP$, corresponds to adding P to itself $k-1$ times.

In this section, the most prominent methods for computing the GLV scalar multiplication on a GLS binary curve $\tilde{E}$ are described. Here, we are specifically interested in the problem of computing the elliptic curve scalar multiplication $Q = kP$, where $q = 2^m$ with prime m, $P \in \tilde{E}(\mathbb{F}_{q^2})$ is a generator of prime order r and $k \in \mathbb{Z}_r$ is a scalar of bitlength $|k| \approx |r| = 2m-1$.

3.3.1 The GLV method and the w-NAF representation

Let ψ be a nontrivial efficiently computable endomorphism of $\tilde{E}$. Also, let us define the integer $\delta \in [2, r-1]$ such that $\psi(Q) = \delta Q$, for all $Q \in \tilde{E}(\mathbb{F}_{q^2})$. Computing kP via the GLV method consists of the following steps.

First, a balanced length-two representation of the scalar $k \equiv k_1 + k_2\delta \mod r$, must be found, where $|k_1|$, $|k_2| \approx |r|/2$. Given k and δ, there exist several methods to find k_1, k_2 [78, 124, 92]. However, considering the efficiency of our implmentation, we decided to follow the suggestion in [61] which selects two integers k_1, k_2 at random, performs the scalar multiplication and then returns $k \equiv k_1 + k_2\delta \mod r$, if required.

Having split the scalar k into two parts, the computation of $kP = k_1P + k_2\psi(P)$ can be performed by simultaneous multiple point multiplication techniques [75], in combination with any of the methods to be described next. A further acceleration can be achieved by representing the scalars k_1, k_2 in the width-w non-adjacent form (w-NAF). In this representation, k_j is written as an n-bit string $k_j = \sum_{i=0}^{n-1} k_{j,i}2^i$, with $k_{j,i} \in \{0, \pm 1, \pm 3, \ldots, \pm 2^{w-1} - 1\}$, for $j \in \{1, 2\}$. A w-NAF string has a length $n \leq |k_j| + 1$, at most one nonzero bit among any w consecutive bits, and its average nonzero-bit density is approximately $1/(w+1)$.

3.3.2 Left-to-right double-and-add

The computation of the scalar multiplication $kP = k_1P + k_2\psi(P)$ via the traditional left-to-right double-and-add method, can be achieved by splitting the scalar k as described above and representing the scalars k_1, k_2 so obtained in their w-NAF form. The precomputation step is accomplished by calculating the 2^{w-2} multiples $P_i = iP$ for odd $i \in \{1, \ldots, 2^{w-1} - 1\}$. For the sake of efficiency, those multiples must be computed in λ-projective form, a task that can be accomplished using the atomic doubling and addition operation described in Section 2.2.1. This is followed by the application of the endomorphism to each point P_i so that the multiples $\psi(P_i)$ are also precomputed and stored. Thereafter, the accumulator Q is initialized at the point at infinity $\mathcal{O}$, and the digits $k_{j,i}$ are scanned from left to right, one at a time. The accumulator is doubled at each iteration of the main loop and in case that $k_{j,i} \neq 0$, the corresponding precomputed multiple is added to the accumulator as, $Q = Q \pm P_{k_{j,i}}$. Algorithm 3 illustrates the method just described.

Algorithm 3 GLV left-to-right double-and-add scalar multiplication

```
Input: $P \in \tilde{E}(\mathbb{F}_{2^{2m}})$, scalars $k_1, k_2$ of bitlength $n \approx |r|/2$, NAF width $w$
Output: $Q = kP$
 1: Compute $w$-NAF($k_i$) for $i \in \{1, 2\}$
 2: for $i \in \{1, \ldots, 2^{w-1} - 1\}$ do $P_i = iP$ and $\tilde{P}_i = \psi(P_i)$ end for
 3: Initialize $Q \leftarrow \mathcal{O}$

 4: for $i = n$ downto $t$ do
 5:     $Q \leftarrow 2Q$
 6:     if $k_{1,i} > 0$ then $Q \leftarrow Q + P_{k_{1,i}}$
 7:     if $k_{1,i} < 0$ then $Q \leftarrow Q - P_{k_{1,i}}$
 8:
 9:     if $k_{2,i} > 0$ then $Q \leftarrow Q + \tilde{P}_{k_{2,i}}$
10:     if $k_{2,i} < 0$ then $Q \leftarrow Q - \tilde{P}_{k_{2,i}}$
11: end for

12: Recode $k_1, k_2 \rightarrow k$.
13: return $Q$
```

3.3.3 Right-to-left halve-and-add

In the halve-and-add method [95, 141], all point doublings are replaced by an operation called point halving. Given a point P, the halving point operation finds R such that $P = 2R$. For the field arithmetic implementation considered in this work, the halving operation is faster than point doubling when applied on binary curves with $Tr(a') = 1$. Halving a point involves computing a field multiplication, a square-root extraction and solving a quadratic equation of the form $x^2 + x = c$ [53], whose solution can be found by calculating the half-trace of the field element c, as discussed in Section 3.1.5.

The halve-and-add method is described as follows. At first, let us compute $k' \equiv 2^{n-1}k \mod r$, with $n = \|r\|_2$. This implies that,

$$k \equiv \sum_{i=0}^{n-1} k'_{n-1-i}/2^i + 2k'_n \mod r,$$

and therefore

$$kP = \sum_{i=0}^{n-1} k'_{n-1-i}(\frac{1}{2^i}P) + 2k'_n P.$$

Then, k' is represented in its w-NAF form, and 2^{w-2} accumulators are initialized as, $Q_i = \mathcal{O}$, for $i \in \{1, 3, \ldots, 2^{w-1} - 1\}$. Thereafter, each one of the n bits of k' are scanned from right to left. Whenever a digit $k'_i \neq 0$, the point $\pm P$ is added to the accumulator $Q_{k'_i}$, followed by $P = \frac{1}{2}P$; otherwise, only the halving of P is performed. In a final post-processing step, all the accumulators are added as $Q = \sum iQ_i$, for $i \in \{1, 3, \ldots, 2^{w-1} - 1\}$. This summation can be efficiently accomplished using Knuth's method [96, Section 4.6.3]. The algorithm outputs the result as $Q = kP$. Algorithm 4, with $t = n$ shows a two-dimensional GLV halve-and-add method.

3.3.4 Lambda-coordinates aftermath

Besides enjoying a slightly cheaper, but at the same time noticeable, computational cost when compared to the LD coordinates, the flexibility of the λ-coordinate system can improve the customary scalar multiplication algorithms in other more subtle ways. For instance, in the case of the double-and-add method, the usage of the atomic doubling and addition operation saves multiplications whenever an addition must be performed in the main loop. The speedup comes from the difference between the cost of the atomic doubling and addition $(10\hat{m} + \hat{m}_a + 6\hat{s})$ shown in Section 2.2.2 versus the expense of performing a doubling and then adding the points in two

Algorithm 4 GLV right-to-left halve-and-add scalar multiplication

Input: $P \in \tilde{E}(\mathbb{F}_{2^{2m}})$, scalars k_1, k_2 of bitlength $n \approx |r|/2$, NAF width w
Output: $Q = kP$

```
 1: Calculate $w$-NAF($k_i$) for $i \in \{1, 2\}$
 2: for $i \in \{1, \dots, 2^{w-1} - 1\}$ do Initialize $Q_i \leftarrow \mathcal{O}$ end for

 3: for $i = n - 1$ downto $0$ do
 4:     if $k_{1,i} > 0$ then $Q_{k_{1,i}} \leftarrow Q_{k_{1,i}} + P$
 5:     if $k_{1,i} < 0$ then $Q_{k_{1,i}} \leftarrow Q_{k_{1,i}} - P$
 6:
 7:     if $k_{2,i} > 0$ then $Q_{k_{2,i}} \leftarrow Q_{k_{2,i}} + \psi(P)$
 8:     if $k_{2,i} < 0$ then $Q_{k_{2,i}} \leftarrow Q_{k_{2,i}} - \psi(P)$
 9:     $P \leftarrow P/2$
10: end for

11: $Q \leftarrow \sum_{i \in \{1, \dots, 2^{w-1}-1\}} iQ_i$
12: Recode $k_1, k_2 \rightarrow k$, if necessary.
13: return $Q$
```

separate steps ($12\hat{m} + \hat{m}_a + 6\hat{s}$). To see the overall impact of this saving in say, the GLV double-and-add method, one has to calculate the probabilities of one, two or no additions in a loop iteration.

Basically, three cases can occur in the 2-GLV double-and-add main loop. The first one, when the digits of both scalars k_1, k_2 equal zero, we just perform a point doubling (D) in the accumulator. The second one, when both scalar digits are different from zero, we have to double the accumulator and sum two points. In this case, we perform one doubling and addition (DA) followed by a mixed addition (A). Finally, it is possible that just one scalar has its digit different from zero. Here, we double the accumulator and add a point, which can be done with only one doubling-and-addition operation.

Then, as the nonzero bit distributions in the scalars represented by the w-NAF are independent, we have for the first case,

$$Pr[k_{1,i} = 0 \wedge k_{2,i} = 0] = \frac{w^2}{(w+1)^2}, \text{ for } i \in \{0, \dots, n-1\}.$$

For the second case,

$$Pr[k_{1,i} \neq 0 \wedge k_{2,i} \neq 0] = \frac{1}{(w+1)^2}, \text{ for } i \in \{0, \dots, n-1\}.$$

And for the third case,

$$Pr[(k_{1,i} \neq 0 \wedge k_{2,i} = 0) \vee (k_{1,i} = 0 \wedge k_{2,i} \neq 0)] = \frac{2w}{(w+1)^2}.$$

Consequently, the operation count can be written as

$$\frac{n}{2}\left(\frac{w^2}{(w+1)^2}D + \frac{1}{(w+1)^2}(DA + A) + \frac{2w}{(w+1)^2}DA\right)$$

$$= \frac{(2w+1)n}{2(w+1)^2}DA + \frac{w^2 n}{2(w+1)^2}D + \frac{n}{2(w+1)^2}A.$$

As mentioned before, it is also possible to apply the doubling and addition operation to speedup the calculation of the multiples of P in the precomputation phase. For that, we modified the original doubling and addition operation to compute simultaneously the points, $R, S = 2Q \pm P$, with an associate cost of just $16\hat{m} + \hat{m}_a + 8\hat{s}$.

More significantly, there is an important multiplication saving in each one of the point additions in the main loop of the halve-and-add method. This is because points in the λ-form $(x, x + \frac{y}{x})$ are already in the required format for the λ-mixed addition operation and, therefore do not need to be reconverted to the regular affine representation as done in [53].

The concrete gains obtained from the λ-projective coordinates can be better appreciated in terms of field operations. Specifically, using the 4-NAF representation of a 254-bit scalar yields the following estimated savings. The double-and-add strategy requires $872\hat{m} + 889\hat{s}$ (considering $\hat{m}_b = \frac{2}{3}\hat{m}$) and $823\hat{m} + 610\hat{s}$ when performed with LD and λ-coordinates, respectively. This amounts for a saving of 31% and 5% in the number of field squarings and multiplications, respectively. The halve-and-add requires $772\hat{m} + 255\hat{s}$ and $721\hat{m} + 101\hat{s}$ when using LD and λ-coordinates, respectively. The savings that the latter coordinate system yields for this case are 60% and 6% fewer field squarings and multiplications, respectively. Notice that these estimations do not consider pre- and post-computation costs.

Table 3.3 presents the estimated costs of the scalar multiplication algorithms in terms of point doublings (D), halvings (H), additions (A), Doubling and additions (DA) and GLS endomorphisms (ψ) when performing the scalar multiplication in the curve $\tilde{E}(\mathbb{F}_{q^2})$.

Table 3.3: Operation counts for selected scalar multiplication methods in a binary GLS curve

	Left-to-right double-and-add	**Right-to-left halve-and-add**
No-GLV (LD)		
Pre/post	$1D + (2^{w-2} - 1)A$	$1D + (2^{w-1} - 2)A$
Sc. mult.	$\frac{n}{w+1}A + nD$	$\frac{n}{w+1}(A + \hat{m}) + nH$
2-GLV (LD)		
Pre/post	$1D + (2^{w-2} - 1)A + 2^{w-2}\psi$	$1D + (2^{w-1} - 2)A$
Sc. mult.	$\frac{n}{w+1}A + \frac{n}{2}D$	$\frac{n}{w+1}(A + \hat{m}) + \frac{n}{2}H + \frac{n}{2(w+1)}\psi$
2-GLV (λ)		
Pre/post	$1D + (2^{w-2} - 1)A + 2^{w-2}\psi$	$1D + (2^{w-1} - 2)A$
Sc. mult.	$\frac{(2w+1)n}{2(w+1)^2}DA + \frac{w^2 n}{2(w+1)^2}D + \frac{n}{2(w+1)^2}A$	$\frac{n}{w+1}A + \frac{n}{2}H + \frac{n}{2(w+1)}\psi$

'Pre/post' and 'Sc. mult.' stands for the pre/post-computation and the scalar multiplication costs, respectively.

3.3.5 Parallel scalar multiplication

In this section, we apply the method given in [7] for computing a scalar multiplication using two CPU cores. The main idea is to compute $k'' \equiv 2^t k \mod r$, for some $0 < t \leq n$. This produces,

$$k \equiv k''_{n-1}2^{n-1-t} + \ldots + k''_t 2^0 + k''_{t-1}/2^{-1} + \ldots + k''_0 2^{-t} \bmod r,$$

which can be rewritten as

$$kP = \sum_{i=t}^{n-1} k''_i (2^{i-t}P) + \sum_{i=0}^{t-1} k''_i \left(\frac{1}{2^{-(t-i)}} P \right).$$

This parallel formulation allows to compute $Q = kP$ using the double-and-add and halve-and-add concurrently, where a portion of k is processed in different cores. The optimal value for the constant t depends on the performance of the scalar multiplication methods and therefore must be found experimentally. The GLV method combined with the parallel technique just explained is presented in Algorithm 5[2].

[2]The pseudo-instruction *Barrier* refers to an OpenMP synchronization clause that forces each thread to wait until all the other threads have completed their assigned tasks.

Algorithm 5 Parallel GLV scalar multiplication

Input: $P \in \tilde{E}(\mathbb{F}_{2^{2m}})$, scalars k_1, k_2 of bitlength $n \approx |r|/2$, NAF width w, constant t
Output: $Q = kP$

```
1:  Calculate $w$-NAF($k_i$) for $i \in \{1, 2\}$

    (left thread)
2:  for $i \in \{1, \ldots, 2^{w-1} - 1\}$ do
3:      Compute $P_i = iP$ and $\tilde{P}_i = \psi(P_i)$
4:  end for

5:  Initialize $Q_0 \leftarrow \mathcal{O}$
6:  for $i = n$ downto $t$ do
7:      $Q_0 \leftarrow 2Q_0$
8:      if $k_{1,i} > 0$ then $Q_0 \leftarrow Q_0 + P_{k_{1,i}}$
9:      if $k_{1,i} < 0$ then $Q_0 \leftarrow Q_0 - P_{k_{1,i}}$
10:     if $k_{2,i} > 0$ then $Q_0 \leftarrow Q_0 + \tilde{P}_{k_{2,i}}$
11:     if $k_{2,i} < 0$ then $Q_0 \leftarrow Q_0 - \tilde{P}_{k_{2,i}}$
12: end for

13: {Barrier}

    (right thread)
2:  for $i \in \{1, \ldots, 2^{w-1} - 1\}$ do
3:      Initialize $Q_i \leftarrow \mathcal{O}$
4:  end for

5:  for $i = t - 1$ downto $0$ do
6:      if $k_{1,i} > 0$ then $Q_{k_{1,i}} \leftarrow Q_{k_{1,i}} + P$
7:      if $k_{1,i} < 0$ then $Q_{k_{1,i}} \leftarrow Q_{k_{1,i}} - P$
8:      if $k_{2,i} > 0$ then $Q_{k_{2,i}} \leftarrow Q_{k_{2,i}} + \psi(P)$
9:      if $k_{2,i} < 0$ then $Q_{k_{2,i}} \leftarrow Q_{k_{2,i}} - \psi(P)$
10:     $P \leftarrow P/2$
11: end for

12: $Q \leftarrow \sum_{i \in \{1,\ldots,2^{w-1}-1\}} iQ_i$
13: {Barrier}

14: Recode $k_1, k_2 \rightarrow k$, if necessary.
15: return $Q \leftarrow Q + Q_0$
```

3.3.6 Protected scalar multiplication

Regular scalar multiplication algorithms attempt to prevent leakage of information about the (possibly secret) scalar, obtained from procedures that have non-constant execution times. There are two main approaches to make a scalar multiplication regular: one is using unified point doubling and addition formulas [21] and the other is recoding the scalar in a predictable pattern [91]. Both halve-and-add and double-and-add methods can be modified in the latter manner, with the additional care that table look-ups to read or write sensitive data need to be completed in constant-time. This can be accomplished by performing linear passes[3] with conditional move instructions over the accumulators or precomputed points, thus thwarting cache-timing attacks.

Implementing timing-attack resistance usually imposes significant performance

[3]The linear pass function is discussed in more details in Chapter 4, Section 4.2.4.

penalties. For example, the density of regular recodings ($\frac{1}{w-1}$) is considerably lower than w-NAF and the access to precomputed data becomes more expensive due to the linear passes. Efficiently computing a point halving in constant time is specially challenging, since the fastest methods for half-trace computation require considerable amounts of memory. This requirement can be relaxed if we assume that the base points are public information and available to the attacker. Notice however that this is a reasonable assumption in most protocols based on elliptic curves, but there are exceptions [35]. In any case, performing linear passes to read and store each one of the 2^{w-2} accumulators used in the halve-and-add procedure discussed in Section 3.3.3, impose a significant impact performance at every point addition.

Because of the above rationale, doubling-based methods seem to be a more promising option for protected implementations. Somewhat surprisingly, the regular recoding method combined with λ-coordinates admits an atomic formula for computing mixed addition plus doubling-and-addition as $2Q + P_i + P_j$ with a cost of $17\tilde{m} + \tilde{m}_a + 8\tilde{s}$, saving one multiplication compared to performing the additions separately. Reading the points P_i, P_j can also be optimized by performing a single linear pass over the precomputed table. These optimizations alone are enough to compensate the performance gap between point doubling and point halving computations to be presented in the next section.

The approach for protected scalar multiplication is shown in Algorithm 6. In this procedure, the scalar k is decomposed into subscalars k_1, k_2 before the main loop. Because the regular recoding requires the input scalar to be odd, we modified slightly the GLV recoding algorithm to produce k_2 always odd, with at most one extra point addition needed to correct the result at the end. This is actually faster than generating random and possibly even k_1, k_2 for reconstructing k, because otherwise two point additions would be needed for correction. These extra point additions must always be performed for satisfying constant-time execution, but conditional move instructions can be used to eliminate incorrect results.

3.3.7 Results and discussion

Our library targets the Intel Sandy Bridge processor family. This multi-core micro-architecture supports carry-less multiplications, the SSE set of instructions [128] that operates on 128-bit registers and the AVX extension [51], which provides SIMD instructions in a three-operand format. However, our code can be easily adapted to any architecture that supports the aforementioned features. The benchmarking was run on an Intel Xeon E31270 3.4GHz and on an Intel Core i5 3570 3.4GHz with the TurboBoost and the HyperThreading technologies disabled. The code was

Algorithm 6 Protected scalar multiplication

```
Input: $P \in E(\mathbb{F}_{2^{2m}})$ of order $r$, $k \in \mathbb{Z}_r$, NAF width $w$
Output: $Q = kP$
 1: Decompose $k$ into $k_1, k_2$, with $k_2$ always odd.
 2: $c \leftarrow 1 - (k_1 \bmod 2)$
 3: $k_1 \leftarrow k_1 + c$
 4: Compute width-$w$ length-$l$ regular recodings of $k_1, k_2$.

 5: for $i \in \{1, \dots, 2^{w-1} - 1\}$ do Compute $P_i = iP$ end for

 6: $Q \leftarrow P_{k_{1,l-1}} + \psi(P_{k_{2,l-1}})$
 7: for $i = l - 2$ downto 0 do
 8:     $Q \leftarrow 2^{w-2}Q$
 9:     Perform a linear pass to recover $P_{k_{1,i}}, P_{k_{2,i}}$.
10:     $Q \leftarrow 2Q + P_{k_{1,i}} + \psi(P_{k_{2,i}})$
11: end for

12: return $Q \leftarrow Q - cP$
```

implemented in the C programming language with intrinsics for vector instructions, compiled with GCC 4.8.1 and executed on 64-bit Linux. Experiments with the ICC 13.0 were also carried out and generated similar results. For that reason, we abstain from presenting timings for that compiler. Also, portions of the code critical for timing-attack resistance (linear passes over precomputed tables, for example), were implemented in Assembly language to prevent undue manipulation by a code-optimizing compiler.

GLS curve parameters

The main parameters of the GLS curve implemented in this chapter are presented below.

Let $q = 2^m$, with $m = 127$. The towering of the fields $\mathbb{F}_q$ and its quadratic extension $\mathbb{F}_{q^2} \cong \mathbb{F}_q[u]/(g(u))$ are constructed by means of the irreducible trinomials $f(x) = x^{127} + x^{63} + 1$ and $g(u) = u^2 + u + 1$, respectively. Let

$$E/\mathbb{F}_q : y^2 + xy = x^3 + ax^2 + b,$$

with $a, b \in \mathbb{F}_q$, be a binary elliptic curve and define the quadratic twist of E as the

Galbraith-Lin-Scott elliptic curve

$$\tilde{E}/\mathbb{F}_{q^2} : y^2 + xy = x^3 + a'x^2 + b,$$

with $a' \in \mathbb{F}_{q^2}$ such that $Tr(a') = 1$. Given $\#E(\mathbb{F}_q) = q + 1 - t$, it follows that $\#\tilde{E}(\mathbb{F}_{q^2}) = (q-1)^2 + t^2$ where t is the trace of Frobenius of the curve E. We selected a curve such that $\#\tilde{E}(\mathbb{F}_{q^2}) = h \cdot r$, where $h = 2$ and r is a 253-bit prime number.

In this work, the binary GLS elliptic curve $\tilde{E}(\mathbb{F}_{q^2})$ was defined with the following parameters

- $a' = u$.
- $b \in \mathbb{F}_q$ is a degree-126 binary polynomial that can be represented in hexadecimal format as, $b =$ 0x59C8202CB9E6E0AE2E6D944FA54DE7E5.
- The 253-bit prime order r of the main subgroup of $\tilde{E}_{a',b}(\mathbb{F}_{q^2})$ is,

$$\begin{aligned} r =& \texttt{0x1FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF} \\ & \texttt{DAC40D1195270779877DABA2A44750A5}. \end{aligned}$$

- The base point $P = (x_p, \lambda_p)$ of order r specified in $\lambda-$affine coordinates is,

$$\begin{aligned} x_p &= \texttt{0x203B6A93395E0432344038B63FBA32DE} \\ &+ \texttt{0x78E51FD0C310696D5396E0681AA10E0D} \cdot u, \\ \lambda_p &= \texttt{0x5BD7653482085F55DEB59C6137074B50} \\ &+ \texttt{0x7F90D98B1589A17F24568FA5A1033946} \cdot u. \end{aligned}$$

Field and elliptic curve arithmetic timings

Table 3.4 shows that the quadratic field arithmetic can handle the base field elements with considerable efficiency. Field inversion, squaring and square-root, as well as the half-trace computational costs are just 1.27, 1.44, 1.87 and 1.43 times higher than their corresponding base field operations, respectively. Field multiplication in the quadratic field can be accomplished at a cost of about 2.23 times base field multiplications, which is significantly better than the theoretical Karatsuba ratio of three.

The lazy reduction technique was employed to optimize the λ-coordinate formulas. Nevertheless, experimental results showed us that this method should be used with caution. Extra savings were obtained by considering the separate case of performing mixed addition where the two points have their Z coordinate equal to one

Table 3.4: Timings (in clock cycles) for the field arithmetic in the Sandy Bridge platform

Field operation	$\mathbb{F}_{2^{127}}$		$\mathbb{F}_{2^{254}}$	
	cycles	op/$\tilde{m}$[a]	cycles	op/m
Multiplication	42	1.00	94	1.00
Mod. reduction[b]	6	0.14	11	0.12
Square root	8	0.19	15	0.16
Squaring	9	0.21	13	0.14
Multi-squaring	55	1.31	n/a[c]	n/a
Inversion	765	18.21	969	10.30
Half-trace	42	1.00	60	0.64
Trace	≈ 0	0	≈ 0	0

[a] Ratio to multiplication.

[b] This cost is included in all operations that require modular reduction.

[c] Multi-squaring is computed only in $\mathbb{F}_{2^{127}}$.

(see Chapter 2). In this case, mixed addition can be performed with just five multiplications and two squarings. This observation helped us to save more than 1000 cycles in the halve-and-add algorithm computation. The reverse recoding calculation, that is, given k_1, k_2 return $k \equiv k_1 + k_2\delta \mod r$ can be omitted if not required. However, in our scalar multiplication timings, this operation was included in all the cases. The timings for the point arithmetic is presented in Table 3.5.

Scalar multiplication timings

From both algorithmic analysis and experimental results, we decided to use $w = 4$ for the w-NAF scalar recoding and $w = 5$ for the regular recoding from [91]. In the case of our parallel implementation (see Algorithm 5), the parameter $t = 72$ was selected, which is consistent with the 1.29 ratio between the double-and-add and halve-and-add computational costs. Notice that in the scalar multiplication procedure, it was assumed that the points are given and returned in λ-affine form. If the input and output points must be represented in conventional affine coordinates, it is necessary to add about 1000 cycles $(2\hat{m} + \hat{i})$ to convert from conventional affine coordinates to the λ ones at the beginning and at the end of the scalar multiplication procedure. Furthermore, we observed an average 2% speedup when executing our code on the newer Ivy Bridge platform. Our scalar multiplication timings, along

Table 3.5: Timings (in clock cycles) for the point arithmetic in the Sandy Bridge platform

Elliptic Curve operation	**GLS** $\tilde{E}/\mathbb{F}_{2^{254}}$	
	cycles	op/M
Doubling	450	4.79
Full addition	1,102	11.72
Mixed addition	812	8.64
Doubling and addition	1,063	11.30
Halving	233	2.48
No-GLV 4-NAF recoding	1,540	16.38
2-GLV 4-NAF recoding	918	9.76
Reverse recoding	396	4.21

with the state-of-the-art implementations, are presented in Table 3.6.

Comparison to related work

Our single-core 4-NAF 2-dimensional GLV implementation achieves 69,500 clock cycles with the halve-and-add method. This result is 20% and 30% faster than the best implementations of point multiplication at the 128-bit security level over prime [50] and binary curves [9], respectively. Furthermore, our two-core parallel implementation using the GLV technique combined with the halve-and-add and double-and-add methods takes 47,900 clock cycles, thus outperforming by 21% the timings reported in [102] for a four-core parallel implementation. Also, the single and multi-core implementations at the 112-bit security level using Koblitz binary curves reported in [148] outperforms our code by just 2% and 3%, respectively. Finally, our single-core protected multiplication is 16% faster than [102], 4% faster than [27] and 16% slower than the current speed record on prime curves [50], but sets a new speed record for binary curves with an improvement of 49% compared to the previous one [148].

A field multiplication comparative

Trying to have a fair comparison that attenuates the diversity of curves, methods and technologies, Table 3.7 compares the estimated number of field multiplications required by implementations that represent the state-of-the-art of unprotected implementations of scalar multiplication computations.

Table 3.6: Timings (in clock cycles) for scalar multiplication with or without timing-attack resistance (TAR) in the Intel Sandy Bridge platform. In our implementation we assume that the input and output points are provided in λ-affine coordinates. Here, (B) and (P) mean that the curve is binary and prime, respectively. Also, the "Sec." column represents the theoretical security in bits

Scalar multiplication	**Curve**	**Sec.**	**Method**	**TAR**	**Cycles**
Taverne *et al.* [148][2]	NIST-K233 (B)	112	No-GLV ♢	no	67,800
Bos *et al.* [27][1]	BK/FKT (P)	128	4-GLV ♣	no	156,000
Aranha *et al.* [9][2]	NIST-K283 (B)	128	2-GLV ♢	no	99,200
Longa and Sica [102][2]	GLV-GLS (P)	128	4-GLV ♣	no	91,000
Faz-H. *et al.* [50][2]	GLV-GLS (P)	128	4-GLV ♣	no	87,000
Taverne *et al.* [148][2]	NIST-K233 (B)	112	No-GLV, (2 cores)	no	46,500
Longa and Sica [102][2]	GLV-GLS (P)	128	4-GLV, (4 cores)	no	61,000
Taverne *et al.* [148][2]	Curve2251 (B)	128	Mont. ladder	yes	225,000
Bernstein [16, 18][2]	Curve25519 (P)	128	Mont. ladder	yes	194,000
Hamburg [74][3]	Montgomery (P)	128	Mont. ladder	yes	153,000
Longa and Sica [102][2]	GLV-GLS (P)	128	4-GLV ♣	yes	137,000
Bos *et al.* [27][1]	Kummer (P)	128	Mont. ladder	yes	117,000
Faz-H. *et al.* [50][2]	GLV-GLS (P)	128	4-GLV ♣	yes	96,000
This work	GLS (B)	127	2-GLV ♣ (LD)	no	116,700
			2-GLV ♣ (λ)	no	92,800
			2-GLV ♡ (LD)	no	82,800
			2-GLV ♡ (λ)	no	**69,500**
			2-GLV (2 cores, λ)	no	**47,900**
			2-GLV ♣ (λ)	yes	**114,800**

[1] Intel Core i7-3520M 2.89GHz (Ivy Bridge)
[2] Intel Core i7-2600 3.4GHz (Sandy Bridge)
[3] Intel Core i7-2720QM 2.2GHz (Sandy Bridge)
♣ Double-and-add ♢ τ-and-add ♡ Halve-and-add

The GLS elliptic curve over a prime field reported in [102] requires 33% more field multiplications than our code. Nevertheless, it benefits from a highly efficient native multiplication with carry instruction (MUL), which allows to generate a fast scalar multiplication. The same observation can be extended to protected implementations when comparing between prime and binary curves.

Table 3.7: A comparison of several elliptic curve libraries by their required number of field multiplications

Implementation	**Field**	**Method**	**Estimated mult.**		**Field mult. cost (cc)**
			pre/post	sc. mult.	
Taverne *et al.* [148]	$\mathbb{F}_{2^{233}}$	No-GLV	92	638	100
Aranha *et al.* [9]	$\mathbb{F}_{2^{283}}$	2-GLV	100	**572**	142
Longa and Sica [102]	$\mathbb{F}_{p^2}$	4-GLV	113	1004	**80**
This work	$\mathbb{F}_{2^{254}}$	2-GLV	**86**	752	94

Faster native multiplication

The Haswell family of processors was launched in 2013, including among other features, the AVX2 set of vector instructions and a faster carry-less multiplier latency and throughput. The latency of this multiplier, compared to previous micro-architectures, was reduced from between 12 and 14 cycles to only 7 cycles, while the reciprocal throughput was reduced from between 7 and 8 cycles to only 2 cycles [52]. In Table 3.8 we report our timings in this platform, specifically in an Intel Core i7 4770K 3.50GHz machine with HyperThreading and TurboBoost disabled.

Table 3.8: Timings and memory requirements for scalar multiplication in the Haswell platform, assuming that the input and output points are provided in λ-affine coordinates

Scalar multiplication	**Method**	**TAR**	**Cycles**	**Memory (bytes)**
This work	2-GLV (double-and-add, λ)	no	**46,700**	$2^{15} + 4 \times 64$
	2-GLV (halve-and-add, λ)	no	**42,100**	$2^{16} + 2^{15} + 4 \times 96$
	2-GLV, parallel (2 cores, λ)	no	**27,300**	$2^{16} + 2^{15} + 4 \times (96 + 64)$
	2-GLV (double-and-add, λ)	yes	**60,000**	8×64

When compared with the Sandy Bridge results (see Table 3.5), the Haswell timings are about 39% faster for the halve-and-add method and about 48% and 50% faster for the protected and unprotected double-and-add implementations, respectively. Note that the faster carry-less multiplication plays the main role in the new results. As a consequence, methods that use more field multiplications, which is the case of the double-and-add, benefit the most. The competitiveness between the double-and-add and halve-and-add methods favors the parallel version, which can almost achieve a two-factor speed-up. When executed in the Haswell platform, the two-core 2-GLV method is 43% faster than the Sandy Bridge timings.

Memory requirements

The library presented in this chapter is intended for its application in high-end platforms where, typically, memory is an abundant resource. Accordingly, several arithmetic operations aggressively use precomputed tables with the aim of achieving a faster computation than what could be obtained by a direct calculation.

In particular, the base field implementation of the half-trace operation, uses a precomputed table of size $2^8 \cdot \lceil \frac{m}{8} \rceil$ field elements. Using $m = 128$, this translates to a 2^{16}-byte table. The faster field inverse implementation invokes four multi-squaring operations, but the constant-time implementations uses slower consecutive squarings. Each one of these multi-squaring operations requires to precompute a table of size $2^4 \cdot \lceil \frac{m}{4} \rceil$ field elements, that translates to a table with a size of 2^{13} bytes. Therefore, the memory cost associated to the faster field inversion computation in our library is of 2^{15} bytes. Finally, the halve-and-add scalar multiplication requires the storage of 4 accumulators in projective coordinates; and the double-and-add scalar multiplication requires the storage of 4 and 8 multiples of the base point for the unprotected and protected versions, respectively. A summary of the memory costs associated to the scalar multiplication algorithms presented in this work are reported in the last column of Table 3.8.

3.4 Montgomery ladder scalar multiplication

In this part, we present new methods aimed to perform fast constant-time variable-base-point multiplication computation for GLS binary elliptic curves. We introduce a novel right-to-left variant of the classical Montgomery-López-Dahab ladder algorithm presented in [104], which efficiently adapted the original ladder idea introduced by Peter Montgomery in his 1987 landmark paper [114]. The new variant presented in this chapter does not require point doublings, but instead, it uses the efficient point halving operation available on binary elliptic curves. In contrast with the algorithm presented in [104] that does not admit the benefit of precomputed tables, our proposed variant *can* take advantage of this technique, a feature that could be valuable for the fixed-base-point multiplication scenario. Moreover, we show that our new right-to-left Montgomery ladder formulation can be nicely combined with the classical ladder to attain a high parallel acceleration factor for a constant-time multi-core implementation of the point multiplication operation.

3.4.1 Montgomery ladder variants

This section presents algorithms for computing the scalar multiplication through the Montgomery ladder method. Again, we let P be a point on a binary elliptic curve of prime order r and k a scalar of bit length n. Our objective is to compute $Q = kP$.

Algorithm 7 Left-to-right Montgomery ladder [114]

```
Input: $P = (x, y)$, $k = (1, k_{n-2}, \dots, k_1, k_0)$
Output: $Q = kP$
1: $R_0 \leftarrow P$; $R_1 \leftarrow 2P$;
2: for $i = n - 2$ downto 0 do
3:     if $k_i = 1$ then
4:         $R_0 \leftarrow R_0 + R_1$; $R_1 \leftarrow 2R_1$
5:     else
6:         $R_1 \leftarrow R_0 + R_1$; $R_0 \leftarrow 2R_0$
7:     end if
8: end for
9: return $Q = R_0$
```

Algorithm 7 describes the classical left-to-right Montgomery ladder approach for point multiplication [114], whose key algorithmic idea is based on the following observation. Given a base point P and two input points R_0 and R_1, such that their difference, $R_0 - R_1 = P$, is known, the x-coordinates of the points, $2R_0$, $2R_1$ and $R_0 + R_1$, are fully determined by the x-coordinates of P, R_0 and R_1.

More than one decade after its original proposal in [114], López and Dahab presented in [104] an optimized version of the Montgomery ladder, which was specifically crafted for the efficient computation of point multiplication on ordinary binary elliptic curves. In this scenario, compact formulas for the point addition and point doubling operations of Algorithm 7 can be derived from the following result.

Lemma 1 ([104])**.** *Let $P = (x, y)$, $R_1 = (x_1, y_1)$, and $R_0 = (x_0, y_0)$ be elliptic curve points, and assume that $R_1 - R_0 = P$, and $x_0 \neq 0$. Then, the x-coordinate of the point $(R_0 + R_1)$, x_3, can be computed in terms of x_0, x_1, and x as follows,*

$$x_3 = \begin{cases} x + \frac{x_0 \cdot x_1}{(x_0 + x_1)^2} & R_0 \neq \pm R_1 \\ x_0^2 + \frac{b}{x_0^2} & R_0 = R_1. \end{cases} \tag{3.2}$$

Moreover, the y-coordinate of R_0 can be expressed in terms of P, and the x-coordinates of R_0, R_1 as,

$$y_0 = x^{-1}(x_0 + x)\left[(x_0 + x)(x_1 + x) + x^2 + y\right] + y. \tag{3.3}$$

Let us denote the projective representation of the points R_0, R_1 and $R_0 + R_1$, without considering their y-coordinates as, $R_0 = (X_0, -, Z_0)$, $R_1 = (X_1, -, Z_1)$ and $R_0 + R_1 = (X_3, -, Z_3)$. Then, for the case $R_0 = R_1$, Lemma 1 implies,

$$\begin{cases} X_3 = X_0^4 + b \cdot Z_0^4 \\ Z_3 = X_0^2 \cdot Z_0^2. \end{cases} \tag{3.4}$$

Furthermore, for the case $R_0 \neq \pm R_1$, one has that,

$$\begin{cases} Z_3 = (X_0 \cdot Z_1 + X_1 \cdot Z_0)^2 \\ X_3 = x \cdot Z_3 + (X_0 \cdot Z_1) \cdot (X_1 \cdot Z_0). \end{cases} \tag{3.5}$$

From Equations (3.4) and (3.5) it follows that the computational cost of each ladder step in Algorithm 7 is 5 multiplications, 1 multiplication by the curve parameter b, 4 or 5 squarings[4] and 3 additions over the binary extension field where the elliptic curve has been defined.

In the rest of this section, we will present a novel right-to-left formulation of the classical Montgomery ladder.

Right-to-left double-and-add Montgomery-LD ladder

Algorithm 8 presents a right-to-left version of the classical Montgomery ladder procedure. At the end of the i-th iteration, the points in the variables R_0, R_1 are $R_0 = 2^{i+1}P$ and $R_1 = \ell P + \frac{P}{2}$, where ℓ is the integer represented by the i rightmost bits of the scalar k. The variable R_2 maintains the relationship, $R_2 = R_0 - R_1$ from the initialization (step 1), until the execution of the last iteration of the main loop (steps 2-9). This comes from the fact that at each iteration, if $k_i = 1$, then the difference $R_0 - R_1$ remains unchanged. If otherwise, $k_i = 0$, then both R_2 and R_0 are updated with their respective original values plus R_0, which ensures that $R_2 = R_0 - R_1$, still holds. Notice however that although the difference $R_2 = R_0 - R_1$ is known, it may vary throughout the iterations.

As stated in Lemma 1, the point additions of steps 4 and 6 in Algorithm 8 can be computed using the x-coordinates of the points R_0, R_1 and R_2, according to the

[4]Either $b = 1$ or $\sqrt{b}$ is precomputed. Formula (3.4) can also be computed as $Z_3 = (X_0 \cdot Z_0)^2$ and $X_3 = (X_0^2 + \sqrt{b} \cdot Z_0^2)^2$

Algorithm 8 Montgomery-LD double-and-add scalar multiplication (right-to-left)

```
Input: $P = (x, y)$, $k = (k_{n-1}, k_{n-2}, \dots, k_1, k_0)$
Output: $Q = kP$
 1: $R_0 \leftarrow P$; $R_1 \leftarrow \frac{P}{2}$; $R_2 \leftarrow \frac{P}{2} = (R_0 - R_1)$;
 2: for $i = 0$ to $n - 1$ do
 3:     if $k_i = 1$ then
 4:         $R_1 \leftarrow R_1 + R_0$;
 5:     else
 6:         $R_2 \leftarrow R_2 + R_0$;
 7:     end if
 8:     $R_0 \leftarrow 2R_0$;
 9: end for
10: return $Q = R_1 - \frac{P}{2}$
```

following analysis. If $k_i = 1$, then the x-coordinate of $R_0 + R_1$ is a function of the x-coordinates of R_0, R_1 and R_2, because $R_2 = R_0 - R_1$. If $k_i = 0$, the x-coordinate of $R_2 + R_0$ is a function of the x-coordinates of the points R_0, R_1 and R_2, because $R_0 - R_2 = R_0 - (R_0 - R_1) = R_1$. Hence, considering the projective representation of the points $R_0 = (X_0, -, Z_0)$, $R_1 = (X_1, -, Z_1)$, $R_2 = (X_2, -, Z_2)$ and $R_0 + R_1 = (X_3, -, Z_3)$, where all the y-coordinates are ignored, and assuming $R_0 \neq \pm R_1$, we have,

$$\begin{cases} T = (X_0 \cdot Z_1 + X_1 \cdot Z_0)^2 \\ Z_3 = Z_2 \cdot T \\ X_3 = X_2 \cdot T + Z_2 \cdot (X_0 \cdot Z_1) \cdot (X_1 \cdot Z_0). \end{cases} \tag{3.6}$$

From Equations (3.4) and (3.6), it follows that the computational cost of each ladder step in Algorithm 8 is 7 multiplications, 1 multiplication by the curve parameter b, 4 or 5 squarings and 3 additions over the binary field where the elliptic curve lies.

Although conceptually simple, the above method has several algorithmic and practical shortcomings. The most important one is the difficulty to recover, at the end of the algorithm, the y-coordinate of R_1, as in none of the available points (R_0, R_1 and R_2) the corresponding y-coordinate is known. This may force the decision to use complete projective formulae for the point addition and doubling operations of steps 4, 6 and 8, which would be costly. Finally, we stress that to guarantee that the case $R_0 = R_2$ will never occur, it is sufficient to initialize R_1 with $\frac{P}{2}$, and perform an affine subtraction at the end of the main loop (step 10).

In the following subsection we present a halve-and-add right-to-left Montgomery ladder algorithm that alleviates the above shortcomings and still achieves a competitive performance.

Right-To-Left halve-and-add Montgomery-LD ladder

Algorithm 9 Montgomery-LD halve-and-add scalar multiplication (right-to-left)

```
Input: $P = (x, y)$, $k' = (k'_{n-1}, k'_{n-2}, \ldots, k'_1, k'_0)$
Output: $Q = kP$
 1: Precomputation: $x(P_i)$, where $P_i = \frac{P}{2^i}$, for $i = 0, \ldots, n$
 2: $R_1 \leftarrow P_n$; $R_2 \leftarrow P_n$;
 3: for $i = 0$ to $n - 1$ do
 4:     $R_0 \leftarrow P_{n-1-i}$;
 5:     if $k'_i = 1$ then
 6:         $R_1 \leftarrow R_0 + R_1$;
 7:     else
 8:         $R_2 \leftarrow R_0 + R_2$;
 9:     end if
10: end for
11: $R_1 \leftarrow R_1 - P_n$
12: return $R_1$
```

Algorithm 9 presents a right-to-left Montgomery ladder procedure similar to Algorithm 8, but in this case, all the point doubling operations are substituted with point halvings. A left-to-right approach using halve-and-add with Montgomery ladder was published in [116], however, this method requires one inversion per iteration, which degrades its efficiency due to the cost of this operation.

As in any halve-and-add procedure, an initial step before performing the actual computation consists of processing the scalar k such that it can be equivalently represented with negative powers of two. To this end, one first computes $k' \equiv 2^{n-1}k \mod r$, with $n = \|r\|_2$. This implies that, $k \equiv \sum_{i=1}^{n} k'_{n-i}/2^{i-1} \mod r$ and therefore, $kP = \sum_{i=1}^{n} k'_{n-i}(\frac{1}{2^{i-1}}P)$. Then, in the first step of Algorithm 9, n halvings of the base point P are computed. We stress that all the precomputed points $P_i = \frac{P}{2^i}$, for $i = 0, \ldots, n$ can be stored in affine coordinates. In fact, just the x-coordinate of each one of the above n points must be stored (with the sole exception of the point P_n, whose y-coordinate is also computed and stored).

As in the preceding algorithm, notice that at the end of the i-th iteration, the points in the variables R_0, R_1 are, $R_0 = \frac{P}{2^{n-i-1}}$, and $R_1 = \ell P + P_n$, where in this case ℓ is the integer represented as $\ell = \sum_{j=0}^{i} \frac{k'_j}{2^{n-j}} \bmod r$. Notice also that the variable R_2 maintains the relationship, $R_2 = R_0 - R_1$, until the execution of the last iteration of the main loop (steps 3-10). This comes from the fact that at each iteration, if $k_i = 1$, then the difference $R_0 - R_1$ remains unchanged. If otherwise, $k_i = 0$, then both R_2 and R_0 are updated with their respective original values plus R_0, which ensures that $R_2 = R_0 - R_1$, still holds.

Since at every iteration, the values of the points R_0, R_1 and $R_0 - R_1$ are all known, the compact point addition formula (3.6) can be used. In practice, this is also possible because the y-coordinate of the output point kP can be readily recovered using Equation 3.3, along with the point $2P$. Moreover, since the points in the precomputed table were generated using affine coordinates, it turns out that the z-coordinate of the point R_0 is always 1 for all the iterations of the main loop. This simplifies (3.6) as,

$$\begin{cases} T = (X_0 \cdot Z_1 + X_1)^2 \\ Z_3 = Z_2 \cdot T \\ X_3 = X_2 \cdot T + Z_2 \cdot (X_0 \cdot Z_1) \cdot (X_1). \end{cases} \tag{3.7}$$

Hence, the computational cost per iteration of Algorithm 9 is 5 multiplications, 1 squaring, 2 additions and one point halving over the binary field where the elliptic curve lies.

GLS Endomorphism The efficient computable endomorphism provided by the GLS curves can be used to implement the 2-GLV method on the Algorithm 9. As a result, only $n/2$ point halving operations must be computed. Besides the speed improvement, the 2-GLV method reduces to a half the number of precomputed points that must be stored.

Multi-core Montgomery ladder

As proposed in [148], by properly recoding the scalar, one can efficiently compute the scalar multiplication in a multi-core environment. Specifically, given a scalar k of size n, we fix a constant t which establishes how many scalar bits will be processed by the double-and-add, and by the halve-and-add procedures. This is accomplished by computing $k' = 2^t k \bmod r$, which yields

$$k = \underbrace{\frac{k'_0}{2^t} + \frac{k'_1}{2^{t-1}} + \cdots + \frac{k'_{t-1}}{2^1}}_{halve-and-add} + \underbrace{\frac{k'_t}{2^0} + 2^1 k'_{t+1} + 2^2 k'_{t+2} + \cdots + 2^{(n-1)-t} k'_{n-1}}_{double-and-add}.$$

In a two-core setting, it is straightforward to combine the left-to-right and right-to-left Montgomery ladder procedures of Algorithms 7 and 9, and distribute them to both cores. In this scenario, the number of necessary pre-computed halved points reduces to $\sim \frac{n}{4}$.

In a four-core platform, we can apply the GLS endomorphism to the left-to-right Montgomery ladder (Algorithm 7). Even though the GLV technique is ineffective for the classical Montgomery algorithm (due to the fact that we cannot share the point doublings between the base point and its endomorphism), the method permits an efficient splitting of the algorithm workload into two cores. In this way, one can use the first two cores for computing t-digits of the GLV subscalars k_1 and k_2 by means of Algorithm 9, while we allocate the other two cores to compute the rest of the scalar's bits using Algorithm 7, as shown in Algorithm 10.

Given t_4 the integer constant that establishes the workload of each algorithm, $P \in E(\mathbb{F}_{q^2})$, and the scalar k represented as $k_1 + k_2 \cdot \delta$ using the GLS-GLV method, cores I and II are both responsible for computing $\lfloor \frac{n}{2} \rfloor - t_4$ bits of the subscalars k_1 and k_2 using the Montgomery-LD double-and-add method. In turn, the cores III and IV, both compute t_4 bits of k_1 and k_2 with the Montgomery-LD halve-and-add algorithm. In the end, on a single core, it is necessary to add all the accumulators Q_i, for $i = 0 \ldots 3$.

Cost comparison of Montgomery ladder variants

Table 3.9 shows the computational costs associated to the Montgomery ladder variants described in this Section. The constants t_2 and t_4 represent the values of the parameter t chosen for the two- and four-core implementations, respectively.[5] All Montgomery ladder algorithms require a basic post-computation cost to retrieve the y-coordinate, which demands ten multiplications, one squaring and one inversion. Due to the application of the GLV technique, the Montgomery-LD-2-GLV halve-and-add version (corresponding to Algorithm 9), requires some few extra operations, namely, the subtraction of a point and the addition of two accumulators, which is performed using the López-Dahab (LD) projective coordinate formulae. In the end,

[5]In our implementations, the values used for the parameters t_2 and t_4 ranged from 53 to 55.

one extra inversion is needed to convert the point representation from LD-projective coordinates to affine coordinates.

Algorithm 10 Parallel Montgomery ladder scalar multiplication (four-core)

```
Input: $P \in E(\mathbb{F}_{q^2})$ of order $r$, scalar $k$ of bit length $n$, integer constant $t_4$
Output: $Q = kP$
 1: $k' \leftarrow 2^{t_4}k \mod r$
 2: Represent $k' = k'_1 + k'_2\lambda$, where $\psi(P) = \lambda P$

{Initialization}
$R_0 \leftarrow \mathcal{O}, R_1 \leftarrow P$
for $i = \lceil \frac{n}{2} \rceil$ downto $t_4$ do
    $b \leftarrow k'_{1,i} \in \{0,1\}$
    $R_{1-b} \leftarrow R_{1-b} + R_b$
    $R_b \leftarrow 2R_b$
end for
$Q_0 \leftarrow R_0$
{Barrier}                      Core I

{Initialization}
$R_0 \leftarrow \mathcal{O}, R_1 \leftarrow P$
for $i = \lceil \frac{n}{2} \rceil$ downto $t_4$ do
    $b \leftarrow k'_{2,i} \in \{0,1\}$
    $R_{1-b} \leftarrow R_{1-b} + R_b$
    $R_b \leftarrow 2R_b$
end for
$Q_1 \leftarrow R_0$
{Barrier}                      Core II

{Precomputation}
for $i = 1$ to $t_4 + 1$ do
    $P_i \leftarrow \frac{P}{2^i}$
end for
{Initialization}
$R_1 \leftarrow P_{t_4+1}$, $R_2 \leftarrow P_{t_4+1}$
for $i = 0$ to $t_4 - 1$ do
    $R_0 \leftarrow P_{t_4-i}$
    $b \leftarrow k'_{1,i} \in \{0,1\}$
    $R_{2-b} \leftarrow R_{2-b} + R_0$
end for
$Q_2 \leftarrow R_1 - P_{t_4+1}$
{Barrier}                      Core III

{Precomputation}
for $i = 1$ to $t_4 + 1$ do
    $P_i \leftarrow \frac{P}{2^i}$
end for
{Initialization}
$R_1 \leftarrow P_{t_4+1}$, $R_2 \leftarrow P_{t_4+1}$
for $i = 0$ to $t_4 - 1$ do
    $R_0 \leftarrow P_{t_4-i}$
    $b \leftarrow k'_{2,i} \in \{0,1\}$
    $R_{2-b} \leftarrow R_{2-b} + R_0$
end for
$Q_3 \leftarrow R_1 - P_{t_4+1}$
{Barrier}                      Core IV

 3: return $Q = Q_0 + Q_2 + \psi(Q_1 + Q_3)$
```

In the case of the parallel versions, the overhead is given by the post-computation done in one single core. The exact costs are mainly determined by the accumulator additions that are performed via full and mixed LD-projective formulas.

Table 3.9: Montgomery-LD algorithms cost comparison. In this table, $\hat{m}, \hat{m}_a, \hat{m}_b, \hat{s}, \hat{i}$ denote the following field operations: multiplication, multiplication by the curve parameters a and b, squaring and inversion. The point halving operation is denoted by H. The scalar bitlength is denoted as n

	Method			**Cost**
1-core	Alg. 7: Montgomery-LD		pre/post	$10\hat{m} + 1\hat{s} + 1\hat{i}$
	(double-and-add, left-to-right)		sc. mult.	$n(5\hat{m} + 1\hat{m}_b + 4\hat{s})$
	Alg. 9: Montgomery-LD-2-GLV		pre/post	$48\hat{m} + 1\hat{m}_a + 13\hat{s} + 3\hat{i}$
	(halve-and-add, right-to-left)		sc. mult.	$(\frac{n}{2} + 1)H + n(5\hat{m} + 1\hat{s})$
2-core	Montgomery-LD-2-GLV	core I	pre/post	$25\hat{m} + 1\hat{m}_a + 5\hat{s} + 2\hat{i}$
	(double-and-add, left-to-right)		sc. mult.	$(n - t_2)(5\hat{m} + 1\hat{m}_b + 4\hat{s})$
	Montgomery-LD-2-GLV	core II	pre/post	$46\hat{m} + 2\hat{m}_a + 12\hat{s} + 2\hat{i}$
	(halve-and-add, right-to-left)		sc. mult.	$(\frac{t_2}{2} + 1)H + t_2(5\hat{m} + 1\hat{s})$
	Overhead			$15\hat{m} + 5\hat{s} + 1\hat{i}$
4-core	Montgomery-LD-2-GLV	cores	pre/post	$10\hat{m} + 1\hat{s} + 1\hat{i}$
	(double-and-add, left-to-right)	I & II	sc. mult.	$(\frac{n}{2} - t_4)(5\hat{m} + 1\hat{m}_b + 4\hat{s})$
	Montgomery-LD-2-GLV	cores	pre/post	$16\hat{m} + 1\hat{m}_a + 4\hat{s} + 1\hat{i}$
	(halve-and-add, right-to-left)	III & IV	sc. mult.	$(\frac{t_4}{2} + 1)H + t_4(5\hat{m} + 1\hat{s})$
	Overhead			$34\hat{m} + 1\hat{m}_a + 12\hat{s} + 1\hat{i}$

3.4.2 Results and discussion

In this part, we discuss several implementation issues. We also present our experimental results and we compare them against state-of-the-art protected point multiplication implementations at the 128-bit security level.

Mechanisms to achieve a constant-time implementation

To protect the previously described algorithms against timing attacks, we observed the following precautions,

Branchless code The main loop, the pre- and post-computation phases are implemented by code that is completely branch-free.

Data veiling To guarantee a constant memory access pattern in the main loop of the Montgomery ladder algorithms, we proposed an efficient data veiling method, which ensures a fixed memory access pattern for all Montgomery-LD ladder algorithms. Given the two Montgomery-LD ladder accumulators A and B, and the scalar $k = (k_{n-1}, k_{n-2}, \dots k_0)$, this method allows us, in the beginning of the i-th main loop iteration, to use the bits k_{i-1} and k_i to decide if A and B will or will not be swapped. As a result, it is not necessary to reapply the procedure at the end of the i-th iteration.

Algorithm 11 saves a considerable portion of the computational effort associated to Algorithm 1 of [25].

Algorithm 11 Data veiling algorithm

```
Input: Scalar digits $k_i$ and $k_{i-1}$, Montgomery-LD ladder accumulators $A$ and $B$
Output: Montgomery-LD ladder accumulators $A$ and $B$
1: $mask \leftarrow 0 - (k_{i-1} \oplus k_i)$
2: $tmp \leftarrow A \oplus B$
3: $tmp \leftarrow tmp \wedge mask$
4: $A \leftarrow A \oplus tmp$
5: $B \leftarrow B \oplus tmp$
6: return $A, B$
```

Field arithmetic Two of the base field arithmetic operations over $\mathbb{F}_q$ were implemented through look-up tables, namely, the half-trace and the multiplicative inverse operations. The half-trace is used to perform the point halving primitive, which is required in the pre-computation phase of the Montgomery-LD halve-and-add algorithm. The multiplicative inverse is one of the operations in the y-coordinate retrieval procedure, at the end of the Montgomery ladder algorithms. Also, whenever post-computational additions are necessary, inverses must be performed to convert a point from LD-projective to affine coordinates.

Although we are aware of the existence of protocols that consider the base point as a secret information [35], in which case one could not consider that our software provides protection against timing attacks, in the vast majority of protocols, the base point is public. Consequently, any attacks aimed at the two field operations mentioned above would be pointless.

GLS curve parameters

For achieving a greater benefit from the multiplication by the curve parameter b in the Montgomery-LD doubling formula $X_3 = {X_0}^4 + b{Z_0}^4 = ({X_0}^2 + \sqrt{b}{Z_0}^2)^2$ we carefully selected a GLS curve with a 64-bit value $\sqrt{b}$. As a result, we saved two carry-less multiplication and a dozen of SSE instructions per field multiplication. Next, we describe the parameters, as polynomials represented in hexadecimal, for our GLS curve $\tilde{E}_{a',b}/\mathbb{F}_{q^2} : y^2 + xy = x^3 + a'x^2 + b$.

- $a' = u$,
- $b =$ `0x54045144410401544101540540515101`,
- $\sqrt{b} =$ `0xE2DA921E91E38DD1`,

The 253-bit prime order r of the main subgroup of $\tilde{E}/\mathbb{F}_{q^2}$ is,

$r =$`0x1FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFA6B89E49D3FECD828CA8D66BF4B88ED5`.

Also, the integer δ such that $\psi(P) = \delta P$ for all $P \in \tilde{E}(\mathbb{F}_{q^2})$ is,

$\delta =$`0x74AEFB81EE8A42E9E9D0085E156A8EFBA3D302F9C74D737FA00360F9395C788`.

The base point $P = (x, y)$ of order r used in this work is,

$x =$`0x4A21A3666CF9CAEBD812FA19DF9A3380` $+$ `0x358D7917D6E9B5A7550B1B083BC299F3` $\cdot u$,
$y =$`0x6690CB7B914B7C4018E7475D9C2B1C13` $+$ `0x2AD4E15A695FD54011BA179D5F4B44FC` $\cdot u$.

Finally, the towering of our field $\mathbb{F}_q \cong \mathbb{F}_2[x]/(f(x))$ and its quadratic extension $\mathbb{F}_{q^2} \cong \mathbb{F}_q[u]/(g(x))$ is constructed by means of the irreducible trinomials $f(x) = x^{127} + x^{63} + 1$ and $g(u) = u^2 + u + 1$.

Scalar multiplication timings

Our implementation was mainly designed for the Intel Haswell processor family, which supports vectorial sets such as SSE and AVX, a carry-less multiplication and some bit manipulation instructions. The programming was done in C with the support of assembly inline code. The compilation was performed via GCC version 4.7.3 with the flags `-m64 -march=core-avx2 -mtune=core-avx2 -O3 -funroll-loops -fomit-frame-pointer`. Finally, the timings were collected on an Intel Core i7-4700MQ, with the Turbo Boost and Hyperthreading features disabled.

Table 3.10 presents the experimental timings obtained for the most prominent building blocks required for computing the point multiplication operation on the GLS binary elliptic curves.

We present, in Table 3.11, a comparison of our timings against a selection of state-of-the-art implementations of the point multiplication operation on binary and prime elliptic curves. Due to the Montgomery-LD point doubling efficiency, which costs

Table 3.10: Timings (in clock cycles) for the elliptic curve operations in the Intel Haswell platform

Elliptic curve operation	GLS $\tilde{E}/\mathbb{F}_{2^{254}}$	
	cycles	op/M^1
Halving	184	4.181
Montgomery-LD D&A (left-to-right) Addition (Eq. (3.5))	161	3.659
Montgomery-LD H&A (right-to-left) Addition (Eq. (3.7))	199	4.522
Montgomery-LD Doubling^2 (Eq. (3.4))	95	2.159

^{1} Ratio to multiplication.

^{2} The flexibility for finding a curve parameter b, provided by the GLS curves, allow us to have a small $\sqrt{b}$. As a consequence, we used the Eq. (3.4) alternative formula.

49% less than a point halving, the GLS-Montgomery-LD-double-and-add achieved the fastest timing in the one-core setting, with 70,800 clock cycles. This is 13% faster than the performance obtained by the GLS-Montgomery-LD-halve-and-add algorithm. In the known-base point setting, we can ignore the GLS-Montgomery-LD-halve-and-add pre-computation expenses associated with its table of halved points. In that case, we can compute the scalar multiplication in an estimated time of 44,600 clock cycles using a table of just 4128 bytes.

Furthermore, the GLS-Montgomery-LD-halve-and-add is crucial for implementing the multi-core versions of the Montgomery ladder. When compared with our one-core double-and-add implementation, Table 3.11 reports a speedup of 1.36 and 2.03 in our two- and four-core Montgomery ladder versions, respectively. Here, besides the overhead costs commented in Section 3.4.1, we can clearly perceive the usual multicore management penalty. Finally, we observe that our GLS-Montgomery-LD-double-and-add surpasses by 48%, 40% and 2% the Montgomery ladder implementations of [25] (Random), [25] (Koblitz) and [17], respectively.

3.5 Summary

In the first part of this chapter, we presented a fast software implementation, aimed at high-end desktop architectures, of the quadratic field $\mathbb{F}_{2^{2 \cdot 127}}$ arithmetic. This arithmetic was used as a base for our speed-record 128-bit scalar multiplication on GLS curves. In order to achieve that, we took advantage of the efficiently computable endomorphism present in those curves to design a point multiplication with the 2-GLV

Table 3.11: Timings (in clock cycles) for 128-bit level scalar multiplication with timing-attack resistance in the Intel Ivy Bridge (I) and Haswell (H) architectures

	Method	**Cycles**	**Arch**
State-of-the-art implementations	Montgomery-DJB-chain (prime) [42]	148,000	I
	Random-Montgomery-LD ladder (binary) [25]	135,000	H
	Genus-2-Kummer (prime) [28]	122,000	I
	Koblitz-Montgomery-LD ladder (binary) [25]	118,000	H
	Twisted-Edwards-4-GLV (prime) [50]	92,000	I
	Genus-2-Kummer Montgomery ladder (prime) [17]	72,200	H
	GLS-2-GLV double-and-add (binary, λ) [120]	60,000	H
Our Work	GLS-Montgomery-LD-2-GLV halve-and-add (**Alg. 7**)	80,800	H
	GLS-Montgomery-LD double-and-add (**Alg. 9**)	70,800	H
	2-core GLS-Montgomery-LD-2-GLV halve-and-add/double-and-add	52,000	H
	4-core GLS-Montgomery-LD-2-GLV halve-and-add/double-and-add (**Alg. 10**)	34,800	H

decomposition method along with the halve-and-add approach, for the not-timing-resistant version, and the double-and-add algorithm for the timing-resistant implementation. In addition, we presented a 2-core version of our not-timing-resistant implementation that took less than 30,000 clock cycles.

In the second part, we applied the Montgomery-LD ladder approach to the GLS curves to achieve an efficent timing-resistant implementation. Also, we designed, for the first time, a Montgomery-LD halve-and-add point multiplication algorithm that makes extensive use of pre-computation. Finally, we merged the Montgomery-LD double-and-add and halve-and-add approaches to generate a 4-core parallel LD-Montgomery ladder algorithm, which took about 35,000 cycles in a Haswell machine.

4 | Koblitz Curves

The anomalous binary curves, generally referred to as Koblitz curves, are binary elliptic curves which satisfies the following Weierstrass equation,

$$E_a : y^2 + xy = x^3 + ax^2 + 1, \tag{4.1}$$

with $a \in \{0, 1\}$. Since their introduction in 1991 by Neal Koblitz [99], these curves were extensively studied for their additional structure that allows, in principle, a performance speedup in the point multiplication computation.

Let $q = 2^m$, with prime m. The set of affine points $P = (x, y) \in \mathbb{F}_q \times \mathbb{F}_q$ that satisfy Equation 4.1 together with a point at infinity represented as $\mathcal{O}$, forms an abelian group denoted by $E_a(\mathbb{F}_{2^m})$ of order $\#E_a(\mathbb{F}_{2^m}) = 2 \cdot (2 - a) \cdot r$. Its group law is defined by the point addition operation.

In λ-affine coordinates (see Chapter 2), where the points are represented as $P = (x, \lambda = x + \frac{y}{x})$, $x \neq 0$, the λ-affine form of the curve equation becomes,

$$E_a : (\lambda^2 + \lambda + a)x^2 = x^4 + 1. \tag{4.2}$$

The Frobenius map $\tau : E_a(\mathbb{F}_q) \to E_a(\mathbb{F}_q)$ defined by $\tau(\mathcal{O}) = \mathcal{O}$, $\tau(x, y) = (x^2, y^2)$, is a curve automorphism satisfying $(\tau^2 + 2)P = \mu\tau(P)$ for $\mu = (-1)^{1-a}$ and all $P \in E_a(\mathbb{F}_q)$. By solving the equation $\tau^2 + 2 = \mu\tau$, the Frobenius map can be seen as the complex number $\tau = (\mu \pm \sqrt{-7})/2$. Notice that in λ-coordinates the Frobenius map action remains the same. Let Λ be the function that tranforms the point coordinates from affine to λ-affine as $\Lambda(x, y) = (x, x + \frac{x}{y})$, then $\tau(\Lambda(x, y)) = (x^2, x^2 + \frac{y^2}{x^2})$, which corresponds to the λ-representation of $\tau(x, y)$.

Let $\mathbb{Z}[\tau]$ be the ring extension of $\mathbb{Z}$. Since the Frobenius map is computationally cheap, as long as it is possible to convert an integer scalar k to its τ-representation $k = \sum_{i=0}^{l-1} u_i\tau^i$, with $u_i \in \mathbb{Z}$, its action can be exploited in a point multiplication computation by adding multiples $u_i\tau^i(P)$. Solinas [145] proposed exactly that, namely, a τ-adic scalar recoding analogous to the signed digit scalar non-adjacent form (NAF) representation.

From the security point of view, it has been argued that the availability of additional structure in the form of endomorphisms can be a potential threat to the hardness of elliptic curve discrete logarithms [20], but limitations observed in approaches based on isogeny walks is evidence to the contrary [97]. Furthermore, the generation of Koblitz curves satisfy by definition the rigidity property in the most strict sense[1].

Constant-time compact implementations for Koblitz curves are also easily obtained by specializing the Montgomery-López-Dahab ladder algorithm [104] for the curve parameter $b = 1$, although we show below that this is not the most efficient constant-time implementation strategy possible. Another practical advantage is the adoption of Koblitz curves by several standards bodies [131], which guarantee interoperability and availability of implementations in many hardware and software platforms.

The present chapter is divided into two parts. First, a regular τ-adic recoding method, which is a necessary step in order to implement a timing-resistant scalar multiplication on Koblitz curves, is proposed. Second, the new recoding method is used to introduce a 128-bit secure protected point multiplication on a new family of Koblitz curves defined over $\mathbb{F}_4$.

4.1 A novel regular τ-adic approach

Let r be a prime order of a subgroup of $E_a(\mathbb{F}_q)$ and k a scalar in $\mathbb{Z}_r$. The recoding approach proposed by Solinas [145] finds an element $\rho \in \mathbb{Z}[\tau]$, of minimal norm as possible, such that $\rho \equiv k \pmod{\frac{\tau^m-1}{\tau-1}}$. A τ-adic expansion with average non-zero density $\frac{1}{3}$ can be obtained by repeatedly dividing ρ by τ and assigning the remainders to the digits u_i to obtain $k = \sum_{i=0}^{i=l-1} u_i\tau^i$. An alternative approach that does not involve multi-precision divisions, is to compute the partial reduction[2] of the element k as $\rho = k \text{ partmod } \left(\frac{\tau^m-1}{\tau-1}\right)$.

A width-w τ-NAF expansion with non-zero density $\frac{1}{w+1}$, where at most one of any w consecutive coefficients is non-zero, can also be obtained by repeatedly dividing ρ' by τ^w and assigning the remainders to the digit set $\{0, \pm\alpha_1, \pm\alpha_3, \dots, \pm\alpha_{2^{w-1}-1}\}$, for $\alpha_i = i \bmod \tau^w$. Under reasonable assumptions, this window-based recoding has length $l \leq m + 1$ [145].

[1]The only degree of freedom in the curve generation process consists in choosing a suitable prime degree extension m that produces a curve with almost-prime order.

[2]The operation is denoted as *partmod* in [145].

In the following sections, a regular recoding version of the width-w τ-NAF expansion is derived. The security advantages of such recoding are the predictable length and locations of non-zero digits in the expansion. This eliminates any side-channel information that an attacker could possibly collect regarding the operation executed at any iteration of the scalar multiplication algorithm (Frobenius map or point addition). As long as access to the pre-computed points is kept constant, the resulting algorithm should be resistant against any timing-based side-channel attacks.

4.1.1 Recoding in τ-adic form

Let us first consider the integer recoding proposed by Joye and Tunstall [91]. They observed that any odd integer $i \in [0, 2^w)$ can be written as $i = 2^{w-1} + (-(2^{w-1} - i))$. Repeatedly dividing an odd n-bit integer $k - ((k \bmod 2^w) - 2^{w-1})$ by 2^{w-1} maintains the parity and assigns the remainders to the digit set $\{\pm 1, \ldots, \pm(2^{w-1} - 1)\}$, producing an expansion of length $\lceil 1 + \frac{n}{w-1} \rceil$ with non-zero density $\frac{1}{w-1}$. Our solution for the problem of finding a regular τ-adic expansion employs the same intuition as explained next.

Let $\phi_w : \mathbb{Z}[\tau] \to \mathbb{Z}_{2^w}$ be a surjective ring homomorphism induced by $\tau \mapsto t_w$, for $t_w^2 + 2 \equiv \mu t_w \pmod{2^w}$, with kernel $\{\alpha \in \mathbb{Z}[\tau] : \tau^w \text{ divides } \alpha\}$. An element $i = i_0 + i_1\tau$ from $\mathbb{Z}[\tau]$ with odd integers $i_0, i_1 \in [0, 2^w)$ satisfies the analogous property $\phi_w(i) = 2^{w-1} + (-(2^{w-1} - \phi_w(i)))$. Repeated division of

$$(r_0 + r_1\tau) - (((r_0 + r_1\tau) \bmod \tau^w) - \tau^{w-1})$$

by τ^{w-1}, correspondingly of $\phi_w(\rho') = (r_0 + r_1 t_w) - ((r_0 + r_1 t_w \bmod 2^w) - 2^{w-1})$ by 2^{w-1}, yields remainders that belong to the set $\{0, \pm\alpha_1, \pm\alpha_3, \ldots, \pm\alpha_{2^{w-1}-1}\}$. The resulting expansion has always length $\lceil 1 + \frac{m+2}{w-1} \rceil$ and non-zero density $\frac{1}{w-1}$.

Algorithm 12 presents the recoding process for any $w \geq 2$. The resulting recoding can also be seen as an adaption of the SPA-resistant recoding of [117], mapping to the digit set $\{0, \pm\alpha_1, \pm\alpha_3, \ldots, \pm\alpha_{2^{w-1}-1}\}$ instead of integers. While the non-zero densities are very similar, our scheme provides a performance benefit in the precomputation step, since the Frobenius map is usually faster than point doubling and preserves affine coordinates, which consequently, allows faster point additions.

4.1.2 Left-to-right regular scalar multiplication

Algorithm 13 presents a complete description of a regular scalar multiplication approach that uses as a building block the regular width-w τ-recoding recoding procedure just described.

Algorithm 12 Regular width-w τ-adic expansion for an m-bit scalar

Input: w, t_w, $\alpha_u = \beta_u + \gamma_u\tau$ for $u = \{\pm 1, \pm 3, \pm 5, \ldots, \pm 2^{w-1} - 1\}$, $\rho = r_0 + r_1\tau \in \mathbb{Z}[\tau]$ with odd r_0, r_1

Output: $\rho = \displaystyle\sum_{i=0}^{\lceil \frac{m+2}{w-1} \rceil} u_i \tau^{i(w-1)}$

```
1:  for $i \leftarrow 0$ to $\lceil \frac{m+2}{w-1} \rceil$ - 1 do
2:      if $w = 2$ then
3:          $u_i \leftarrow ((r_0 - 2r_1) \mod 4) - 2$
4:          $r_0 \leftarrow r_0 - u_i$
5:      else
6:          $u \leftarrow (r_0 + r_1 t_w \mod 2^w) - 2^{w-1}$
7:          if $u > 0$ then $s \leftarrow 1$ else $s \leftarrow -1$
8:          $r_0 \leftarrow r_0 - s\beta_u, r_1 \leftarrow r_1 - s\gamma_u, u_i \leftarrow s\alpha_u$
9:      end if
10:     for $j \leftarrow 0$ to $(w - 2)$ do
11:         $t \leftarrow r_0, r_0 \leftarrow r_1 + \mu r_0/2, r_1 \leftarrow -t/2$
12:     end for
13: end for
14: if $r_0 \neq 0$ and $r_1 \neq 1$ then
15:     $u_i \leftarrow r_0 + r_1\tau$
16: else
17:     if $r_1 \neq 0$ then
18:         $u_i \leftarrow r_1$
19:     else
20:         $u_i \leftarrow r_0$
21:     end if
22: end if
```

Algorithm 13 Timing attack resistant scalar multiplication

Input: $P = (x, \lambda)$, $k \in \mathbb{Z}_r$, width w

Output: $Q = kP$

```
1:  Compute $\rho = r_0 + r_1\tau = k$ partmod $\left(\frac{\tau^m - 1}{\tau - 1}\right)$
2:  if $2|r_0$ then $r_0' = r_0 + 1$ end if
3:  if $2|r_1$ then $r_1' = r_1 + 1$ end if
4:  Compute the width-$w$ length-$l$ regular $\tau$-adic representation of $\rho' = r_0' + r_1'\tau$ as
    $\sum_{i=0}^{\lceil 1 + \frac{m+2}{w-1} \rceil} u_i \tau^{i(w-1)}$ (Alg. 12)
5:  for $i \in \{1, \ldots, 2^{w-1} - 1\}$ do Compute $P_u = \alpha_u P$ end for
6:  $Q \leftarrow \mathcal{O}$
7:  for $i = l - 1$ downto 0 do
8:      $Q \leftarrow \tau^{w-1}(Q)$
9:      Perform a linear pass to recover $P_{u_i}$
10:     $Q \leftarrow Q + P_{u_i}$
11: end for
12: return $Q = Q - (r_0' - r_0)P - (r_1' - r_1)\tau(P)$.
```

4.1.3 Results and discussion

In this section, we present an implementation of the novel regular recoding technique on a NIST standardized Koblitz curve defined over $\mathbb{F}_{2^{283}}$ (NIST K-283). In addition, our work is compared with the state-of-the-art 128-bit secure scalar multiplications.

Mechanisms to achieve a constant-time Koblitz implementation

Implementing Algorithm 13 in constant time requires some attention, since all of its building blocks must be implemented in constant time.

Finite field arithmetic. Modern implementations of finite field arithmetic make extensive use of vector registers, which removes timing variances due to the cache hierarchy. For our illustrative implementation of the curve NIST K-283, we closely follow the arithmetic described in Bluhm-Gueron [25], adopting the incomplete reduction improvement proposed by Nègre-Robert [116].

Integer recoding. All the branches in Algorithm 12 must be eliminated by conditional execution statements in order to prevent leakage[3] of the scalar k. Moreover, to remove the remaining sign-related branches, multiple precision integer arithmetic must be implemented in two's complement. If two constants, say β_u, γ_u, are stored in a precomputed table, then they need to be recovered by a linear pass across the table in constant time. Finally, it is essential that the partial reduction step also be implemented in constant time by removing all of its branches. Notice that the requirement for r_0, r_1 to be odd is not a problem, since partial reduction can be modified to always result in odd integers, with a possible correction at the end of the scalar multiplication by performing a protected conditional subtraction of points (see Algorithm 13, line 14).

Timings

Similarly to our GLS-Montgomery ladder scalar multiplication implementation (see Chapter 3), we run our timing-protected point multiplication in a Intel Core i7-4700MQ (Haswell architecture) with the Turbo Boost and Hyperthreading technologies disabled. The code was programmed in C language and compiled with GCC 4.7.3 with the flags `-m64 -march=core-avx2 -mtune=core-avx2 -O3 -funroll-loops`

[3]In the context of side-channel attacks.

`-fomit-frame-pointer`. In Table 4.1 we present the costs of the functions that form the base of our protected NIST K-283 point multiplication.

Table 4.1: Timings (in clock cycles) for the NIST K-283 elliptic curve operations

Elliptic curve operation	Koblitz $E/\mathbb{F}_{2^{283}}$	
	cycles	op/m[1]
Frobenius	70	1.235
Integer τ-adic recoding (Alg. 12) ($w = 5$)	8,900	156.863
Point addition	602	10.588

[1] Ratio to multiplication in $\mathbb{F}_{2^{283}}$.

In Table 4.2, we show our scalar multiplication results. For benchmarking purposes we also included a baseline implementation of the customary Montgomery López-Dahab ladder. This allows easier comparisons with related work and permits to evaluate the impact of incomplete reduction in the field arithmetic performance.

Table 4.2: Timings (in clock cycles) for different 128-bit secure scalar multiplication implementations with timing-attack resistance in the Intel Ivy Bridge (I) and Haswell (H) architectures

	Method	**Cycles**	**Arch**
State-of-the-art implementations	Montgomery-DJB-chain (prime) [42]	148,000	I
	Random-Montgomery-LD ladder (binary) [25]	135,000	H
	Genus-2-Kummer (prime) [28]	122,000	I
	Koblitz-Montgomery-LD ladder (binary) [25]	118,000	H
	Twisted-Edwards-4-GLV (prime) [50]	92,000	I
	Genus-2-Kummer Montgomery ladder (prime) [17]	72,200	H
	GLS-2-GLV double-and-add (binary, λ) [120]	60,000	H
Our Work	Koblitz-Montgomery-LD (left-to-right)	122,000	H
	Koblitz-regular τ-and-add (left-to-right, $w = 5$)	99,000	H

The fast τ endomorphism allows us to have a regular-recoding implementation that outperforms a standard Montgomery ladder for Koblitz curves by 18%. In addition, our fastest Koblitz code surpasses by 16% the recent implementation reported in [25] [4]. Finally, note that, in spite of the fact that the τ endomorphism is

[4]We could not reproduce the timing of 118,000 cycles with the code available from [25], which indicates that TurboBoost could be possibly turned on on their benchmarks. Considering this,

26% faster than the Montgomery-LD point doubling, the superior efficiency of the GLS quadratic field arithmetic produces faster results for the GLS double-and-add algorithm.

4.2 Koblitz curves over $\mathbb{F}_4$

Koblitz curves defined over $\mathbb{F}_4$ were also proposed in 1991 by Neal Koblitz [99]. However, until nowadays, works related to Koblitz curves have analyzed the security and performance of curves defined only over $\mathbb{F}_{2^m}$, with prime m (for instance, [148, 9, 155]). On the other hand, it has been shown recently [120, 103] that the quadratic extension field arithmetic is quite efficient when implemented in software. This is because we execute the same operation in each base element of the extension field element. For instance, given the quadratic field elements $a = a_0+a_1i$ and $b = b_0+b_1i$, the addition $c = a+b$ can be performed as $c = (a_0+b_0)+(a_1+b_1)i$ (for more examples, see Section 3.1.6). As a result, we can fully employ the current high-end processors pipelines and their inherent instruction-level parallelism.

In this work, we designed and implemented, for the first time, a 128-bit secure and timing attack resistant scalar multiplication on a Koblitz curve defined over $\mathbb{F}_4$. In the next sections, we present the details of the Koblitz curves defined over quadratic extensions along with the field arithmetic functions. Finally, we discuss our implementation timings and compare it against the state-of-the-art works.

4.2.1 Introduction

Let $q = 2^m$, with prime m. Koblitz curves over $\mathbb{F}_4$ are defined by the following equation

$$E_a : y^2 + xy = x^3 + a\gamma x^2 + \gamma, \tag{4.3}$$

where $\gamma \in \mathbb{F}_{2^2}$ satisfies $\gamma^2 = \gamma + 1$ and $a \in \{0, 1\}$.

It is known that, for each proper divisor l of k, $E(\mathbb{F}_{4^l})$ is a subgroup of $E(\mathbb{F}_{4^k})$ and $\#E(\mathbb{F}_{4^l})$ divides $\#E(\mathbb{F}_{4^k})$. Since m is prime, $E_a(\mathbb{F}_{4^m})$ can have almost-prime order (for instance, $E_0(\mathbb{F}_{2^{2\cdot 163}})$ and $E_1(\mathbb{F}_{2^{2\cdot 167}})$).

Note that $\#E_0(\mathbb{F}_4) = 4$ and $\#E_1(\mathbb{F}_4) = 6$. In Table 4.3, we present the group orders $\#E_a(\mathbb{F}_{4^m})$ of Koblitz curves defined over $\mathbb{F}_4$ for prime degrees $m \in [127, 191]$.

our implementation of Koblitz-Montgomery-LD becomes 9% faster than [25], reflecting the savings from partial reduction, and the speedup achieved by the Koblitz-regular implementation increases to 26%.

The chosen range is convenient for implementing a 128-bit secure scalar multiplication on architectures that are provided with 64-bit carry-less multipliers, such as the modern personal desktops.

Table 4.3: Group orders $\#E_a(\mathbb{F}_{2^{2m}})$ with prime $m \in [127, 191]$. Prime factors are underlined. The size (in bits) of the largest prime factor is presented in parenthesis

m	a	**Factorization of** $\#E_a(\mathbb{F}_{2^{2m}})$
127	0	$0x4 \cdot \underline{0x1268F1298760419} \cdot \underline{0xDE7D169BED4130151CD618CF5713077271FF51A4B1CFB75BF}$ (196)
127	1	$0x6 \cdot 0x41603EAF071 \cdot \underline{0x29C4C778B6D2CD0FA36B3CA951A32DAC100C9C63576EEF7BF1F21}$ (209)
131	0	$0x4 \cdot 0x14E3BEE4283C895368536FD0FCF0049D152D78B \cdot \underline{0xC41400B084478F241C495042459}$ (108)
131	1	$0x6 \cdot 0x4267F1026F4F \cdot \underline{0x2806BB97FB5F7C2F9E1EDE20BF59AC390DABBA7621D9A0F26AA1}$ (205)
137	0	$0x4 \cdot 0x763DB379950B73D200B971F1D \cdot \underline{0x22A41FB03F2428B44188DD9FFEA796DC6D197A91BA21}$ (173)
137	1	$0x6 \cdot 0x4337925B3141B99447C1273 \cdot \underline{0x289FE5979AC03A2E5CFCE8E6024FEF0863C633AE96A0DF}$ (182)
149	0	$0x4 \cdot 0x29B66B578C9FAEB \cdot \underline{0x62322066993B57A8857E552587C80A567018483F2E493DBB7750AB7DB623}$ (239)
149	1	$0x6 \cdot \underline{0x1B73C442E8D} \cdot \underline{0x637845F7F8BFAB325B85412FB54061F148B7F6E79AE11CC843ADE1470F7E4E29}$ (255)
151	0	$0x4 \cdot 0x1C4AEB2D8E194A47D0382EB3617226E64298205F16F \cdot \underline{0x90C5C79B46EC78B84E022CB2715ED8281}$ (131)
151	1	$0x6 \cdot 0x1BFFB49BB65DF97968C6F644AF7D0F4DB6F5163 \cdot \underline{0xF9ABD46E3960E5060364D59EBACA8C8326B}$ (140)
157	0	$0x4 \cdot 0x499D09449B55C7D71FC18A2B0265785F \cdot \underline{0x37A45BD5E114A84FCB8900BAEA9E731E0C4B3EDEC15F327}$ (186)
157	1	$0x6 \cdot 0xEECA8C4698A0916800B4E7 \cdot \underline{0xB6F74A858FF10701D113E39259417F04CF038B297F3C6573F6E14F33}$ (224)
163	0	$0x4 \cdot \underline{0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF}\backslash \underline{EA48D724AAB2045E5CFE286F8372017024DFF7BB3}$ (324)
163	1	$0x6 \cdot 0x71977BB40CF524BCA9A8DFB19BD9B251D5 \cdot \underline{0x180A101E65451B46A75AC029CF08711513C17FDE760B92E5}$ (189)

Table 4.3: Continued from previous page

m	a	**Factorization of** $\#E_a(\mathbb{F}_{2^{2m}})$
167	0	$0x4 \cdot 0x6B30E725707929FA94FEFAA012F999 \cdot$ $\underline{0x26364FB489C8B628D0E48E36B3BB4F3C70B651945484571B06BA77}$ (213)
167	1	$0x6 \cdot \underline{0xAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA}\backslash$ $\underline{D45C6A4A8565763007E9FEFA42E0EA9B9E8B7F3541}$ (331)
173	0	$0x4 \cdot 0x163D79633AE74D69B1F95475535FB6B057D397 \cdot$ $\underline{0xB82BEB20E4D8E6D2BFFC1AB84B6BC625C94C6002336E2573F}$ (196)
173	1	$0x6 \cdot 0xBA3DEF139 \cdot$ $\underline{0xEA9746EEF14E1638A503FA6FB739A623894A590811B6939A30D7A016E8A77815}\backslash$ $\underline{0084D9C4D6E0D}$ (308)
179	0	$0x4 \cdot 0x10C01861F3F8F0AC2767CD \cdot$ $\underline{0xF4882969C296A9493FEAA3C9F58DA166B76D3236BF15C2F10E2B0421F3F7E50DCC6F}$ (272)
179	1	$0x6 \cdot 0x9D1C1699F1F6977990F2FFDF75540051322D7023 \cdot$ $\underline{0x116171487AD893A0E28972203861592DD2828EF2D71B9D5B03}$ (196)
181	0	$0x4 \cdot 0xCBB \cdot$ $\underline{0x141BF6E35420FDE10CF60620853943A20D5A91F2F5DDE75B04126F3100B191AF1}\backslash$ $\underline{E338F81FB8ED77C1C57BEF3}$ (348)
181	1	$0x6 \cdot 0x1C0B0F8135C51501AD7DC439F84CF88FA90C9907A08AAE56D243E127CF \cdot$ $\underline{0x615FA176A8D559A3FFDB2ECDACAF97A9B}$ (130)
191	0	$0x4 \cdot 0x65E935E0087F8CBE7343A713158023856DFD17A25EE004B0837F \cdot$ $\underline{0x2831230707A836BC4B2B625A55960A5506F5CCD1B719}$ (174)
191	1	$0x6 \cdot 0x23D01 \cdot$ $\underline{0x4C3F9B376D369D04F03499007A43FE6460A012C86B2C575858EE9FC7F67A566813}\backslash$ $\underline{B39DA28DC9D58285BC07F8811}$ (362)

The τ-adic representation

Given a Koblitz curve $E_a/\mathbb{F}_{2^{2m}}$ with group order $\#E_a(\mathbb{F}_{2^{2m}}) = h \cdot p \cdot r$, where r is the order of our subgroup of interest, we can express a scalar $k \in \mathbb{Z}_r$ as an element in $\mathbb{Z}[\tau]$ using the partial reduction by Solinas [145] with a few modifications. The modified version is based on the fact that $\tau^2 = \mu\tau - 4$ and is presented in Algorithm 14. The `Round` function is the $\mathbb{Z}[\tau]$ rounding-off method described in [145, Routine 60].

Given that the norm of τ is $N(\tau) = 4$, $N(\tau - 1) = h$, $N(\tau^m - 1) = h \cdot p \cdot r$ and $N((\tau^m - 1)/(\tau - 1)) = p \cdot r$, the subscalars r_0 and r_1 resulted from the partial

Algorithm 14 Partial reduction modulo $(\tau^m - 1)/(\tau - 1)$

```
Input: The scalar $k \in [1, r-1]$, $s_0 = d_0 + \mu d_1$, $s_1 = -d_1$, where
    $(\tau^m - 1)/(\tau - 1) = d_0 + d_1\tau$
Output: $\rho = (r_0 + r_1\tau) = k$ partmod $(\tau^m - 1)/(\tau - 1)$
1: $t \leftarrow s_0 + \mu \cdot s_1$
2: $\lambda_0 \leftarrow s_0 \cdot k/(p \cdot r)$
3: $\lambda_1 \leftarrow s_1 \cdot k/(p \cdot r)$
4: $q_0, q_1 \leftarrow \texttt{Round}(\lambda_0, \lambda_1)$
5: $r_0 \leftarrow k - t \cdot q_0 - 4 \cdot s_1 \cdot q_1$
6: $r_1 \leftarrow s_1 \cdot q_0 - s_0 \cdot q_1$
7: return $(r_0, r_1)$.
```

modulo function will be both of size approximately $(p \cdot r)/2$. As a consequence, our scalar multiplication will need more iterations than expected, since it will consider the order p of a subgroup which is not of cryptographic interest.

For that reason, we considered that the input scalar of our point multiplication algorithm is already represented in $\mathbb{Z}[\tau]$. As a result, it is not required to perform a partial reduction in the scalar k, and the number of iterations in the point multiplication will be consistent with the scalar k size. If one needs to retrieve the scalar k represented in $\mathbb{Z}_r$, it can be easily computed with one multiplication and one addition in $\mathbb{Z}_r$. This design decision was based on the degree-2 scalar decomposition method, in the GLS curves context, suggested in [61].

The width-w τNAF form

After representing the scalar k in $\mathbb{Z}[\tau]$, we can apply the slightly modified version of the Algorithm 12 in order to express the scalar in the regular width-w τNAF form. The adjusted method is presented in Algorithm 15.

Given a width w, after running Algorithm 15, we have $2^{2(w-1)-1}$ digits[5]. As a result, it is necessary to be more conservative when choosing the width w, when compared to the Koblitz curves defined over $\mathbb{F}_2$. For widths $w = 2, 3, 4, 5$ we have to pre- or post-compute $2, 8, 32$ and 128 points, respectively. For the 128-bit point multiplication, we estimated that the value of the width w must be at most four, otherwise, the costs of the point pre/post-processing are greater than the addition savings obtained in the main iteration.

[5]We are considering only positive digits, since the cost of applying signs to points in binary elliptic curves is negligible.

Algorithm 15 Regular width-w τ-recoding for m-bit scalar

```
Input: $w, t_w, \alpha_u = \beta_u + \gamma_u\tau$ for $u = \{\pm 1, \pm 3, \pm 5, \ldots, \pm 4^{w-1} - 1\}, \rho = r_0 + r_1\tau \in \mathbb{Z}[\tau]$
    with odd $r_0, r_1$
Output: $\rho = \sum_{i=0}^{\lceil \frac{m+2}{w-1} \rceil} u_i \tau^{i(w-1)}$
 1: for $i \leftarrow 0$ to $\lceil \frac{m+2}{w-1} \rceil$ - 1 do
 2:     if $w = 2$ then
 3:         $u_i \leftarrow ((r_0 - 4 \cdot r_1) \mod 8) - 4$
 4:         $r_0 \leftarrow r_0 - u_i$
 5:     else
 6:         $u \leftarrow (r_0 + r_1 t_w \bmod 2^{2w-1}) - 2^{2(w-1)}$
 7:         if $u > 0$ then $s \leftarrow 1$ else $s \leftarrow -1$
 8:         $r_0 \leftarrow r_0 - s\beta_u, r_1 \leftarrow r_1 - s\gamma_u, u_i \leftarrow s\alpha_u$
 9:     end if
10:     for $j \leftarrow 0$ to $(w-2)$ do
11:         $t \leftarrow r_0, r_0 \leftarrow r_1 + (\mu \cdot r_0)/4, r_1 \leftarrow -t/4$
12:     end for
13: end for
14: if $r_0 \neq 0$ and $r_1 \neq 1$ then
15:     $u_i \leftarrow r_0 + r_1\tau$
16: else
17:     if $r_1 \neq 0$ then
18:         $u_i \leftarrow r_1$
19:     else
20:         $u_i \leftarrow r_0$
21:     end if
22: end if
```

In addition, we must find efficient expressions of $\alpha_u = u \pmod{\tau^w}$. The method for searching the best expressions in Koblitz curves over $\mathbb{F}_2$ [150] cannot be directly applied in the $\mathbb{F}_4$ case. As a result, we manually provided α_u representations for $w \in \{2, 3\}$ and $a = 1$, which are our implementation parameters. In Table 4.5 we present the α_u representations along with the operations required to generate the points.

Therefore, one point doubling and full addition are required to generate the points α_u for $w = 2$ and one point doubling, four full additions, three mixed additions and four applications of the Frobenius map for the $w = 3$ case.

4.2.2 Base field arithmetic

In this section, we present the techniques used in our work in order to implement the binary field arithmetic. We selected a Koblitz curve with the parameter $a = 1$ defined over $\mathbb{F}_{4^m}$ with $m = 149$. This curve was chosen because the order of its subgroup of interest is of size 2^{254} (see Table 4.3), which is a security-level equivalent of a 128-bit secure scalar multiplication.

Table 4.5: Representations of $\alpha_u = u \pmod{\tau^w}$, for $w \in \{2,3\}$ and $a = 1$ and the required operations for computing α_u. Here we denote by D, FA, MA, T the point doubling, full addition, mixed addition and the Frobenius map, respectively. In addition, we consider that the point α_1 is represented in affine coordinates

w	u	$u \,(\mathrm{mod}\, \tau^w)$	α_u	**Operations**
2	1	1	1	n/a
	3	3	3	$t_0 \leftarrow 2\alpha_1$, $\alpha_3 \leftarrow t_0 + \alpha_1$ $(D + FA)$
3	1	1	1	n/a
	3	3	3	$t_0 \leftarrow 2\alpha_1$, $\alpha_3 \leftarrow t_0 + \alpha_1$ $(D + FA)$
	5	5	$-\tau - \alpha_{15}$	$\alpha_5 \leftarrow -t_1 - \alpha_{15}$ (MA)
	7	$3\tau + 3$	$\tau^2\alpha_3 + \alpha_3$	$\alpha_7 \leftarrow \tau^2\alpha_3 + \alpha_3$ $(FA + 2T)$
	9	$3\tau + 5$	$\alpha_7 + 2$	$\alpha_9 \leftarrow \alpha_7 + t_0$ (FA)
	11	$3\tau + 7$	$\alpha_9 + 2$	$\alpha_{11} \leftarrow \alpha_9 + t_0$ (FA)
	13	$-\tau - 7$	$\tau^2 - \alpha_3$	$\alpha_{13} \leftarrow t_2 - \alpha_3$ (MA)
	15	$-\tau - 5$	$\tau^2 - 1$	$t_1 \leftarrow \tau\alpha_1$, $t_2 \leftarrow \tau t_1$, $\alpha_{15} \leftarrow t_2 - \alpha_1$ $(MA + 2T)$

Modular reduction

As discussed in Section 3.1, we can construct a binary extension field $\mathbb{F}_{2^m}$ by taking a polynomial $f(x) \in \mathbb{F}_2[x]$ of degree m which is irreducible over $\mathbb{F}_2$. Also, it is very important that the form of our polynomial $f(x)$ allows us to efficiently compute the modular reduction. The criteria for selecting $f(x)$ depends on the architecture to be implemented and was extensively discussed in [142].

In our case, we do not have degree-149 trinomials which are irreducible over $\mathbb{F}_2$. An alternative solution is to construct the field through irreducible pentanomials. Given an irreducible pentanomial $f(x) = x^m + x^a + x^b + x^c + 1$, the efficiency of the shift-and-add reduction method depends on that (mostly of) the term degree differences $m - a$, $m - b$ and $m - c$ be equal to 0 modulo W, where W is the architecture word size in bits. Since our scalar multiplication is implemented with the SSE/AVX set of instructions, which provides byte shifts in one clock cycle, we considered $W = 8$, however we could obtain important speed-ups if $W = 64$ or 128.

Using the terminology of [142], lucky irreducible pentanomials are the ones where the three previously mentioned differences are equal to 0 modulo W. Fortunate irreducible pentanomials are the ones whose two of the differences are equal to 0 modulo W. The remaining cases are called ordinary irreducible pentanomials. Performing an extensive search with $W = 8$, we found no lucky pentanomials, 189 fortunate pentanomials and 9491 ordinary pentanomials.

The problem is that fortunate pentanomials make the modular reduction too costly if we compare with the field multiplication computed with carry-less instructions. This is because we need to perform four shift-and-add operations per reduction step. Besides, two of those operations require complex shift instructions, since they are shifts not divisible by 8.

Redundant trinomials As a consequence, we considered the redundant trinomials option introduced in [33, 45]. Given a non-irreducible trinomial $g(x)$ of degree n that factorizes into an irreducible polynomial $f(x)$ of degree $m < n$, the idea is to perform the field reduction modulo $g(x)$ throughout the scalar multiplication and, at the end of the algorithm, reduce the point coordinates modulo $f(x)$.

In other words, throughout the algorithm, we represent the base field elements as polynomials in the ring $\mathbb{F}_2[x]$ reduced modulo $g(x)$. At the end of the algorithm, the elements are reduced modulo $f(x)$ in order to bring them to the field $\mathbb{F}_{2^{149}}$. For the sake of simplicity, throughout this chapter, we will refer to those elements as field elements.

Since our architecture is embedded with a 64-bit carry-less multiplier, an efficient representation of the field elements must have at most 192 bits (three 64-bit words). For that reason, we searched for redundant trinomials of degree at most 192. In Table 4.6, we present the available redundant trinomials.

We selected the trinomial $g(x) = x^{192} + x^{19} + 1$ for two reasons. First, the difference $(m - a) > 128$, which allow us to perform the shift-and-add reduction in just two steps, since our architecture contains 128-bit vectorial registers. Second, the property $m \mod 64 = 0$ allow us to perform efficiently the first part of the shift-and-add reduction. The steps to perform the modular reduction are described in Algorithm 16. The notation is similar to the one presented in Section 3.1.4 but here translated to 64-bit registers. The reduction using 128-bit registers is presented later, in Section 4.2.3, which discusses the arithmetic in the quadratic field extension.

Algorithm 16 Modular reduction by the trinomial $g(x) = x^{192} + x^{19} + 1$

```
Input: A 384-bit polynomial $r(x) = F \cdot x^{320} + E \cdot x^{256} + D \cdot x^{192} + C \cdot x^{128} + B \cdot x^{64} + A$
    in $F_2[x]$ stored into six 64-bit registers (A - F).
Output: A 192-bit polynomial $s(x) = r(x) \mod g(x) = I \cdot x^{128} + H \cdot x^{64} + G$ stored
    into three 64-bit registers (G - I).
1: $G \leftarrow A \oplus D \oplus (F \gg 45) \oplus ((D \oplus (F \gg 45)) \ll 19)$
2: $H \leftarrow B \oplus E \oplus (E \ll 19) \oplus (D \gg 45)$
3: $I \leftarrow C \oplus F \oplus (F \ll 19) \oplus (E \gg 45)$
```

Table 4.6: Redundant trinomials $g(x) = x^m + x^a + 1$ of degree ≤ 192 which factorizes into a irreducible polynomial of degree 149

Trinomial	$m - a$	$m \mod 64$	$(m - a) \mod 64$
$x^{151} + x^2 + 1$	149	23	21
$x^{151} + x^{149} + 1$	2	23	2
$x^{156} + x^{73} + 1$	83	28	19
$x^{156} + x^{83} + 1$	73	28	9
$x^{163} + x^{61} + 1$	102	35	38
$x^{163} + x^{80} + 1$	83	35	19
$x^{163} + x^{83} + 1$	80	35	16
$x^{163} + x^{102} + 1$	61	35	61
$x^{166} + x^{43} + 1$	123	38	59
$x^{166} + x^{123} + 1$	43	38	43
$x^{169} + x^{53} + 1$	116	41	52
$x^{169} + x^{116} + 1$	53	41	53
$x^{173} + x^{36} + 1$	137	45	9
$x^{173} + x^{137} + 1$	36	45	36
$x^{179} + x^{78} + 1$	101	51	37
$x^{179} + x^{101} + 1$	78	51	14
$x^{187} + x^{15} + 1$	172	59	44
$x^{187} + x^{172} + 1$	15	59	15
$x^{191} + x^{74} + 1$	117	63	53
$x^{191} + x^{117} + 1$	74	63	10
$x^{192} + x^{19} + 1$	173	0	45
$x^{192} + x^{173} + 1$	19	0	19

The overall cost of the modular reduction is ten `xors` and six bitwise shifts. At the end of the scalar multiplication, we have to reduce the 192-bit polynomial to an element of the field $\mathbb{F}_{2^{149}}$. Note that the trinomial $g(x) = x^{192} + x^{19} + 1$ factorizes into a 69-term irreducible polynomial $f(x)$ of degree 149 defined by

$$\begin{aligned} f(x) =& x^{149} + x^{146} + x^{143} + x^{141} + x^{140} + x^{139} + x^{138} + x^{137} + x^{129} + x^{123} + x^{122} + \\ & x^{121} + x^{119} + x^{117} + x^{114} + x^{113} + x^{111} + x^{108} + x^{107} + x^{106} + x^{105} + x^{99} + \\ & x^{94} + x^{92} + x^{91} + x^{90} + x^{86} + x^{85} + x^{83} + x^{81} + x^{80} + x^{78} + x^{77} + x^{75} + \\ & x^{71} + x^{70} + x^{68} + x^{67} + x^{65} + x^{64} + x^{63} + x^{54} + x^{53} + x^{51} + x^{49} + x^{48} + \\ & x^{43} + x^{42} + x^{41} + x^{40} + x^{39} + x^{38} + x^{37} + x^{35} + x^{28} + x^{26} + x^{23} + x^{18} + \\ & x^{17} + x^{16} + x^{15} + x^{12} + x^{11} + x^{10} + x^9 + x^3 + x^2 + x + 1. \end{aligned}$$

The final reduction is performed via the mul-and-add reduction[6] which, experimentally, performed more efficiently than the shift-and-add reduction. Concisely, the mul-and-add technique consists in a series of steps which includes shifts (in order to align the bits in the registers), carry-less multiplications and `xors` for eliminating the extra bits.

The basic mul-and-add step is described in Algorithm 17. Here, besides the usual notation, we represent the 64-bit carry-less multiplication by the symbol $\times_{ij}$, where $i, j = \{L, H\}$, with L and H representing the lowest and highest 64-bit word packed in a 128-bit register, respectively. For example, if we want to multiply the 128-bit register A lowest 64-bit word by the 128-bit register B highest 64-bit word, we would express this operation as $T \leftarrow A \times_{LH} B$.

Algorithm 17 Basic step of the mul-and-add reduction modulo the 69-term irreducible polynomial $f(x)$

```
Input: A j-bit polynomial $r(x) = B \cdot x^{128} + A$ stored into two 128-bit registers
    (A, B), the irreducible polynomial $f(x) = F \cdot x^{128} + E$ stored into two 128-bit
    registers (E, F).
Output: A $(j-3)$-bit polynomial $s(x) = D \cdot x^{128} + C$ stored into two 128-bit registers
    (C, D).
1: $T_0 \leftarrow B \gg 21$ (64-bit word alignment, $149 \mod 64 = 21$)
2: $T_1 \leftarrow E \times_{LL} T_0$
3: $T_2 \leftarrow E \times_{HL} T_0$
4: $T_0 \leftarrow F \times_{LL} T_0$
5: $T_1 \leftarrow T_1 \oplus (T_2 \ll 64)$
6: $T_0 \leftarrow T_0 \oplus (T_2 \gg 64)$
7: $C \leftarrow A \oplus T_1$
8: $D \leftarrow B \oplus T_0$
```

The algorithm requires four `xors`, three bitwise shifts and three carry-less multiplications. In our particular case, the difference between the degrees of the two most significant monomials of $f(x)$ is three. Also, note that we need to reduce 43 bits (191-148). As a result, it is required $\lceil \frac{43}{3} \rceil = 15$ applications of the Algorithm 17 in order to conclude the reduction.

[6]For a more detailed explanation of the shift-and-add and the mul-and-add reduction methods to binary fields, see [25].

4.2.3 Quadratic field arithmetic

In this part, we present the arithmetic functions in the quadratic field. Most of the issues discussed here are related to our concrete implementation, since the basic algorithms are similar to the ones examined in Section 3.1. As usual, our quadratic field $\mathbb{F}_{2^{2\cdot 149}}$ was constructed by the degree two monic polynomial $h(u) = u^2 + u + 1$, and its elements are represented as $a_0 + a_1 u$, with $a_0, a_1 \in \mathbb{F}_{2^{149}}$.

Register allocation The first topic to analyze is the element allocation into the architecture available registers. In our case, we have to store two polynomials of 192 bits into 128-bit registers in such way that it allows an efficient modular reduction and, at the same time, generates a minimum overhead in the two main arithmetic operations, namely, the multiplication and squaring.

Let us consider an element $a = (a_0 + a_1 u) \in \mathbb{F}_{2^{2\cdot 149}}$, where $a_0 = C \cdot x^{128} + B \cdot x^{64} + A$ and $a_1 = F \cdot x^{128} + E \cdot x^{64} + D$ are 192-bit polynomials, each stored in three 64-bit words (A-C, D-F). Also, let us have three 128-bit registers R_i, with $i \in \{0, 1, 2\}$, which can store two 64-bit words each. In this section, we adopted the following notation, given a 128-bit register R, its most and least significant packed 64-bit words, denoted respectively by S and T, are represented as $R = S|T$.

The first option is to rearrange the 384-bit element $a = (a_0 + a_1 u)$ as

$$R_0 = D|A, \quad R_1 = C|D, \quad R_2 = E|F.$$

The problem with this representation is that a significant overhead is generated in the multiplication function, more specifically, in the pre-computation phase of the Karatsuba procedure (see the Multiplication paragraph below, computation of $V_{0,1}$, $V_{0,2}$ and $V_{1,2}$). Besides, in order to efficiently perform the subsequent reduction phase, we must temporarily store the polynomial terms into four 128-bit vectors, which can cause a register overflow. A better method for storing the element a is presented as follows,

$$R_0 = D|A, \quad R_1 = E|B, \quad R_2 = F|C.$$

Now, we still have some overhead in the multiplication and squaring functions, even though the penalty on the latter function is almost negligible. However, the terms of the elements a_0, a_1 do not need to be rearranged and the modular reduction of the these two base field elements can be performed in parallel, as discussed in the next paragraphs.

Multiplication Given two $\mathbb{F}_{2^{2\cdot 149}}$ elements $a = (a_0 + a_1u)$ and $b = (b_0 + b_1u)$, with a_0, a_1, b_0, b_1 in $\mathbb{F}_{2^{149}}$, we perform the multiplication $c = a \cdot b$ as,

$$\begin{aligned} c = a \cdot b &= (a_0 + a_1u) \cdot (b_0 + b_1u) \\ &= (a_0b_0 + a_1b_1) + (a_0b_0 + a_1b_1 + (a_0 + a_1) \cdot (b_0 + b_1))u, \end{aligned}$$

where each element $a_i, b_i \in \mathbb{F}_{2^{149}}$ is composed by three 64-bit words. The analysis of the Karatsuba algorithm cost for different word sizes was presented in [153]. There, it was shown that the most efficient way to multiply three 64-bit word polynomials $s(x) = s_2x^2 + s_1x + s_0$ and $t(x) = t_2x^2 + t_1x + t_0$ as $v(x) = s(x) \cdot t(x)$ is through the one-level Karatsuba method,

$$V_0 = s_0 \cdot t_0, \quad V_1 = s_1 \cdot t_1, \quad V_2 = s_2 \cdot t_2,$$

$$V_{0,1} = (s_0 + s_1) \cdot (t_0 + t_1), \quad V_{0,2} = (s_0 + s_2) \cdot (t_0 + t_2) \quad V_{1,2} = (s_1 + s_2) \cdot (t_1 + t_2),$$

$$v(x) = V_2 \cdot x^4 + (V_{1,2} + V_1 + V_2) \cdot x^3 + (V_{0,2} + V_0 + V_1 + V_2) \cdot x^2 + (V_{0,1} + V_0 + V_1) \cdot x + V_0,$$

which costs six multiplications and twelve additions. The Karatsuba algorithm is presented in the Algorithm 18.

Algorithm 18 Karatsuba algorithm for multiplying three 64-bit word polynomials $s(x)$ and $t(x)$

Input: Six 128-bit registers R_i, with $i \in \{0 \dots 5\}$, containing the elements $R_0 = t_0|s_0, R_1 = t_1|s_1$, $R_2 = t_2|s_2, R_3 = (t_0 \oplus t_1)|(s_0 \oplus s_1)$, $R_4 = (t_0 \oplus t_2)|(s_0 \oplus s_2)$, $R_5 = (t_1 \oplus t_2)|(s_1 \oplus s_2)$.

Output: Three 128-bit registers R_i, with $i \in \{6 \dots 8\}$, which store the value $v(x) = s(x) \cdot t(x) = v_5 \cdot x^{320} + v_4 \cdot x^{256} + v_3 \cdot x^{192} + v_2 \cdot x^{128} + v_1 \cdot x^{64} + v_0$ as $R_6 = v_1|v_0, R_7 = v_3|v_2, R_8 = v_5|v_4$.

```
1: $tmp_0 \leftarrow R_0 \times_{HL} R_0$
2: $tmp_1 \leftarrow R_1 \times_{HL} R_1$
3: $tmp_2 \leftarrow R_2 \times_{HL} R_2$
4: $tmp_3 \leftarrow R_3 \times_{HL} R_3$
5: $tmp_4 \leftarrow R_4 \times_{HL} R_4$
6: $tmp_5 \leftarrow R_5 \times_{HL} R_5$
7: $tmp_5 \leftarrow tmp_5 \oplus tmp_1$
8: $tmp_5 \leftarrow tmp_5 \oplus tmp_2$
9: $tmp_1 \leftarrow tmp_1 \oplus tmp_0$
10: $tmp_4 \leftarrow tmp_4 \oplus tmp_1$
11: $tmp_4 \leftarrow tmp_4 \oplus tmp_2$
12: $tmp_3 \leftarrow tmp_3 \oplus tmp_1$
13: $R_6 \leftarrow (tmp_3 \ll 64)$
14: $R_8 \leftarrow (tmp_5 \gg 64)$
15: $R_7 \leftarrow ((tmp_5, tmp_3) \rhd 64)$
```

The algorithm requires six carry-less instructions, six vectorial `xors` and three bitwise shift instructions. In order to calculate the total multiplication cost, it is necessary to include the Karatsuba pre-computation operations at the base field level (tweve vectorial `xors` and six byte interleaving instructions) and at the quadratic field level (six vectorial `xors`). Also, we must consider the reorganization of the registers in order to proceed with the modular reduction (six vectorial `xors`).

Modular reduction The modular reduction of an element $a \in \mathbb{F}_{2^{2\cdot 149}}$ takes nine vectorial `xors` and six bitwise shifts. The gains of the previously discussed register configuration can be seen when we compare the reduction of quadratic field elements, presented in Algorithm 19 with the modular reduction of the base field elements (see Algorithm 16). The cost of reducing an element in $\mathbb{F}_{2^{149}}$ in 64-bit registers is similar to the cost of the reduction of an element in $\mathbb{F}_{2^{2\cdot 149}}$ stored into 128-bit registers. Thus, we achieved the expected speedup of 100%.

Algorithm 19 Modular reduction of the terms a_0, a_1 of an element $a = (a_0 + a_1 u)$ modulo $g(x) = x^{192} + x^{19} + 1$

Input: An element $a = a_0 + a_1 u = (F \cdot x^{320} + E \cdot x^{256} + D \cdot x^{192} + C \cdot x^{128} + B \cdot x^{64} + A) + (L \cdot x^{320} + K \cdot x^{256} + J \cdot x^{192} + I \cdot x^{128} + H \cdot x^{64} + G)u$, with the 64-bit words (A-L) arranged in six 128-bit registers as $R_0 = G|A, R_1 = H|B, R_2 = I|C, R_3 = J|D, R_4 = K|E, R_5 = L|F$

Output: Elements $(a_0, a_1) \bmod g(x) = M \cdot x^{128} + N \cdot x^{64} + O, P \cdot x^{128} + Q \cdot x^{64} + R$, with the 64-bit words (M-R) organized in three 128-bit registers as $R_6 = R|O, R_7 = Q|N, R_8 = P|M$

```
1: $R_8 \leftarrow R_2 \oplus R_5$
2: $R_7 \leftarrow R_1 \oplus R_4$
3: $R_8 \leftarrow R_8 \oplus (R_5 \ll 19)$
4: $R_7 \leftarrow R_7 \oplus (R_4 \ll 19)$
5: $R_8 \leftarrow R_8 \oplus (R_4 \gg 45)$
6: $R_7 \leftarrow R_7 \oplus (R_3 \ll 19)$
7: $R_6 \leftarrow R_3 \oplus (R_5 \gg 45)$
8: $R_6 \leftarrow R_6 \oplus (R_6 \ll 19)$
9: $R_6 \leftarrow R_6 \oplus R_0$
```

Squaring Squaring is a very important function in the Koblitz curve point multiplication algorithm, since it forms the building block for computing the τ endomorphism. In our implementation, we computed the squaring operation through carry-less multiplication instructions which, experimentally, was less expensive than the bit interleaving method (see [78, Section 2.3.4]). The pre-processing phase is straightforward, we just need to rearrange the 32-bit packed words of the 128-bit registers in order to prepare them for the subsequent modular reduction.

The pre- and post-processing phases require three shuffle instructions, three vectorial `xors` and three bitwise shifts. The complete function is described in Algorithm 20. Given 128-bit registers R_i, we depict the SSE 32-bit shuffle operation as

$R_0 \leftarrow R_1 \between xxxx$. For instance, if we compute $R_0 \leftarrow R_1 \between 3210$, it just maintains the 32-bit word order of the register R_1, in other words, it just copy R_1 to R_0. The operation $R_0 \leftarrow R_1 \between 2103$ rotates the register R_1 to the left by 32-bits. See [130, 129] for more details.

Algorithm 20 Squaring of an element $a = (a_0 + a_1 u) \in \mathbb{F}_{2^{2 \cdot 149}}$

Input: Element $a = a_0 + a_1 u = (C \cdot x^{128} + B \cdot x^{64} + A) + (F \cdot x^{128} + E \cdot x^{64} + D)u \in \mathbb{F}_{2^{2 \cdot 149}}$, with the 64-bit words (A-F) arranged in three 128-bit registers as $R_0 = D|A, R_1 = E|B, R_2 = F|C$

Output: Element $a^2 = c = c_0 + c_1 u = (I \cdot x^{128} + H \cdot x^{64} + G) + (L \cdot x^{128} + K \cdot x^{64} + J)u \in \mathbb{F}_{2^{2 \cdot 149}}$, where both elements $(c_0, c_1) \in \mathbb{F}_2[x]$ are reduced modulo $x^{192} + x^{19} + 1$. The 64-bit words (G-L) are arranged in three 128-bit registers as $r_3 = J|G, r_4 = H|K, r_5 = I|L$.

```
1: $tmp_0 \leftarrow r_0 \between 3120$
2: $tmp_1 \leftarrow r_1 \between 3120$
3: $tmp_2 \leftarrow r_2 \between 3120$
4: $aux_0 \leftarrow tmp_0 \times_{LL} tmp_0$
5: $aux_1 \leftarrow tmp_0 \times_{HH} tmp_0$
6: $aux_2 \leftarrow tmp_1 \times_{LL} tmp_1$
7: $aux_3 \leftarrow tmp_1 \times_{HH} tmp_1$
8: $aux_4 \leftarrow tmp_2 \times_{LL} tmp_2$
9: $aux_5 \leftarrow tmp_2 \times_{HH} tmp_2$
10: $r_3, r_4, r_5 \leftarrow \texttt{ModularReduction}(aux_{0...5})$
11: $tmp_0 \leftarrow r_3 \gg 64$
12: $tmp_1 \leftarrow r_4 \gg 64$
13: $tmp_2 \leftarrow r_5 \gg 64$
14: $r_3 \leftarrow r_3 \oplus tmp_0$
15: $r_4 \leftarrow r_4 \oplus tmp_1$
16: $r_5 \leftarrow r_5 \oplus tmp_2$
```

Inversion The inversion operation is computed via the Itoh-Tsujii method [84]. Given an element $c \in \mathbb{F}_{2^m}$, we compute $c^{-1} = c^{(2^{m-1}-1) \cdot 2}$ through an addition chain. For the case $m = 149$, the following chain is used,

$$1 \leftarrow 2 \leftarrow 4 \leftarrow 8 \leftarrow 16 \leftarrow 32 \leftarrow 33 \leftarrow 66 \leftarrow 74 \leftarrow 148.$$

This addition chain is optimal and was found through the procedure described in [38]. Note that although we compute the inversion operation over polynomials in $\mathbb{F}_2[x]$ (reduced modulo $g(x) = x^{192} + x^{19} + 1$), we still have to perform the addition chain with $m = 149$, since we are in fact interested in the embedded $\mathbb{F}_{2^{149}}$ field element.

The addition chain is computed by a series of multiplications and squarings. Given an element $a_0 \in \mathbb{F}_{2^{149}}$, in each step we calculate ${a_0}^{2^i - 1}$, where the value i represents the integers that form the addition chain. Experimentally, we found that when $i \geq 4$, it is cheaper to compute the exponentiation through table look-ups. Our pre-computed tables process four bits per iteration, therefore, it is required $\lceil \frac{192}{4} \rceil = 48$ table queries in order to complete the multisquaring function.

4.2.4 τ-and-add scalar multiplication

In this part we discuss the single-core algorithms that compute a timing-resistant scalar multiplication through the τ-and-add method over Koblitz curves defined over $\mathbb{F}_4$. There are two basic approaches, the right-to-left and the left-to-right algorithms. Those methods differ by the order which the scalar digits are processed in the main iteration of the point multiplication.

Left-to-right τ-and-add This algorithm is similar to the left-to-right double-and-add approach discussed in Section 3.3.2. Here, the point doubling operation is replaced by the computationally cheaper τ endomorphism. In addition, we need to compute the width w-τNAF representation of the scalar k and perform linear passes (this function is discussed at the end of this section) in the accumulators in order to avoid cache attacks [151, 121]. The method is shown in Algorithm 21.

Algorithm 21 Left-to-right regular w-TNAF τ-and-add on Koblitz curves defined over $\mathbb{F}_4$

```
Input: A Koblitz curve $E_a/\mathbb{F}_{2^{2m}}$, a point $P \in E_a(\mathbb{F}_{2^{2m}})$ of order $r$, $k \in \mathbb{Z}_r$
Output: $Q = kP$
 1: Compute $\rho = r_0 + r_1\tau = k$ partmod $\left(\frac{\tau^m-1}{\tau-1}\right)$
 2: Ensure that $r_0$ and $r_1$ are odd.
 3: Compute the width-$w$ regular $\tau$-NAF of $r_0 + r_1\tau$ as $\sum_{i=0}^{\lceil\frac{m+2}{w-1}+1\rceil} u_i\tau^{i(w-1)}$
 4: for $i \in \{1, 3, \ldots 4^{w-1} - 1\}$ do Compute $P_i$ end for
 5: $Q \leftarrow \mathcal{O}$
 6: for $i = \frac{m+2}{w-1} + 1$ to 0 do
 7:     $Q \leftarrow \tau^{w-1}(Q)$
 8:     Perform a linear pass to recover $P_{u_i}$
 9:     $Q \leftarrow Q \pm P_{u_i}$
10: end for
11: Subtract $P, \tau(P)$ if necessary
12: return $Q = kP$
```

The main advantage of this method is that the sensitive data is indirectly placed in the points P_{u_i}. However, those points are only read and then added to the unique accumulator Q. As a consequence, only one linear pass per iteration is required before reading P_{u_i}. On the other hand, the operation $\tau^{w-1}(Q)$ must be performed by successive squarings, since computing it through look-up tables could leak information about the scalar k.

Right-to-left τ-and-add This other method processes the scalar k from the least to the most significant digit. It is similar to the algorithm depicted in Chapter 3. In this case, the point halving is substituted by the τ endomorphism, and the GLV method is brought to its full extent, through the τ-adic representation of the scalar k. This approach is presented in Algorithm 22.

Algorithm 22 Right-to-left regular w-TNAF τ-and-add on Koblitz curves defined over $\mathbb{F}_4$

```
Input: A Koblitz curve $E_a/\mathbb{F}_{2^{2m}}$, a point $P \in E_a(\mathbb{F}_{2^{2m}})$ of order $r$, $k \in \mathbb{Z}_r$
Output: $Q = kP$
 1: Compute $\rho = r_0 + r_1\tau = k$ partmod $\pmod{\frac{\tau^m-1}{\tau-1}}$
 2: Ensure that $r_0$ and $r_1$ are odd.
 3: Compute the width-w regular $\tau$-NAF of $r_0 + r_1\tau$ as $\sum_{i=0}^{\lceil\frac{m+2}{w-1}+1\rceil} u_i\tau^{i(w-1)}$
 4: for $i \in \{1, 3, \dots 4^{w-1} - 1\}$ do $Q_i = \mathcal{O}$
 5: for $i = 0$ to $\frac{m+2}{w-1} + 1$ do
 6:     Perform a linear pass to recover $Q_{u_i}$
 7:     $Q_{u_i} \leftarrow Q_{u_i} \pm P$
 8:     Perform a linear pass to store $Q_{u_i}$
 9:     $P \leftarrow \tau^{w-1}(P)$
10: $Q \leftarrow \mathcal{O}$
11: for $i \in \{1, 3, \dots 4^{w-1} - 1\}$ do $Q = i \cdot Q_i$
12: Subtract $P, \tau(P)$ if necessary
13: return $Q = kP$
```

Here, we have to perform a post-computation in the accumulators instead of precomputing the points P_i as in the previous approach. Also, the τ endomorphism is applied to the point P, which is usually public. For that reason, we can compute τ with table look-ups instead of performing squarings multiple times.

The downside of this algorithm is that the accumulators carry sensitive information about the digits of the scalar. Also, the accumulators are read and written. As a result, it is necessary to apply the linear pass algorithm to the accumulators Q_i twice per iteration.

Linear pass The linear pass is a method designed to protect sensitive information against side-channel attacks associated with the CPU cache access patterns. Let us consider an array A of size l. Before reading a value $A[i]$, with $i \in [0, l-1]$, the linear pass technique reads the entire array A but only stores, possibly into an output register, the requested value $A[i]$. In that way, the attacker does not know which array index was accessed just by analyzing the location of the cache-miss in his own artificially injected data. However, this method causes a considerable overhead, which depends on the size of the array.

In this work, we implemented the linear pass method using 128-bit SSE vectorial instructions and registers. For each array index i, we copy it to a register and compare it with the current scalar k τNAF digit. The SSE instruction `pcmpeqq` compares the values of two 128-bit registers A and B and sets the resulting register C with bits one, if A and B are equal, and bits zero otherwise. For that reason, we can use the register C as a mask, which is applied to each value $A[i]$ before copying it to the register that will hold the requested data.

Experimental results shown that the implementation of the linear pass technique with SSE registers is more efficient than using 64-bit conditional move instructions [120] by an order of 2.125. Our approach is depicted in Algorithm 23.

Algorithm 23 Linear pass using 128-bit AVX vectorial instructions

```
Input: An array $A$ of size $l$, a requested index $i$, SSE 128-bit registers $tmp, dst$.
Output: The register $dst$ containing $A[d]$.
1: $dst \leftarrow 0$
2: for $i \in \{0, \dots, l-1\}$ do
3:     $tmp \leftarrow i$
4:     $tmp \leftarrow$ compare( $tmp, i$ )
       (compare returns $1^{128}$ if the operands are equal and $0^{128}$ otherwise.)
5:     $tmp \leftarrow tmp \wedge P_i$
6:     $dst \leftarrow dst \vee tmp$
7: end for
```

4.2.5 Results and discussion

Our software library can be executed in any Intel platform, which comes with the SSE4.1 vector instructions and the 64-bit carry-less multiplier instruction `pclmulqdq`. The benchmarking was executed in a Intel Core i7 4770k 3.50 GHz machine (Haswell architecture) with the TurboBoost and HyperThreading features disabled. Also, the library was coded in the GNU11 C and Assembly languages.

Regarding the compilers, we performed an experimental analysis on the performance of our code compiled with different systems: GCC (Gnu Compiler Collection) versions 4.7, 4.8, 4.9, 5.1, 5.2; ICC (Intel C++ Compiler) version 15; and the clang frontend for the LLVM compiler infrastructure versions 3.4 and 3.7. All compilations were done with the flags `-O3 -march=core-avx2`[7] `-fomit-frame-pointer`. For the sake of comparison, we reported our timings for all of the previously mentioned compilers. However, when comparing our code with the state-of-the-art works, we opted for the clang/llvm 3.4, since it gave us the best performance.

Parameters Given $q = 2^m$, with $m = 149$, we constructed our base binary field $\mathbb{F}_q \cong \mathbb{F}_2[x]/(f(x))$ with the 69-term irreducible polynomial $f(x)$ described in Section 4.2.3. The quadratic extension $\mathbb{F}_{q^2} \cong \mathbb{F}_q[u]/(h(u))$ was built through the irreducible quadratic $h(u) = u^2+u+1$. However, our base field arithmetic was computed modulo the redundant trinomial $g(x) = x^{192}+x^{19}+1$, which has among its irreducible factors, the polynomial $f(x)$.

Our Koblitz curve was defined over $\mathbb{F}_{q^2}$ as

$$E_1/\mathbb{F}_{q^2} : y^2 + xy = x^3 + ux^2 + u,$$

and the group $E_1(\mathbb{F}_{q^2})$ contains a subgroup of interest of order

$$r = \texttt{0x637845F7F8BFAB325B85412FB54061F148B7F6E79AE11CC843ADE1470F7E4E29},$$

which corresponds to approximately 255 bits. In addition, our scalar multiplication was computed using a base point P represented in lambda coordinates as

$$\begin{aligned} x_P &= \texttt{0x1B0CB55BC0B41C3EC1820E4E24EBC310451476} \\ &\quad + \texttt{0x4649A2FF1A1B8BA00AA8A706C04D6D97DF60C} \cdot u, \\ \lambda_P &= \texttt{0x6B64DFA496D1DEEA880545B44AC9CC4950C1C} \\ &\quad + \texttt{0x1ADB1DA167DBDF597F03D9A0889FF76FB0B2A1} \cdot u. \end{aligned}$$

Field and elliptic curve arithmetic timings In Table 4.7, we present the timings for the base and the quadratic field arithmetic. The multisquaring operation is used to support the Itoh-Tsujii addition chain, therefore, is implemented only in $\mathbb{F}_{2^{149}}$ (actually, in a 192-bit polynomial in $\mathbb{F}_2[x]$). In addition, we gave timings to reduce a 192-bit polynomial element in $\mathbb{F}_2[x]$ modulo $f(x)$. Finally, all timings of operations in the quadratic field include the subsequent modular reduction.

[7]For the ICC, instead of using `-march-core-avx2`, we used `-xCORE-AVX2`.

Table 4.7: A comparison of the base arithmetic timings (in clock cycles) between different compiler families

Compilers	**Multiplication**	**Squaring**	**Multisquaring**	**Inversion**	**Reduction modulo** $f(x)$
GCC 4.7	68	20	136	2,184	444
GCC 4.8	56	20	176	2,376	452
GCC 4.9	56	20	168	2,388	432
GCC 5.1	52	20	188	2,396	452
GCC 5.2	52	20	184	2,396	452
ICC 15	60	20	116	2,076	416
clang 3.4	60	20	100	1,928	460
clang 3.7	60	24	100	1,916	456

Applying the techniques presented in [123], we saw that our machine has a margin of error of four cycles. This range is not of significance when considering the timings of the point arithmetic or the scalar multiplication; however, for inexpensive functions such as multiplication and squaring, it is recommended to consider it when comparing the timings between different compilers.

Interestingly, the *clang 3.4* compiler does not perform efficiently either in the multiplication function or in the reduction modulo $f(x)$. However, the latter is used only once throughout the scalar multiplication. Also, the next timings show that the *clang* compiler processes the multiplication more efficiently when it is integrated into a more complex arithmetic function. Next, we compare in Table 4.8 the base arithmetic operation timings with the multiplication function, which is the main operation of our library.

Table 4.8: The relation between the timings of the base arithmetic and the multiplication function. The timings were taken from the code compiled with the clang 3.4

Operations	**Squaring**	**Multisquaring**	**Inversion**	**Reduction modulo** $f(x)$
operation / multiplication ratio	0.33	1.66	32.13	7.60

The ratio squaring/multiplication is more expensive than the one in the GLS curve implementation (see Chapter 3). This is because $g(x) = x^{192} + x^{19} + 1$ does not allow a reduction specially designed for the squaring operation. Furthermore, the multisquaring and the inversion functions are relatively more costly than the GLS curve work. A possible explanation is that here, we are measuring timings in a Haswell architecture, which has a computationally cheaper carry-less multiplication when compared with the Sandy Bridge platform [130].

In Table 4.9 we give the timings of the point arithmetic functions. There, we presented the costs of applying the τ endomorphism to an affine point (two coordinates) and a λ-projective point (three coordinates). The reason is that, depending on the scalar multiplication algorithm, one can apply the Frobenius map on the accumulator (projective) or the base point (affine). In addition, we included, in the following table, the mixed-doubling operation. Given a point $P = (x_P, y_P)$, the mixed-doubling function performs $R = (X_R, L_R, Z_R) = 2P$. In other words, it performs a point doubling on an affine point and returns the point in the projective representation. Such function is used in the computation of the τNAF representations $\alpha_u = u \pmod{\tau^w}$ (see Section 4.2.1).

Table 4.9: A comparison of the point arithmetic timings (in clock cycles) between different compiler families

Compilers	**Full Addition**	**Mixed Addition**	**Full Doubling**	**Mixed Doubling**	τ **endomorphism**	
					2 coord.	3 coord.
GCC 4.7	828	624	420	180	92	136
GCC 4.8	816	608	380	144	80	120
GCC 4.9	792	616	376	148	84	124
GCC 5.1	796	592	368	148	80	120
GCC 5.2	796	592	368	148	80	120
ICC 15	780	604	364	148	84	124
clang 3.4	772	568	400	168	84	112
clang 3.7	760	580	404	168	84	124

Table 4.9 also shows the predominance of the *clang* compiler in the point arithmetic timings, since the only operations which it has a clear disadvantage are the full and mixed point doubling. However, those functions are rarely used throughout a Koblitz curve scalar multiplication. More precisely, they are used only in the precomputing phase.

In the left-to-right scalar multiplication approach, the most performed functions are the mixed addition, whose *clang* timings outperformed the GCC family by 4% and the ICC compiler by 6%, and the τ endomorphism applied on three coordinates. Here, the clang code is 6% faster than the GCC family and almost 10% more efficient than the ICC. Next, in Table 4.10, we show the relation of the point arithmetic timings with the field multiplication.

Table 4.10: The relation between the point arithmetic timings and the multiplication function. The timings were taken from the code compiled with the *clang 3.4* compiler

Operations	**Full Addition**	**Mixed Addition**	**Full Doubling**	**Mixed Doubling**	**τ endomorphism**	
					2 coord.	3 coord.
operation / multiplication ratio	12.86	9.46	6.66	2.80	1.40	1.86

Scalar multiplication timings In this part, we present timings for the left-to-right regular w-τNAF τ-and-add scalar multiplication, with $w = 2, 3$. The setting $w = 2$ is presented in order to analyze how the balance between the pre-computation and the main iteration costs works in practice. Our main result lies in the setting $w = 3$. Also, among the scalar multiplication timings, we show, in Table 4.11, the costs of the regular recoding (see Section 4.1) and the linear pass functions.

Table 4.11: A comparison of the scalar multiplication and its support functions timings (in clock cycles) between different compiler families

Compilers	**Regular recoding**		**Linear pass**		**Scalar multiplication**	
	2-τNAF	**3-τNAF**	**2-τNAF**	**3-τNAF**	**2-τNAF**	**3-τNAF**
GCC 4.7	1,696	2,652	20	76	103,400	73,468
GCC 4.8	1,724	2,628	20	64	102,036	73,012
GCC 4.9	1,688	2,744	20	68	100,892	72,180
GCC 5.1	1,684	2,728	16	64	100,560	71,868
GCC 5.2	1,676	2,728	16	64	100,504	71,844
ICC 15	1,992	3,272	16	72	102,516	73,436
clang 3.4	1,828	2,680	?	?	96,822	69,656
clang 3.7	1,860	2,748	?	?	97,240	69,860

Regarding the regular recoding function, we saw an increase of about 46% in the 3-τNAF timings when comparing with the $w = 2$ case. The reason is that, for the $w = 3$ case, we must compute a more complicated arithmetic. Also, when selecting the digits, we must perform a linear pass in the array that stores them. Otherwise, an attacker could learn about the scalar k by performing a timing-attack based on the CPU cache.

The linear pass function also becomes more expensive in the $w = 3$ case, since we have more points in the array. However, in the $m = 149$ case, we have to process 64 more iterations with the width $w = 2$, when comparing it with the 3-τNAF point multiplication. As a result, the linear pass function overhead is smaller than the savings in mixed additions and τ endomorphisms applications.

Furthermore, the *clang* scalar multiplication timings are significantly better than the other compilers. For the $w = 2$ case, it outperforms the GCC family by 3682 and the ICC by 5694 clock cycles. In the $w = 3$ point multiplication, the clang code is 2188 and 3780 cycles faster than the GCC family and the ICC compiler, respectively.

Also, the question marks in Table 4.11 means that the linear passes could not be measured in the *clang compilers*. Somehow it knows that we are performing dummy operations in order to measure the code and optimizes it by just avoiding the execution of the code. Finally, our scalar multiplication measurements consider that the point $Q = kP$ is returned in the λ-projective coordinate representation. If the affine representation is required, it is necessary to add 2000 cycles to the total scalar multiplication timings.

Comparisons In Table 4.12, we compare our implementation with the state-of-the-art works. Our 3-τNAF left-to-right τ-and-add point multiplication outperformed by 29.64% the work in [118], which is considered the fastest protected 128-bit secure Koblitz implementation. When compared with prime curves, our work is surpassed by 15.29% and 21.91% by the works in [43] and [17], respectively.

As a future work, we intend to implement the case $w = 4$, which would likely be compatible with the state-of-the-art prime curves. Considering that the regular recoding overhead does not change from the case $w = 3$, and that the linear pass would take about 100 cc, we expect that, without any optimization, a 4-τNAF left-to-right τ-and-add point multiplication would take about 62,000 cc.

Moreover, we can optimize our code with larger AVX 256-bit instructions. In addition, our implementation is supposed to be more efficient with the Broadwell architecture, which provides faster carry-less multipliers. Finally, we would like to design a version of our point multiplication in the multi-core and known point scenarios.

Table 4.12: Scalar multiplication timings (in clock cycles) on 128-bit secure ellitpic curves

Curve/Method	**TAR**[1]	**Architecture**	**Timings**
Koblitz over $\mathbb{F}_{2^{283}}$ (NIST K-283) τ-and-add, 5-τNAF [118]	yes	Haswell	99,000
Twisted Edwards over $\mathbb{F}_{(2^{127}-1)^2}$ double-and-add [43]	yes	Haswell	59,000
Kummer genus-2 over $\mathbb{F}_{2^{127}-1}$ Kummer ladder [17]	yes	Haswell	54,389
Koblitz over $\mathbb{F}_{4^{149}}$ τ-and-add, 2-τNAF (our work)	yes	Haswell	96,822
Koblitz over $\mathbb{F}_{4^{149}}$ τ-and-add, 3-τNAF (our work)	yes	Haswell	69,656

[1] Timing-attack resistant.

4.3 Summary

At first, we adapted the Joye-Tunstall recoding method to generate, for the first time, a timing-resistant τ-and-add scalar multiplication on Koblitz curves. This implementation outperformed by 26% the protected Koblitz point multiplication based on the Montgomery ladder algorithm.

Next, we designed completely novel 128-bit secure scalar multiplication algorithms on Koblitz curves defined over $\mathbb{F}_4$. In order to achieve that, we implemented a fast base and quadratic field arithmetic, which took advantage of the redundant trinomials method and allowed an efficient modular reduction.

Part II

The Discrete Logarithm Problem

5 | Finite Fields

Let $\mathbb{F}_Q$ denote the finite field of order Q. The discrete logarithm problem (DLP) in $\mathbb{F}_Q$ is that of determining, given a generator g of $\mathbb{F}_Q^*$ and an element $h \in \mathbb{F}_Q^*$, the integer $x \in [0, Q-2]$ satisfying $h = g^x$. In the remainder of this chapter, we shall assume that the characteristic of $\mathbb{F}_Q$ is 2 or 3.

Until recently, the fastest general-purpose algorithm known for solving the DLP in $\mathbb{F}_Q$ was Coppersmith's 1984 index-calculus algorithm [40] with a running time of $L_Q[\frac{1}{3}, (32/9)^{1/3}] \approx L_Q[\frac{1}{3}, 1.526]$, where as usual $L_Q[\alpha, c]$ with $0 < \alpha < 1$ and $c > 0$ denotes the expression

$$\exp\left((c + o(1))(\log Q)^{\alpha}(\log\log Q)^{1-\alpha}\right)$$

that is subexponential in $\log Q$. In February 2013, Joux [86] presented a new DLP algorithm with a running time of $L_Q[\frac{1}{4} + o(1), c]$ (for some undetermined c) when $Q = q^{2n}$ and $q \approx n$. Shortly thereafter, Barbulescu, Gaudry, Joux and Thomé [13] presented an algorithm with *quasi-polynomial* running time $(\log Q)^{O(\log\log Q)}$ when $Q = q^{2n}$ with $q \approx n$.

These dramatic developments were accompanied by some striking computational results. For example, Göloğlu *et al.* [68] computed logarithms in $\mathbb{F}_{2^{8\cdot 3\cdot 255}} = \mathbb{F}_{2^{6120}}$ in only 750 CPU hours, and Joux [87] computed logarithms in $\mathbb{F}_{2^{8\cdot 3\cdot 257}} = \mathbb{F}_{2^{6168}}$ in only 550 CPU hours. The small computational effort expended in these experiments depends crucially on the special nature of the fields $\mathbb{F}_{2^{6120}}$ and $\mathbb{F}_{2^{6168}}$ — namely that $\mathbb{F}_{2^{6120}}$ is a degree-255 extension of $\mathbb{F}_{2^{8\cdot 3}}$ with $255 = 2^8 - 1$ (a Kummer extension), and $\mathbb{F}_{2^{6168}}$ is a degree-257 extension of $\mathbb{F}_{2^{8\cdot 3}}$ with $257 = 2^8 + 1$ (a twisted Kummer extension). Adj *et al.* [2] presented a concrete analysis of the new algorithms and demonstrated that logarithms in $\mathbb{F}_{3^{6\cdot 509}}$ can be computed in approximately 2^{82} steps, which is considerably less than the 2^{128} steps required by Coppersmith's algorithm. Unlike the aforementioned experimental results, the analysis by Adj *et al.* does not exploit any special properties of the fields $\mathbb{F}_{3^{6\cdot 509}}$ and $\mathbb{F}_{3^{6\cdot 1429}}$.

The purpose of this work is to demonstrate that, with modest computational resources, the new algorithms can be used to solve instances of the discrete logarithm problem that remain beyond the reach of classical algorithms. The first target field is the 1303-bit field $\mathbb{F}_{3^{6\cdot 137}}$; this field does not enjoy any Kummer-like properties. More precisely, we are interested in solving the discrete logarithm problem in the order-r subgroup $\mathcal{G}$ of $\mathbb{F}^*_{3^{6\cdot 137}}$, where $r = (3^{137} - 3^{69} + 1)/7011427850892440647$ is a 155-bit prime. The discrete logarithm problem in this group is of cryptographic interest because the values of the bilinear pairing derived from the supersingular elliptic curve $E : y^2 = x^3 - x + 1$ over $\mathbb{F}_{3^{137}}$ lie in $\mathcal{G}$.[1] Consequently, if logarithms in $\mathcal{G}$ can be computed efficiently then the associated bilinear pairing is rendered cryptographically insecure. Note that since r is a 155-bit prime, Pollard's rho algorithm [126] for computing logarithms in $\mathcal{G}$ is infeasible. Moreover, recent work on computing logarithms in the 809-bit field $\mathbb{F}_{2^{809}}$ [12] suggests that Coppersmith's algorithm is infeasible for computing logarithms in $\mathcal{G}$, whereas recent work on computing logarithms in the 923-bit field $\mathbb{F}_{3^{6\cdot 97}}$ [79] (see also [144]) indicates that computing logarithms in $\mathcal{G}$ using the Joux-Lercier algorithm [88] would be a formidable challenge. In contrast, we show that Joux's algorithm can be used to compute logarithms in $\mathcal{G}$ in just a few days using a small number of CPUs; more precisely, our computation consumed a total of 888 CPU hours. The computational effort expended in our experiment is relatively small, despite the fact that our implementation was done using the computer algebra system Magma V2.20-2 [31] and is far from optimal.

The second target field is the 1551-bit field $\mathbb{F}_{3^{6\cdot 163}}$; this field does not enjoy any Kummer-like properties. More precisely, we are interested in solving the discrete logarithm problem in the order-r subgroup $\mathcal{G}$ of $\mathbb{F}^*_{3^{6\cdot 163}}$, where $r = 3^{163} + 3^{82} + 1$ is a 259-bit prime. The discrete logarithm problem in this group is of cryptographic interest because the values of the bilinear pairing derived from the supersingular elliptic curve $E : y^2 = x^3 - x - 1$ over $\mathbb{F}_{3^{163}}$ lie in $\mathcal{G}$. This bilinear pairing was first considered by Boneh, Lynn and Shacham in their landmark paper on short signature schemes [26]; see also [73]. Furthermore, the bilinear pairing derived from the quadratic twist of E was one of the pairings implemented by Galbraith, Harrison and Soldera [59]. Again, we show that Joux's algorithm can be used to compute logarithms in $\mathcal{G}$ in just a few days using a small number of CPUs; our computation used 1201 CPU hours.

After we had completed the $\mathbb{F}_{3^{6\cdot 137}}$ discrete logarithm computation, Granger, Kleinjung and Zumbrägel [70] presented several practical improvements and refinements of Joux's algorithm. These improvements allowed them to compute logarithms in the 4404-bit field $\mathbb{F}_{2^{12\cdot 367}}$ in approximately 52,240 CPU hours, and drastically low-

[1]We note that the supersingular elliptic curves $y^2 = x^3 - x \pm 1$ over $\mathbb{F}_{3^n}$ have embedding degree 6 and were proposed for cryptographic use in several early papers on pairing-based cryptography [15, 26, 59, 72].

ered the estimated time to compute logarithms in the 4892-bit field $\mathbb{F}_{2^{4\cdot 1223}}$ to 2^{59} modular multiplications. More recently, Joux and Pierrot [89] presented a more efficient algorithm for computing logarithms of factor base elements. The new algorithm was used to compute logarithms in the 3796-bit characteristic-three field $\mathbb{F}_{3^{5\cdot 479}}$ in less than 8600 CPU hours.

Also, we present an analysis of the cost for solving the DLP in the characteristic-three fields $\mathbb{F}_{3^{6\cdot 509}}$ and $\mathbb{F}_{3^{6\cdot 1429}}$. Both fields are used to construct $k = 6$ pairings derived from supersingular elliptic curves $Y^2 = X^3 - X + 1$ and $Y^2 = X^3 - X - 1$ over $\mathbb{F}_{3^\ell}$ considered in [26, 56, 15, 59] and implemented in [14, 72, 122, 6, 77, 24, 34, 22]. Finally, we briefly discuss the practical implications of the quasi-polynomial algorithm (QPA) of Barbulescu *et al.* [13] for solving the DLP in small-characteristic fields.

5.1 Joux's $L[1/4 + o(1)]$ algorithm

Let $\mathbb{F}_{q^{3n}}$ be a finite field where $n \leq 2q + 1$.[2] The elements of $\mathbb{F}_{q^{3n}}$ are represented as polynomials of degree at most $n-1$ over $\mathbb{F}_{q^3}$. Let $N = q^{3n} - 1$, and let r be a prime divisor of N. In this work, we are interested in the discrete logarithm problem in the order-r subgroup of $\mathbb{F}_{q^{3n}}^*$. More precisely, we are given two elements α, β of order r in $\mathbb{F}_{q^{3n}}^*$ and we wish to find $\log_\alpha \beta$. Let g be an element of order N in $\mathbb{F}_{q^{3n}}^*$. Then $\log_\alpha \beta = (\log_g \beta)/(\log_g \alpha) \bmod r$. Thus, in the remainder of this section we will assume that we need to compute $\log_g h \bmod r$, where h is an element of order r in $\mathbb{F}_{q^{3n}}^*$.

The algorithm proceeds by first finding the logarithms (mod r) of all degree-one elements in $\mathbb{F}_{q^{3n}}$ (see Section 5.1.1). Then, in the *descent stage*, $\log_g h$ is expressed as a linear combination of logarithms of degree-one elements. The descent stage proceeds in several steps, each expressing the logarithm of a degree-D element as a linear combination of the logarithms of elements of degree $\leq m$ for some $m < D$. Four descent methods are employed; these are described in Sections 5.1.2 – 5.1.5.

Notation. $N_{q^3}(m, n)$ denotes the number of monic m-smooth degree-n polynomials in $\mathbb{F}_{q^3}[X]$, $A_{q^3}(m, n)$ denotes the average number of distinct monic irreducible factors among all monic m-smooth degree-n polynomials in $\mathbb{F}_{q^3}[X]$, and $S_{q^3}(m, d)$ denotes the cost of testing m-smoothness of a degree-d polynomial in $\mathbb{F}_{q^3}[X]$. Formulas for $N_{q^3}(m, n)$, $A_{q^3}(m, n)$ and $S_{q^3}(m, n)$ are given in [2]. For $\gamma \in \mathbb{F}_{q^3}$, $\overline{\gamma}$ denotes the element γ^{q^2}. For $P \in \mathbb{F}_{q^3}[X]$, $\overline{P}$ denotes the polynomial obtained by raising

[2]More generally, one could consider fields $\mathbb{F}_{q^{kn}}$ where $n \leq 2q + 1$. We focus on the case $k = 3$ since our target fields are $\mathbb{F}_{3^{6n}}$ with $n \in \{137, 163\}$, which we will embed in $\mathbb{F}_{(3^4)^{3\cdot n}}$.

each coefficient of P to the power q^2. The cost of an integer addition modulo r is denoted by A_r, and the cost of a multiplication in $\mathbb{F}_{q^3}$ is denoted by M_{q^3}. The projective general linear group of degree 2 over $\mathbb{F}_q$ is denoted $\mathrm{PGL}_2(\mathbb{F}_q)$. $\mathcal{P}_q$ is a set of distinct representatives of the left cosets of $\mathrm{PGL}_2(\mathbb{F}_q)$ in $\mathrm{PGL}_2(\mathbb{F}_{q^3})$; note that $\#\mathcal{P}_q = q^6 + q^4 + q^2$. A matrix $\left(\begin{smallmatrix} a & b \\ c & d \end{smallmatrix}\right) \in \mathcal{P}_q$ is identified with the quadruple (a, b, c, d).

5.1.1 Setup

Select polynomials $h_0, h_1 \in \mathbb{F}_{q^3}[X]$ of small degree so that

$$X \cdot h_1(X^q) - h_0(X^q) \tag{5.1}$$

has an irreducible factor I_X of degree n in $\mathbb{F}_{q^3}[X]$; we will henceforth assume that $\max(\deg h_0, \deg h_1) = 2$, whence $n \leq 2q + 1$. Note that

$$X \equiv \frac{h_0(X^q)}{h_1(X^q)} \equiv \left(\frac{\overline{h}_0(X)}{\overline{h}_1(X)}\right)^q \pmod{I_X}. \tag{5.2}$$

The field $\mathbb{F}_{q^{3n}}$ is represented as $\mathbb{F}_{q^{3n}} = \mathbb{F}_{q^3}[X]/(I_X)$ and the elements of $\mathbb{F}_{q^{3n}}$ are represented as polynomials in $\mathbb{F}_{q^3}[X]$ of degree at most $n - 1$. Let g be a generator of $\mathbb{F}^*_{q^{3n}}$.

Finding logarithms of linear polynomials

Let $\mathcal{B}_1 = \{X + a \mid a \in \mathbb{F}_{q^3}\}$, and note that $\#\mathcal{B}_1 = q^3$. To compute the logarithms of $\mathcal{B}_1$-elements, we first generate linear relations of these logarithms. Let $(a, b, c, d) \in \mathcal{P}_q$. Substituting $Y \mapsto (aX + b)/(cX + d)$ into the systematic equation

$$Y^q - Y = \prod_{\alpha \in \mathbb{F}_q} (Y - \alpha) \tag{5.3}$$

and using (5.2) yields

$$\begin{aligned} &\Big((aX + b)(\overline{c}\overline{h}_0 + \overline{d}\,\overline{h}_1) - (\overline{a}\overline{h}_0 + \overline{b}\,\overline{h}_1)(cX + d)\Big)^q \\ &\qquad\qquad \equiv \overline{h}_1^q \cdot (cX + d) \cdot \prod_{\alpha \in \mathbb{F}_q} [(a - \alpha c)X + (b - \alpha d)]. \end{aligned} \tag{5.4}$$

If the polynomial on the left side of (5.4) is 1-smooth, then taking logarithms (mod r) of both sides of (5.4) yields a linear relation of the logarithms of $\mathcal{B}_1$-elements and the logarithm of $\overline{h}_1$. The probability that the left side of (5.4) is 1-smooth is $N_{q^3}(1,3)/q^9 \approx \frac{1}{6}$. Thus, after approximately $6q^3$ trials one expects to obtain q^3 relations. The cost of the relation generation stage is $6q^3 \cdot S_{q^3}(1,3)$. The logarithms can then be obtained by using Wiedemann's algorithm for solving sparse systems of linear equations [156, 41]. The expected cost of the linear algebra is $q^7 \cdot A_r$ since each equation has approximately q nonzero terms.

5.1.2 Continued-fractions descent

Recall that we wish to compute $\log_g h \bmod r$, where $h \in \mathbb{F}_{q^{3n}} = \mathbb{F}_{q^3}[X]/(I_X)$ has order r. We will henceforth assume that $\deg h = n-1$. The descent stage begins by multiplying h by a random power of g. The extended Euclidean algorithm is used to express the resulting field element h' in the form $h' = w_1/w_2$ where $\deg w_1, \deg w_2 \approx n/2$ [83]; for simplicity, we shall assume that n is odd and $\deg w_1 = \deg w_2 = (n-1)/2$. This process is repeated until both w_1 and w_2 are m-smooth for some chosen $m < (n-1)/2$. This gives $\log_g h'$ as a linear combination of logarithms of polynomials of degree at most m. The expected cost of this continued-fractions descent step is approximately

$$\left(\frac{(q^3)^{(n-1)/2}}{N_{q^3}(m,(n-1)/2)}\right)^2 \cdot S_{q^3}(m,(n-1)/2). \tag{5.5}$$

The expected number of distinct irreducible factors of w_1 and w_2 is $2A_{q^3}(m,(n-1)/2)$. In the concrete analysis, we shall assume that each of these irreducible factors has degree exactly m. The logarithm of each of these degree-m polynomials is then expressed as a linear combination of logarithms of smaller degree polynomials using one of the descent methods described in Sections 5.1.3, 5.1.4 and 5.1.5.

5.1.3 Classical descent

Let p be the characteristic of $\mathbb{F}_q$, and let $q = p^\ell$. Let $s \in [0,\ell]$, and let $R \in \mathbb{F}_{q^3}[X,Y]$. Then it can be seen that

$$\left[R(X, (\overline{h}_0/\overline{h}_1)^{p^{\ell-s}})\right]^{p^s} \equiv R'(X^{p^s}, X) \pmod{I_X} \tag{5.6}$$

where R' is obtained from R by raising all its coefficients to the power p^s. Let $\mu = \deg_Y R$. Then multiplying both sides of (5.6) by $\overline{h}_1^{q\mu}$ gives

$$\left[\overline{h}_1^{p^{\ell-s}\cdot\mu} \cdot R(X, (\overline{h}_0/\overline{h}_1)^{p^{\ell-s}})\right]^{p^s} \equiv \overline{h}_1^{q\mu} \cdot R'(X^{p^s}, X) \pmod{I_X}. \tag{5.7}$$

Let $Q \in \mathbb{F}_{q^3}[X]$ with $\deg Q = D$, and let $m < D$. In the Joux-Lercier descent method [88], as modified by Göloğlu *et al.* [67], one selects $s \in [0, \ell]$ and searches for a polynomial $R \in \mathbb{F}_{q^3}[X, Y]$ such that (i) $Q \mid R_2$ where $R_2 = R'(X^{p^s}, X)$; (ii) $\deg R_1$ and $\deg R_2/Q$ are appropriately balanced where $R_1 = \overline{h}_1^{p^{\ell-s}\mu} R(X, (\overline{h}_0/\overline{h}_1)^{p^{\ell-s}})$; and (iii) both R_1 and R_2/Q are m-smooth. Taking logarithms of both sides of (5.7) then gives an expression for $\log_g Q$ in terms of the logarithms of polynomials of degree at most m.

A family of polynomials R satisfying (i) and (ii) can be constructed by finding a basis $\{(u_1, u_2), (v_1, v_2)\}$ of the lattice

$$L_Q = \{(w_1, w_2) \in \mathbb{F}_{q^3}[X] \times \mathbb{F}_{q^3}[X] \ : \ Q \mid (w_1(X) - w_2(X)X^{p^s})\}$$

where $\deg u_1$, $\deg u_2$, $\deg v_1$, $\deg v_2 \approx D/2$. By writing $(w_1, w_2) = a(u_1, u_2) + b(v_1, v_2) = (au_1 + bv_1, au_2 + bv_2)$ with $a \in \mathbb{F}_{q^3}[X]$ monic of degree δ and $b \in \mathbb{F}_{q^3}[X]$ of degree $\delta - 1$, the points (w_1, w_2) in L_Q can be sampled to obtain polynomials $R(X, Y) = w_1''(Y) - w_2''(Y)X$ satisfying (i) and (ii) where w'' is obtained from w by raising all its coefficients to the power p^{-s}. The number of lattice points to consider is therefore $(q^3)^{2\delta}$. We have $\deg w_1, \deg w_2 \approx D/2 + \delta$, so $\deg R_1 = t_1 \approx 2(D/2 + \delta)p^{\ell-s} + 1$ and $\deg R_2 = t_2 \approx (D/2 + \delta) + p^s$. In order to ensure that there are sufficiently many such lattice points to generate a polynomial R for which both R_1 and R_2/Q are m-smooth, the parameters s and δ must be selected so that

$$q^{6\delta} \gg \frac{q^{3t_1}}{N_{q^3}(m, t_1)} \cdot \frac{q^{3(t_2-D)}}{N_{q^3}(m, t_2 - D)}. \tag{5.8}$$

Ignoring the time to compute a balanced basis of L_Q, the expected cost of finding a polynomial R satisfying (i)–(iii) is

$$\frac{q^{3t_1}}{N_{q^3}(m, t_1)} \cdot \frac{q^{3(t_2-D)}}{N_{q^3}(m, t_2 - D)} \cdot \min(S_{q^3}(m, t_1), S_{q^3}(m, t_2 - D)). \tag{5.9}$$

The expected number of distinct irreducible factors of R_1 and R_2/Q is $A_{q^3}(m, t_1) + A_{q^3}(m, t_2 - D)$.

5.1.4 Gröbner bases descent

Let $Q \in \mathbb{F}_{q^3}[X]$ with $\deg Q = D$. Let $m = \lceil (D + 1)/2 \rceil$, and suppose that $3m < n$. In Joux's new descent method [86, Section 5.3], one finds degree-m polynomials $k_1, k_2 \in \mathbb{F}_{q^3}[X]$ such that $G = k_1\widetilde{k}_2 - \widetilde{k}_1 k_2 = QR$, where $\widetilde{k}_1 = \overline{h}_1^m \overline{k}_1(\overline{h}_0/\overline{h}_1)$ and

$\widetilde{k}_2 = \overline{h}_1^m \overline{k}_2(\overline{h}_0/\overline{h}_1)$, and $R \in \mathbb{F}_{q^3}[X]$. Note that $\deg R = 3m - D$. If R is m-smooth, then we obtain a linear relationship between $\log_g Q$ and logs of degree-m polynomials (see [4, Section 3.7]):

$$\overline{h}_1^{mq} \cdot k_2 \cdot \prod_{\alpha \in \mathbb{F}_q} (k_1 - \alpha k_2) \equiv (Q(X)R(X))^q \pmod{I_X}. \tag{5.10}$$

To determine (k_1, k_2, R) that satisfy

$$k_1 \widetilde{k}_2 - \widetilde{k}_1 k_2 = QR, \tag{5.11}$$

one can transform (5.11) into a system of multivariate quadratic equations over $\mathbb{F}_q$. Specifically, each coefficient of k_1, k_2 and R is written using three variables over $\mathbb{F}_q$. The coefficients of $\widetilde{k}_1$ and $\widetilde{k}_2$ can then be written in terms of the coefficients of k_1 and k_2. Hence, equating coefficients of X^i of both sides of (5.11) yields $3m+1$ quadratic equations. Equating $\mathbb{F}_q$-components of these equations then yields $9m+3$ bilinear equations in $15m - 3D + 9$ variables over $\mathbb{F}_q$. This system of equations can be solved by finding a Gröbner basis for the ideal it generates. Finally, solutions (k_1, k_2, R) are tested until one is found for which R is m-smooth. This yields an expression for $\log_g Q$ in terms of the logarithms of approximately $q + 1 + A_{q^3}(m, 3m - D)$ polynomials of degree (at most) m; in the concrete analysis we shall assume that each of the polynomials has degree exactly m.

5.1.5 2-to-1 descent.

The Gröbner bases descent methodology of §2.5 can be employed in the case $(D, m) = (2, 1)$. However, as also reported by Joux in his $\mathbb{F}_{2^{6168}}$ discrete log computation [87], we found the descent to be successful for only about 50% of all irreducible quadratic polynomials. Despite this, some strategies can be used to increase this percentage.

Let $Q(X) = X^2 + uX + v \in \mathbb{F}_{q^3}[X]$ be an irreducible quadratic polynomial for which the Gröbner bases descent method failed.

Strategy 1. Introduced by Joux [87] and Göloğlu *et al.* [68], this strategy is based on the systematic equation derived from $Y^{q'} - Y$ where $q' < q$ and $\mathbb{F}_{q'}$ is a proper subfield of $\mathbb{F}_{q^3}$ instead of the systematic equation (5.3) derived from $Y^q - Y$. Let p be the characteristic of $\mathbb{F}_q$, and let $q = p^\ell$, $q' = p^{\ell'}$, and $s = \ell - \ell'$. Then $q = p^s \cdot q'$. Now, one searches for $a, b, c, d \in \mathbb{F}_{q^3}$ such that

$$G = (aX + b)(\overline{c}\overline{h}_0 + \overline{d}\,\overline{h}_1)^{p^s} - (\overline{a}\overline{h}_0 + \overline{b}\,\overline{h}_1)^{p^s}(cX + d) = QR$$

with $R \in \mathbb{F}_{q^3}[X]$. Note that $\deg R = 2p^s - 1$.[3] If R is 1-smooth, then we obtain a linear relationship between $\log_g Q$ and logs of linear polynomials since

$$G^q \equiv \overline{h}_1^{p^s q} \cdot (cX+d)^{p^s} \cdot \prod_{\alpha \in \mathbb{F}_{q'}} \left((aX+b)^{p^s} - \alpha(cX+d)^{p^s} \right) \pmod{I_X},$$

as can be seen by making the substitution $Y \mapsto (aX+b)^{p^s}/(cX+d)^{p^s}$ into the systematic equation derived from $Y^{q'} - Y$.

Unfortunately, in all instances we considered, the polynomial R never factors completely into linear polynomials. However, it hopefully factors into a quadratic polynomial Q' and $2p^s - 3$ linear polynomials, thereby yielding a relation between Q and another quadratic which has a roughly 50% chance of descending using Gröbner bases descent. Combined with the latter, this strategy descends about 95% of all irreducible quadratic polynomials in the fields $\mathbb{F}_{3^{6\cdot 137}}$ and $\mathbb{F}_{3^{6\cdot 163}}$.

Strategy 2. We have

$$\begin{aligned} \overline{h}_1^{2q} Q(X) \equiv \overline{h}_1^{2q} Q((\overline{h}_0/\overline{h}_1)^q) &= \overline{h}_0^{2q} + u\overline{h}_0^q \overline{h}_1^q + v\overline{h}_1^{2q} \\ &= (\overline{h}_0^2 + \overline{u}\overline{h}_0\overline{h}_1 + \overline{v}\,\overline{h}_1^2)^q \pmod{I_X}. \end{aligned} \tag{5.12}$$

It can be seen that the degree-4 polynomial $f_Q(X) = \overline{h}_0^2 + \overline{u}\overline{h}_0\overline{h}_1 + \overline{v}\,\overline{h}_1^2$ is either a product of two irreducible quadratics or itself irreducible. In the former case, we apply the standard Gröbner bases descent method to the two irreducible quadratics. If both descents are successful, then we have succeeded in descending the original Q.

The strategies are combined in the following manner. For an irreducible quadratic $Q \in \mathbb{F}_{q^3}[X]$, we first check if the Gröbner bases descent is successful. If the descent fails, we apply Strategy 2 to Q. In the case where f_Q factors into two irreducible quadratics, and at least one of them fails to descent with Gröbner bases descent, we apply Strategy 1 to Q. If Strategy 1 fails on Q, we apply it to the two quadratic factors of f_Q. In the case where f_Q is irreducible, we apply Strategy 1 to Q.

If none of the attempts succeed, we declare Q to be "bad", and avoid it in the higher-degree descent steps by repeating a step until all the quadratics encountered are "good". In our experiments with $\mathbb{F}_{3^{6\cdot 137}}$ and $\mathbb{F}_{3^{6\cdot 163}}$, we observed that approximately 97.2% of all irreducible quadratic polynomials Q were "good".

[3]For our $\mathbb{F}_{3^{6\cdot 137}}$ and $\mathbb{F}_{3^{6\cdot 163}}$ computations, we have $q = 3^4$ and used $q' = 3^3$, so $s = 1$ and $\deg R = 5$.

To see that this percentage is sufficient to complete the descent phase in these two fields, consider a 3-to-2 descent step where the number of resulting irreducible quadratic polynomials is 42 on average (cf. equation (5.10)). Then the probability of descending a degree-3 polynomial after finding one useful solution (k_1, k_2, R) in Gröbner bases descent is $0.972^{42} \approx 0.3$. Therefore, after at most four trials we expect to successfully descend a degree-3 polynomial. Since the expected number of distinct solutions of (5.11) is approximately q^3 (according to equation (10) in [70]), one can afford this many trials.

5.2 Computing discrete logarithms in $\mathbb{F}_{3^{6\cdot 137}}$

The supersingular elliptic curve $E : y^2 = x^3 - x + 1$ has order $\#E(\mathbb{F}_{3^{137}}) = cr$, where

$$c = 7 \cdot 4111 \cdot 5729341 \cdot 42526171$$

and

$$r = (3^{137} - 3^{69} + 1)/c = 33098280119090191028775580055082175056428495623$$

is a 155-bit prime [23]. The Weil and Tate pairing attacks [112, 55] efficiently reduce the discrete logarithm problem in the order-r subgroup $\mathcal{E}$ of $E(\mathbb{F}_{3^{137}})$ to the discrete logarithm problem in the order-r subgroup $\mathcal{G}$ of $\mathbb{F}^*_{3^{6\cdot 137}}$.

Our approach to computing logarithms in $\mathcal{G}$ is to use Joux's algorithm to compute logarithms in the quadratic extension $\mathbb{F}_{3^{12\cdot 137}}$ of $\mathbb{F}_{3^{6\cdot 137}}$ (so $q = 3^4$ and $n = 137$ in the notation of Section 5.1). More precisely, we are given two elements α, β of order r in $\mathbb{F}^*_{3^{12\cdot 137}}$ and we wish to find $\log_\alpha \beta$. Let g be a generator of $\mathbb{F}^*_{3^{12\cdot 137}}$. Then $\log_\alpha \beta = (\log_g \beta)/(\log_g \alpha) \bmod r$. Thus, in the remainder of the section we will assume that we need to compute $\log_g h \bmod r$, where h is an element of order r in $\mathbb{F}^*_{3^{12\cdot 137}}$.

The DLP instance we solved is described in Section 5.2.1. The concrete estimates from Section 5.1 for solving the DLP instances are given in Section 5.2.2. These estimates are only upper bounds on the running time of the algorithm. Nevertheless, they provide convincing evidence for the feasibility of the discrete logarithm computations. Our experimental results are presented in Section 5.2.3.

5.2.1 Problem instance

Let N denote the order of $\mathbb{F}^*_{3^{12\cdot 137}}$. Using the tables from the Cunningham Project [134], we determined that the factorization of N is $N = p_1^4 \cdot \prod_{i=2}^{31} p_i$, where the p_i are the following primes (and $r = p_{25}$):

$$
\begin{aligned}
&p_1 = 2 \quad p_2 = 5 \quad p_3 = 7 \quad p_4 = 13 \quad p_5 = 73 \quad p_6 = 823 \quad p_7 = 4111 \quad p_8 = 4933 \\
&p_9 = 236737 \quad p_{10} = 344693 \quad p_{11} = 2115829 \quad p_{12} = 5729341 \quad p_{13} = 42526171 \\
&p_{14} = 217629707 \quad p_{15} = 634432753 \quad p_{16} = 685934341 \quad p_{17} = 82093596209179 \\
&p_{18} = 4354414202063707 \quad p_{19} = 18329390240606021 \quad p_{20} = 46249052722878623693 \\
&p_{21} = 201820452878622271249 \quad p_{22} = 113938829134880224954142892526477 \\
&p_{23} = 51854546646328186791017417700430486396513 \\
&p_{24} = 273537065683369412556888964042827802376371 \\
&p_{25} = 33098280119090191028775580055082175056428495623 \\
&p_{26} = 706712258201940254667826642673008768387229115048379 \\
&p_{27} = 108081809773839995188256800499141543684393035450350551 \\
&p_{28} = 91321974595662761339222271626247966116126450162880692588587183952237 \\
&p_{29} = 39487531149773489532096996293368370182957526257988573877031054477249 \\
&\qquad 393549 \\
&p_{30} = 40189860022384850044254854796561182547553072730738823866986300807613 \\
&\qquad 29207749418522920289 \\
&p_{31} = 19064323153825272072803685870803955622834286523139037403580752310822 \\
&\qquad 7896644646984063736942624066227406898132113366226593158464419713.
\end{aligned}
$$

We chose $\mathbb{F}_{3^4} = \mathbb{F}_3[U]/(U^4 + U^2 + 2)$ and $\mathbb{F}_{3^{12}} = \mathbb{F}_{3^4}[V]/(V^3 + V + U^2 + U)$, and selected $h_0(X) = V^{326196}X^2 + V^{35305}X + V^{204091} \in \mathbb{F}_{3^{12}}[X]$ and $h_1 = 1$. Then $I_X \in \mathbb{F}_{3^{12}}[X]$ is the degree-137 monic irreducible factor of $X - h_0(X^{3^4})$; the other irreducible factor has degree 25.

We chose the generator $g = X + V^{113713}$ of $\mathbb{F}^*_{3^{12 \cdot 137}}$. To generate an order-$r$ discrete logarithm challenge h, we computed

$$h' = \sum_{i=0}^{136} \left(V^{\lfloor \pi \cdot (3^{12})^{i+1} \rfloor \bmod 3^{12}} \right) X^i$$

and then set $h = (h')^{N/r}$. The discrete logarithm $\log_g h \bmod r$ was found to be

$$x = 27339619076975093920245515973214186963025656559.$$

This can be verified by checking that $h = (g^{N/r})^y$, where $y = x \cdot (N/r)^{-1} \bmod r$.

5.2.2 Estimates

The factor base $\mathcal{B}_1$ has size $3^{12} \approx 2^{19}$. The cost of the relation generation is approximately $2^{29.2}M_{q^3}$, whereas the cost of the linear algebra is approximately $2^{44.4}A_r$. Figure 5.1 shows the estimated running times for the descent stage. Further information about the parameter choices are provided below.

1. For the continued-fractions descent stage, we selected $m = 13$. The expected cost of this descent is $2^{43.2}M_{q^3}$, and the expected number of irreducible factors of degree (at most) 13 obtained is $2A_{3^{12}}(68, 13) \approx 20$.

2. Two classical descent stages are employed. In the first stage, we have $D = 13$ and select $m = 7$, $s = 3$, $\delta = 1$, which yield $t_1 = 43$ and $t_2 = 34$. The expected cost of the descent for each of the 20 degree-13 polynomials is approximately $2^{33.7}M_{q^3}$. The expected total number of distinct irreducible polynomials of degree (at most) 7 obtained is approximately 320.

   In the second classical descent stage, we have $D = 7$ and select $m = 5$, $s = 3$, $\delta = 1$, which yield $t_1 = 25$ and $t_2 = 31$. The expected cost of the descent for each of the 320 degree-7 polynomials is approximately $2^{34.8}M_{q^3}$. The expected total number of distinct irreducible polynomials of degree (at most) 5 obtained is approximately $5,120$.

3. Our implementation of the Gröbner bases descent stage used Magma's implementation of Faugére's F4 algorithm [48] and took 26.5 minutes on average for a 5-to-3 descent, 34.7 seconds for a 3-to-2 descent, and 0.216 seconds for a 2-to-1 descent. The total expected running time for each of these stages is 94, 211 and 168 days, respectively.

Since all the descent stages can be effectively parallelized, our estimates suggest that a discrete logarithm can be computed in a week or so given a few dozen processors. In fact (and as confirmed by our experimental results), the actual running time is expected to be significantly less than the estimated running time since the estimates are quite conservative; for example, our estimates for the number of branches in a descent step assumes that each distinct irreducible polynomial has degree exactly m, whereas in practice many of these polynomials will have degree significantly less than m.

5.2.3 Experimental results

Our experiments were run on an Intel i7-2600K 3.40 GHz machine (Sandy Bridge), and on an Intel i7-4700MQ 2.40 GHz machine (Haswell).

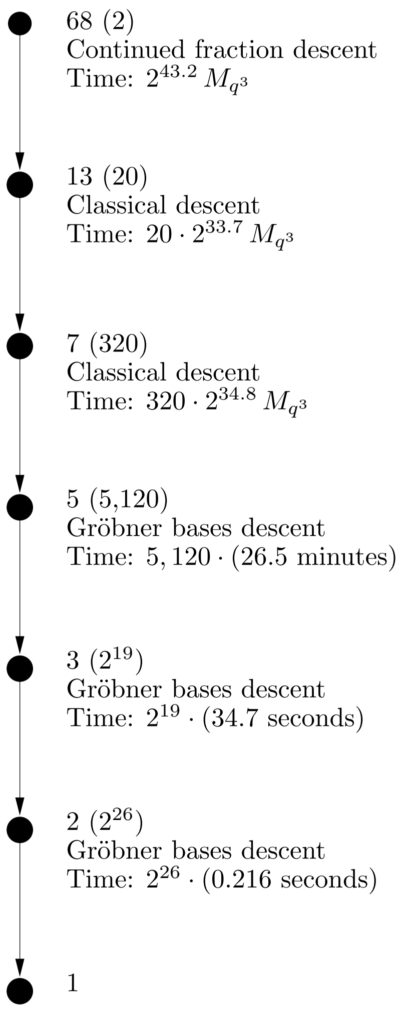
68 (2)
Continued fraction descent
Time: $2^{43.2}\,M_{q^3}$
13 (20)
Classical descent
Time: $20 \cdot 2^{33.7}\,M_{q^3}$
7 (320)
Classical descent
Time: $320 \cdot 2^{34.8}\,M_{q^3}$
5 (5,120)
Gröbner bases descent
Time: $5,120 \cdot$ (26.5 minutes)
3 (2^{19})
Gröbner bases descent
Time: $2^{19} \cdot$ (34.7 seconds)
2 (2^{26})
Gröbner bases descent
Time: $2^{26} \cdot$ (0.216 seconds)
1

Figure 5.1: A typical path of the descent tree for computing an individual logarithm in $\mathbb{F}_{3^{12\cdot 137}}$ ($q = 3^4$). The numbers in parentheses next to each node are the expected number of nodes at that level. 'Time' is the expected time to generate all nodes at a level.

Relation generation took 1.05 CPU hours (Sandy Bridge, 1 core). The resulting sparse linear system of linear equation was solved using Magma's multi-threaded parallel version of the Lanczos algorithm; the computation took 556.8 CPU hours (Sandy Bridge, 4 cores).

In the continued-fractions descent stage, the first degree-68 polynomial yielded 9 irreducible factors of degrees 12, 12, 11, 10, 8, 6, 6, 2, 1, and the second degree-68 polynomial yielded 11 irreducible factors of degrees 13, 12, 10, 10, 7, 6, 5, 2, 1, 1, 1. The computation took 22 CPU hours (Haswell, 4 cores).

Classical descent was used on the 9 polynomials of degree ≥ 8 to obtain polynomials of degree ≤ 7, and then on the 23 polynomials of degree 7 and 23 polynomials of degree 6 to obtain polynomials of degree ≤ 5. These computations took 80 CPU hours (Haswell, 4 cores).

Finally, we used 5-to-3, 4-to-3, 3-to-2 and 2-to-1 Gröbner bases descent procedures. The average time for a 4-to-3 descent was 33.8 seconds; the other average times are given in Figure 5.1. In total, we performed 233 5-to-3 descents, 174 4-to-3 descents, and 11573 3-to-2 descents. These computations took 115.2 CPU hours, 1.5 CPU hours, and 111.2 CPU hours, respectively (Haswell, 4 cores). We also performed 493537 2-to-1 descents; their running times are incorporated into the running times for the higher-level descents.

5.3 Computing discrete logarithms in $\mathbb{F}_{3^{6\cdot 163}}$

The supersingular elliptic curve $E : y^2 = x^3 - x - 1$ has order $\#E(\mathbb{F}_{3^{163}}) = 3^{163} + 3^{82} + 1 = r$, where r is the following 259-bit prime:

$$r = 589881151426658740854227725580736348850640632297373414091790995505756\\623268837.$$

The Weil and Tate pairing attacks [112, 55] efficiently reduce the discrete logarithm problem in the order-r group $\mathcal{E} = E(\mathbb{F}_{3^{163}})$ to the discrete logarithm problem in the order-r subgroup $\mathcal{G}$ of $\mathbb{F}^*_{3^{6\cdot 163}}$.

As in Section 5.2, we will compute logarithms in $\mathcal{G}$ by using Joux's algorithm to compute logarithms in the quadratic extension $\mathbb{F}_{3^{12\cdot 163}}$ of $\mathbb{F}_{3^{6\cdot 163}}$ (so $q = 3^4$ and $n = 163$ in the notation of Section 5.1). We will compute $\log_g h \bmod r$, where g is a generator of $\mathbb{F}^*_{3^{12\cdot 163}}$ and h is an element of order r in $\mathbb{F}^*_{3^{12\cdot 163}}$.

5.3.1 Problem instance

Let N denote the order of $\mathbb{F}^*_{3^{12\cdot163}}$. Using the tables from the Cunningham Project [134], we partially factored N as $N = C \cdot p_1^4 \cdot \prod_{i=2}^{22} p_i$, where the p_i are the following primes (and $r = p_{20}$):

$$
\begin{aligned}
&p_1 = 2 \quad p_2 = 5 \quad p_3 = 7 \quad p_4 = 13 \quad p_5 = 73 \quad p_6 = 653 \quad p_7 = 50857\\
&p_8 = 107581 \quad p_9 = 489001 \quad p_{10} = 105451873 \quad p_{11} = 380998157\\
&p_{12} = 8483499631 \quad p_{13} = 5227348213873 \quad p_{14} = 8882811705390167\\
&p_{15} = 4956470591980320134353 \quad p_{16} = 2321081703582927570592 9\\
&p_{17} = 3507171060957186767994912136200333814689659449\\
&p_{18} = 6351885141964057411259499526611848626072045955243\\
&p_{19} = 842687359180941058363182465115337641211400104811307410674430711031148\\
&\qquad 817701717\\
&p_{20} = 5898811514266587408542277255807363488506406322973734140917909955057 5\\
&\qquad 6623268837\\
&p_{21} = 1326290578404372337003402566761812108154043828317726868004518688485 3\\
&\qquad 2620412724278105428771691382890569577153531961762590484982180238880 1\\
&p_{22} = 2487998472767501120519871818305554760112258297437457690889886964157 0\\
&\qquad 0926912242398570439592596492295941044800988653984249495592713645064 3\\
&\qquad 31019158574269,
\end{aligned}
$$

and C is the following 919-bit composite number

$$
\begin{aligned}
C = {} &287332203665612050739450194991228343672298354626595155150763295732576 7\\
&027521632874777379256652372965509784810211348879569893676839449499262 1\\
&231202281901101934095762050200004569108166947564891990134699175198145 0\\
&83115345709455585222288272983378262150437440948615147544541514931 77.
\end{aligned}
$$

We verified that $\gcd(C, N/C) = 1$ and that C is not divisible by any of the first 10^7 primes. Consequently, if an element g is selected uniformly at random from $\mathbb{F}^*_{3^{12\cdot163}}$, and g satisfies $g^{(N-1)/p_i} \neq 1$ for $1 \le i \le 22$, then g is a generator with very high probability.[4]

We chose $\mathbb{F}_{3^4} = \mathbb{F}_3[U]/(U^4 + U^2 + 2)$ and $\mathbb{F}_{3^{12}} = \mathbb{F}_{3^4}[V]/(V^3 + V + U^2 + U)$, and selected $h_0(X) = 1$ and

$$h_1(X) = X^2 + V^{530855} \in \mathbb{F}_{3^{12}}[X].$$

[4]More precisely, since C has at most 34 prime factors, each of which is greater than the prime $p = 179424673$, the probability that g is a generator is at least $(1 - \frac{1}{p})^{34} > 0.99999981$.

Then $I_X \in \mathbb{F}_{3^{12}}[X]$ is the degree-163 irreducible polynomial $X \cdot h_1(X^{3^4}) - 1$:

$$I_X = X^{163} + V^{530855}X + 2.$$

We chose $g = X + V^2$, which we hope is a generator of $\mathbb{F}^*_{3^{12\cdot 163}}$.

To generate an order-r discrete logarithm challenge h, we computed

$$h' = \sum_{i=0}^{162} \left(V^{\lfloor \pi \cdot (3^{12})^{i+1} \rfloor \bmod 3^{12}} \right) X^i$$

and then set $h = (h')^{N/r}$. The discrete logarithm $\log_g h \bmod r$ was found to be

$$x = 42639595149827919371329139195344900073259255425113252567203978435605 \\ 4526194343.$$

This can be verified by checking that $h = (g^{N/r})^y$, where $y = x \cdot (N/r)^{-1} \bmod r$.

5.3.2 Experimental results

Our experiments were run on an Intel i7-2600K 3.40 GHz machine (Sandy Bridge), and on an Intel Xeon E5-2650 2.00 GHz machine (Sandy Bridge-EP). The descent strategy was similar to the one used for the $\mathbb{F}_{3^{6\cdot 137}}$ computation.

Relation generation took 0.84 CPU hours (Sandy Bridge, 1 core). The resulting sparse system of linear equations was solved using Magma's multi-threaded parallel version of the Lanczos algorithm; the computation took 852.5 CPU hours (Sandy Bridge, 4 cores).

In the continued-fractions descent stage with $m = 15$, the first degree-81 polynomial yielded 8 irreducible factors of degrees 15, 15, 14, 14, 10, 7, 5, 1, and the second degree-81 polynomial yielded 12 irreducible factors of degrees 12, 10, 9, 9, 9, 8, 6, 6, 6, 4, 1, 1. The computation took 226.7 CPU hours (Sandy Bridge-EP, 16 cores).

Classical descent was used on the 11 polynomials of degree ≥ 8 to obtain polynomials of degree ≤ 7, and then a variant of classical descent (called the "alternative" method in Section 3.5 of [4]) was used on the 15 polynomials of degree 7 and 30 polynomials of degree 6 to obtain polynomials of degree ≤ 5. These computations took 51.0 CPU hours (Sandy Bridge-EP, 16 cores).

Finally, we used 5-to-3, 4-to-3 and 3-to-2 Gröbner bases descent procedures. The descent was sped up by writing the coefficients of R (cf. equation (5.11)) in terms of the coefficients of k_1 and k_2; this reduced the number of variables in the resulting bilinear equations from $15m - 3D + 9$ to $9m + 3$. In total, we performed 213 5-to-3 descents, 187 4-to-3 descents, and 11442 3-to-2 descents. These computations took 24.0 CPU hours (Sandy Bridge-EP 16 cores), 0.8 CPU hours (Sandy Bridge, 4 cores), and 44.8 CPU hours (Sandy Bridge, 4 cores), respectively. The running times of the 2-to-1 descents were incorporated into the running times for the higher-level descents.

5.4 Higher extension degrees

As mentioned in the introduction of this chapter, there have been several practical improvements and refinements in discrete logarithm algorithms since Joux's $L[\frac{1}{4}+o(1)]$ algorithm. Most notably, Granger, Kleinjung and Zumbrägel [70] presented several refinements that allowed them to compute logarithms in the 4404-bit characteristic-two field $\mathbb{F}_{2^{12\cdot 367}}$, and Joux and Pierrot [89] presented a faster algorithm for computing logarithms of factor base elements and used it to compute logarithms in the 3796-bit characteristic-three field $\mathbb{F}_{3^{5\cdot 479}}$.

In Section 5.4.1, we show that the techniques from [89] and [70] can be used to lower the estimate from [2] for computing discrete logarithms in the 4841-bit characteristic-three field $\mathbb{F}_{3^{6\cdot 509}}$ from $2^{81.7}M_{q^2}$ to $2^{58.9}M_q$ (where $q = 3^6$). In Section 5.4.2, we use techniques from [70] to lower the estimate from [4] for computing discrete logarithms in the 13590-bit characteristic-three field $\mathbb{F}_{3^{6\cdot 1429}}$ from $2^{95.8}M_{q^2}$ to $2^{78.8}M_{q^2}$ (where $q = 3^6$). We emphasize that these estimates are *upper bounds* on the running times of known algorithms for computing discrete logarithms. Of course, it is possible that these upper bounds can be lowered with a more judicious choice of algorithm parameters, or with a tighter analysis, or with improvements to the algorithms themselves.

5.4.1 Computing discrete logarithms in $\mathbb{F}_{3^{6\cdot 509}}$.

As in Section 4 of [2], we are interested in computing discrete logarithms in the order r-subgroup of $\mathbb{F}^*_{3^{6\cdot 509}}$, where $r = (3^{509} - 3^{255} + 1)/7$ is an 804-bit prime.

We use the algorithm developed by Joux and Pierrot [89], whence $q = 3^6$ and $k = 1$. The field $\mathbb{F}_{3^6}$ is represented as $\mathbb{F}_3[u]/(u^6 + 2u^4 + u^2 + 2u + 2)$. The field $\mathbb{F}_{3^{6\cdot 509}}$ is represented as $\mathbb{F}_{3^6}[X]/(I_X)$, where I_X is the degree-509 irreducible factor of $h_1(X)X^q - h_0(X)$ with $h_0(X) = u^{46}X + u^{219}$ and $h_1(X) = X(X + u^{409})$. Joux and Pierrot [89] exploit the special form of $h_0(X)$ and $h_1(X)$ to accelerate the computation of logarithms of polynomials of degree ≤ 4; the dominant step is the computation of logarithms of degree-3 polynomials, where q linear algebra problems are solved each taking time approximately $q^5/27\, A_r$. The continued-fractions, classical and Gröbner bases descents are all performed over $\mathbb{F}_q$.

The new cost estimates are presented in Table 5.1. We used the estimates for smoothness testing from [69], and the ‘bottom-top’ approach from [70] for estimating the cost of Gröbner bases descent from degree 15 to degree 4. We assume that 2^{27} multiplications in $\mathbb{F}_{3^6}$ can be performed in 1 second; we achieved this performance using a look-up table approach. The timings for Gröbner bases descent and $\mathbb{F}_{3^6}$ multiplications were obtained on an Intel i7-3930K 3.2 GHz machine. In a non-optimized C implementation, we have observed an A_r cost of 43 clock cycles, where lazy reduction is used to amortize the cost of a modular reduction among many integer additions. This yields the cost ratio $A_r/M_q \approx 2$.

The main effect of the improvements is the removal of the QPA descent stage from the estimates in [2]. The overall running time is $2^{58.9}M_q$, a significant improvement over the $2^{81.7}M_{q^2}$ estimate from [2]. In particular, assuming the availability of processors that can perform 2^{27} $\mathbb{F}_{3^6}$-multiplications per second, the estimated running time is approximately 127 CPU years — this is a feasible computation if one has access to a few hundred cores.

Table 5.1: Estimated costs of the main steps for computing discrete logarithms in $\mathbb{F}_{3^{6\cdot 509}}$ ($q = 3^6$). A_r and M_q denote the costs of an addition modulo the 804-bit prime $r = (3^{509} - 3^{255} + 1)/7$ and a multiplication in $\mathbb{F}_{3^6}$. We use the cost ratio $A_r/M_q = 2$, and also assume that 2^{27} multiplications in $\mathbb{F}_{3^6}$ can be performed in 1 second

Finding logarithms of polynomials of degree ≤ 4		
Linear algebra	$2^{52.3}A_r$	$2^{53.3}M_q$
Descent		
Continued-fractions (254 to 40)	$2^{56.9}M_q$	$2^{56.9}M_q$
Classical (40 to 21)	$12.7 \times 2^{54.2}M_q$	$2^{57.9}M_q$
Classical (21 to 15)	$159 \times 2^{49.4}M_q$	$2^{56.7}M_q$
Gröbner bases (15 to 4)	1924×8249 seconds	$2^{50.9}M_q$

Remark 1. The strategy for computing logarithms in $\mathbb{F}_{3^{6\cdot 509}}$ can be employed to compute logarithms in $\mathbb{F}_{3^{6\cdot 239}}$. The latter problem is of cryptographic interest because the prime-order elliptic curve $y^2 = x^3 - x - 1$ over $\mathbb{F}_{3^{239}}$ has embedding degree 6 and has been considered in several papers including [73] and [22]. One could use continued-fractions descent from degree 119 to degree 20 with an estimated cost of $2^{50}M_q$, followed by a classical descent stage from degree 20 to degree 15 at a cost of $2^{53.2}M_q$, and finally Gröbner bases descent to degree 4 at a cost of $2^{47.2}M_q$. The total computational effort is $2^{54.3}M_q$, or approximately 5.2 CPU years.

5.4.2 Computing discrete logarithms in $\mathbb{F}_{3^{6\cdot 1429}}$.

As in Section 4 of [4], we are interested in computing discrete logarithms in the order r-subgroup of $\mathbb{F}^*_{3^{6\cdot 1429}}$, where $r = (3^{1429} - 3^{715} + 1)/7622150170693$ is a 2223-bit prime. To accomplish this, we embed $\mathbb{F}_{3^{6\cdot 1429}}$ in its quadratic extension $\mathbb{F}_{3^{12\cdot 1429}}$. Let $q = 3^6$ and $k = 2$. The field $\mathbb{F}_{3^{12\cdot 1429}}$ is represented as $\mathbb{F}_{q^2}[X]/(I_X)$, where I_X is a monic degree-1429 irreducible factor of $h_1(X^q)\cdot X - h_0(X^q)$ with $h_0, h_1 \in \mathbb{F}_{q^2}[X]$ and $\max(\deg h_0, \deg h_1) = 2$.

The techniques from [70] employed to improve the estimates of [4] are the following:

1. Since logarithms are actually sought in the field $\mathbb{F}_{3^{6\cdot 1429}}$, the continued fractions and classical descent stages are performed over $\mathbb{F}_q$ (and not $\mathbb{F}_{q^2}$).

2. In the final classical descent stage to degree 11, one permits irreducible factors over $\mathbb{F}_q$ of even degree up to 22; any factors of degree $2t \geq 12$ that are obtained can be written as a product of two degree-t irreducible polynomials over $\mathbb{F}_{q^2}$.

3. The number of irreducible factors of an m-smooth degree-t polynomial is estimated as t/m.

4. The smoothness testing estimates from Appendix B of [69] were used.

The remaining steps of the algorithm, namely finding logarithms of linear polynomial, finding logarithms of irreducible quadratic polynomials, QPA descent, and Gröbner bases descent, are as described in [4].

The new cost estimates are presented in Table 5.2. The main effect of the techniques from [70] is the removal of one QPA descent stage. The overall running time is $2^{78.8}M_{q^2}$, a significant improvement over the $2^{95.8}M_{q^2}$ estimate from [4].

Table 5.2: Estimated costs of the main steps for computing discrete logarithms in $\mathbb{F}_{3^{12 \cdot 1429}}$ ($q = 3^6$). A_r, M_q, and M_{q^2} denote the costs of an addition modulo the 2223-bit prime r, a multiplication in $\mathbb{F}_{3^6}$, and a multiplication in $\mathbb{F}_{3^{12}}$. We use the cost ratio $A_r/M_{q^2} = 4$, and also assume that 2^{26} (resp. 2^{27}) multiplications in $\mathbb{F}_{3^{12}}$ (resp. $\mathbb{F}_{3^6}$) can be performed in 1 second (cf. Section 5.4.1)

Finding logarithms of linear polynomials		
Relation generation	$2^{28.6} M_{q^2}$	$2^{28.6} M_{q^2}$
Linear algebra	$2^{47.5} A_r$	$2^{49.5} M_{q^2}$
Finding logarithms of irreducible quadratic polynomials		
Relation generation	$3^{12} \times 2^{37.6} M_{q^2}$	$2^{56.6} M_{q^2}$
Linear algebra	$3^{12} \times 2^{47.5} A_r$	$2^{68.5} M_{q^2}$
Descent		
Continued-fractions (714 to 88)	$2^{77.6} M_q$	$2^{77.6} M_q$
Classical (88 to 29)	$16.2 \times 2^{73.5} M_q$	$2^{77.5} M_q$
Classical (29 to 11)	$267.3 \times 2^{70.8} M_q$	$2^{78.9} M_q$
QPA (11 to 7)	$2^{13.9} \times (2^{44.4} M_{q^2} + 2^{47.5} A_r)$	$2^{63.4} M_{q^2}$
Gröbner bases (7 to 4)	$2^{35.2} \times$ (76.9 seconds)	$2^{67.5} M_{q^2}$
Gröbner bases (4 to 3)	$2^{44.7} \times$ (0.03135 seconds)	$2^{65.7} M_{q^2}$
Gröbner bases (3 to 2)	$2^{54.2} \times$ (0.002532 seconds)	$2^{71.6} M_{q^2}$

5.5 On the asymptotic nature of the QPA algorithm

Let E denote the supersingular elliptic curve $y^2+y = x^3+x$ or $y^2+y = x^3+x+1$ over $\mathbb{F}_{2^n}$ where n is prime, and suppose that $\#E(\mathbb{F}_{2^n}) = cr$ where r is prime and $c \ll r$. The Weil and Tate pairings reduce the discrete logarithm problem in the order-r subgroup of $E(\mathbb{F}_{2^n})$ to the discrete logarithm problem in the order-r subgroup of the multiplicative group of $\mathbb{F}_{2^{4n}}$. Coppersmith's subexponential-time algorithm [40] can be used to solve the latter problem.

In contrast, the QPA algorithm of Barbulescu *et al.* [13] tackles the problem by embedding $\mathbb{F}_{2^{4n}}$ in $\mathbb{F}_{q^{2n}}$ where $q = 2^\ell \approx n$. The running time of the QPA algorithm is dominated by the descent stage. In this stage, one begins with a polynomial of degree (at most) $n-1$ over $\mathbb{F}_{q^2}$ whose logarithm is sought. One then expresses the logarithm of this polynomial in terms of the logarithms of roughly q^2 polynomials of degree at most $n/2$. This process is applied recursively to each polynomial encountered in the "descent tree"; the logarithm of each such polynomial of degree d is expressed in terms of the logarithms of roughly q^2 polynomials of degree at most $d/2$. To terminate the recursion, the logarithms of all degree-1 polynomials are obtained using a relatively fast method. The number of nodes in the descent tree gives a very crude lower bound on the running time of the QPA algorithm. Since $n \approx q$, the descent tree has approximately $\log_2 q$ levels and at least $q^{2\log_2 q}$ nodes.

Table 5.3 compares the running time $C(q) = \exp(1.526(\log 2^{4q})^{1/3}(\log\log 2^{4q})^{2/3})$ of Coppersmith's algorithm for computing discrete logarithms in $\mathbb{F}_{2^{4q}}$, and the lower bound $q^{2\log_2 q}$ on the running time of the QPA algorithm for computing discrete logarithms in $\mathbb{F}_{q^{2n}}$ with $q \approx n$.

Table 5.3: Comparison of the running time $q^{2\log_2 q}$ of the QPA algorithm for computing logarithms in $\mathbb{F}_{q^{2n}}$ with $q \approx n$, and the running time $C(q)$ of Coppersmith's algorithm for computing logarithms in $\mathbb{F}_{2^{4n}}$

q	$q^{2\log_2 q}$	$C(q)$
2^9	2^{162}	2^{93}
2^{10}	2^{200}	2^{124}
2^{11}	2^{242}	2^{165}
2^{12}	2^{288}	2^{219}
2^{13}	2^{338}	2^{290}
2^{14}	2^{392}	2^{382}
2^{15}	2^{450}	2^{501}

We see from Table 5.3 that the QPA algorithm is faster than Coppersmith's algorithm only when $n \approx q = 2^{15}$. However, such n is too large to be of interest in cryptography based on pairings over $E(\mathbb{F}_{2^n})$.

As already stated in [13, Section 6.2], to determine the practical efficiency of the QPA algorithm, and therefore the implications of QPA to the security of pairing-based cryptosystems based on $E(\mathbb{F}_{2^n})$, it is imperative that the descent stage of QPA be combined with descent steps from classical algorithms. The asymptotic running time of the resulting hybrid algorithm is difficult to determine. Instead, the framework and tools introduced in [2] are used to perform a concrete analysis which provides a reasonably accurate picture of the effectiveness of the hybrid algorithm.

5.6 Summary

In this chapter, we applied the recent techniques for solving the DLP on small-characteristic fields to the cases $\mathbb{F}_{3^{6\cdot 137}}$ and $\mathbb{F}_{3^{6\cdot 163}}$. The implementations were done completely in the Magma algebra system and took 918 CPU hours and 1201 CPU hours, respectively.

Next, we realized a concrete analisis of the cost to solve the DLP in the fields $\mathbb{F}_{3^{6\cdot 509}}$ and $\mathbb{F}_{3^{6\cdot 1429}}$ using the Joux-Pierrot approach. Both fields were previously proposed as primitives for pairing-based protocols. Finally, we presented estimations for the Barbulescu *et al.* quasi-polynomial algorithm on different extension fields in order to verify its feasibility on being applied to fields of cryptographic interest.

6 | Elliptic and Hyperelliptic Curves

In the last two decades, the elliptic curve cryptosystems introduced by Koblitz and Miller [98, 113] have been increasingly employed to instantiate public-key standards [132] and protocols [44, 146]. The main reason for that is their reduced key size, which accommodate fast and lightweight implementations.

In 2011, Galbraith, Lin and Scott (GLS) [61] introduced efficient computable endomorphisms for a large class of elliptic curves defined over $\mathbb{F}_{p^2}$, where p is a prime number. Later, Hankerson, Karabina and Menezes [76] analyzed the GLS curves defined over characteristic two fields $\mathbb{F}_{2^{2n}}$, with prime n. For more details of the GLS curves, see Chapter 3.

Since then, many authors combined the GLS efficient endomorphisms with the Gallant-Lambert-Vanstone decomposition method [63] to present high-performance scalar multiplication software implementations over binary [76, 120] and prime [81, 103, 29, 50] fields.

The theoretical security of an elliptic curve is given by the complexity of solving the discrete logarithm problem (DLP) on its group of points. Given an elliptic curve E defined over a field $\mathbb{F}_q$, a generator point $P \in E(\mathbb{F}_q)$ of order r and a challenge point $Q \in \langle P \rangle$, the DLP on E consists in computing the integer $\lambda \in \mathbb{Z}_r$ such that $Q = \lambda P$.

Among the classical methods for solving the DLP on $E(\mathbb{F}_q)$ we can cite the Baby Step Giant Step [39, Section 19.4] and Pollard Rho [126] algorithms. Both of them run in time $O(\sqrt{q})$. In 1993, Menezes, Okamoto and Vanstone presented a method [112] that uses the Weil pairing to reduce the DLP on $E(\mathbb{F}_q)$ to the same problem on $\mathbb{F}^*_{q^k}$, where k is the smallest positive integer such that $r \mid q^k - 1$. In binary curves where k is small, the attack is highly effective, because there exist quasi-polynomial algorithms for solving the discrete logarithm on small-characteristic finite fields [13]. For binary curves, we also have algorithms based on the index-calculus approach [49] which run in time $O(\sqrt{q^\omega})$, where ω is a constant related to the linear algebra.

In 2000, Gaudry, Hess and Smart (GHS) [65] applied the ideas in [54, 62] to reduce any instance of the DLP on a binary curve $E/\mathbb{F}_{2^{ln}}$ to one on the Jacobian of a hyperelliptic curve defined over a subfield $\mathbb{F}_{2^l}$. Afterwards, Galbraith, Hess and Smart [60] extended the attack by using isogenies. Next, Hess [80] generalized the attack (gGHS) to arbitrary Artin-Schreier extensions.

The analysis of the practical implications of the GHS Weil descent method were made by Menezes and Qu [108] who demonstrated that the attack is infeasible for elliptic curves over $\mathbb{F}_{2^n}$ with primes $n \in [160, \dots, 600]$ and by Menezes, Teske and Weng [111] who showed that the attack can be applied to curves defined over composite extensions of a binary field. Finally, the authors in [76] analyzed the application of the gGHS attack over GLS binary curves $E/\mathbb{F}_{2^{2n}}$ and concluded that for $n \in [80, \dots, 256]$, the degree-127 extension is the only one that contains vulnerable curve isogeny classes.

In this work, we wanted to get a practical perspective of the GHS Weil descent attack. In order to achieve this goal, we implemented the attack against a binary GLS elliptic curve on the Magma computer algebra system. The implementation included the construction of vulnerable curves, the search for susceptible isogenous curves and the adaptation of the Enge-Gaudry algorithm [46] to solve the discrete logarithm problem on the generated hyperelliptic curve.

Moreover, we proposed a mechanism to check for unsafe binary curve parameters against the GHS attack. The Magma source code for the algorithms presented in this document is available at `http://computacion.cs.cinvestav.mx/~thomaz/gls.tar.gz`. Our program can be easily adapted for any extension field and can be executed on single and multi-core architectures.

6.1 Hyperelliptic Curves

Let $\mathbb{F}_{2^l}$ be a finite field of 2^l elements, for some positive integer l, and let $\mathbb{F}_{2^{ln}}$ be a degree-n extension field of $\mathbb{F}_{2^l}$. A hyperelliptic curve $H/\mathbb{F}_{2^l}$ of genus g is given by the following non-singular equation,

$$H/\mathbb{F}_{2^l} \colon y^2 + h(x)y = f(x), \tag{6.1}$$

where $f, h \in \mathbb{F}_{2^l}[x]$, $\deg(f) = 2g + 1$ and $\deg(h) \leq g$. The set of $\mathbb{F}_{2^{ln}}$-rational points on H is $H(\mathbb{F}_{2^{ln}}) = \{(x, y) \colon x, y \in \mathbb{F}_{2^{ln}},\ y^2 + h(x)y = f(x)\} \cup \{\mathcal{O}\}$. The opposite of a point $P = (x, y) \in H(\mathbb{F}_{2^{ln}})$ is denoted as $\overline{P} = (x, y + h(x))$ and $\overline{\mathcal{O}} = \mathcal{O}$.

The group law is not defined over the curve itself but on the Jacobian of H, denoted by $J_H(\mathbb{F}_{2^l})$, which is defined in terms of the set of divisors on H. A divisor is a finite formal sum of points on the curve and the set of all divisors on H yield an abelian group denoted by $\text{Div}(H)$. Let c_i be an integer, then for each divisor

$D = \sum_{P_i \in H} c_i(P_i)$, $\deg(D) = \sum c_i$ is the degree of D. The set $\mathrm{Div}^0(H)$ of degree-zero divisors forms a subgroup of $\mathrm{Div}(H)$.

The function field $\overline{\mathbb{F}}_{2^{ln}(H)}$ of H is the set of rational functions on H. For each non-zero function $\varphi \in \overline{\mathbb{F}}_{2^{ln}}(H)$, we can associate a divisor $\mathrm{div}(\varphi) = \sum_{P_i \in H} \nu_{P_i}(\varphi)(P_i)$, where $\nu_{P_i}(\varphi)$ is an integer defined as follows:

$$\nu_{P_i}(\varphi) = \begin{cases} \text{the multiplicity of } P_i \text{ with respect to } \varphi & \text{if } \varphi \text{ has a zero at } P_i \\ \text{the negative of the multiplicity of } P_i & \\ \text{with respect to } \varphi & \text{if } \varphi \text{ has a pole at } P_i \\ 0 & \text{otherwise.} \end{cases}$$

A non-zero rational function has only finitely many zeroes and poles. In addition, the number of poles equals the number of zeroes (with multiplicity). Therefore, $\nu_{P_i}(\varphi)$ is equal to zero for almost all P_i and $\mathrm{div}(\varphi)$ is consequently well defined.

The divisor $\mathrm{div}(\varphi)$ is called principal. Given two functions φ_0 and $\varphi_1 \in \overline{\mathbb{F}}_{2^{ln}}(H)$, the difference of two principal divisors $\mathrm{div}(\varphi_0)$ and $\mathrm{div}(\varphi_1)$ is also a principal divisor, corresponding to the fraction of the two functions. The set $\mathcal{P}(H)$ of principal divisors contains 0 as $\mathrm{div}(1)$ and is a subgroup of $\mathrm{Div}^0(H)$. The Jacobian of the curve H is given by the quotient group $J_H(\overline{\mathbb{F}}_{2^l}) = \mathrm{Div}^0(H)/\mathcal{P}(H)$ and $J_H(\mathbb{F}_{2^l})$ is the Jacobian over $\mathbb{F}_{2^l}$. Note that $\#J_H(\mathbb{F}_{2^l}) \approx 2^{lg}$.

A consequence of the Riemann-Roch theorem [39, Section 4.4.2] is that every element of the Jacobian can be represented by a divisor of the form

$$D = (P_1) + (P_2) \cdots + (P_r) - r(\mathcal{O}) \tag{6.2}$$

where $P_i \in H$ for $i = 1, \ldots, r$ and $r \leq g$. Furthermore, if $P_i \neq \overline{P_j}$ for all $i \neq j$, then D is called a reduced divisor. A reduced divisor can be uniquely represented by a pair of polynomials $U, V \in \mathbb{F}_{2^l}[x]$ such that (i) $\deg(V) < \deg(U) \leq g$; (ii) U is monic; and (iii) $U | (V^2 + Vh - f)$.

If U and V are two polynomials that satisfy the above conditions, we denote by $\mathrm{div}(U, V)$ the corresponding element of $J_H(\mathbb{F}_{2^l})$. When U is irreducible in $\mathbb{F}_{2^l}[x]$ we say that $\mathrm{div}(U, V)$ is a prime divisor. Let $D = \mathrm{div}(U, V) \in J_H(\mathbb{F}_{2^l})$ and $U = \prod U_i$, where each U_i is an irreducible polynomial in $\mathbb{F}_{2^l}[x]$, and let $V_i = V \bmod U_i$. Then $D_i = \mathrm{div}(U_i, V_i)$ is a prime divisor and $D = \sum D_i$.

6.2 The Hyperelliptic Curve Discrete Logarithm Problem

Let $q = 2^l$ and g be the genus of the hyperelliptic curve $H/\mathbb{F}_q$. The discrete logarithm problem on $J_H(\mathbb{F}_q)$ is defined as follows: given $D_1 \in J_H(\mathbb{F}_q)$ of order r and $D_2 \in \langle D_1 \rangle$, find $\lambda \in \mathbb{Z}_r$ such that $D_2 = \lambda D_1$.

Besides the Pollard Rho algorithm, whose time complexity is $O(\sqrt{\frac{\pi q^g}{2}})$, the methods proposed in the literature for solving the DLP on H are index-calculus-based algorithms:

1. Gaudry in [64] proposed an algorithm whose complexity is $O(g^3 q^2 \log^2 q + g^2 g! q \log^2 q)$. If one considers a fixed genus g, the algorithm executes in time $O(q^{2+\epsilon})$. In [65], the algorithm is modified to perform in time $O(q^{\frac{2g}{g+1}+\epsilon})$. Here, ϵ is a number less than 1.

2. The Enge-Gaudry algorithm [46] has an expected running time of $L_{q^g}[\sqrt{2}]$ when $g/\log q \to \infty$. Here, $L_x[c]$ denotes the expression $e^{((c+o(1))\sqrt{\log x}\sqrt{\log\log x})}$.

3. In [66], Gaudry *et al.* propose a double large prime variation in order to improve the relation collection phase. For curves with fixed genus $g \geq 3$ the algorithm runs in time $\tilde{O}(q^{2-\frac{2}{g}})$.

4. The approach from Sarkar and Singh [137, 138], based on the Nagao's work [115], avoids the requirement of solving a multi-variate system and combines a sieving method proposed by Joux and Vitse [90]. They showed that it is possible to obtain a single relation in about $(2g+3)!$ trials.

6.3 The Gaudry-Hess-Smart (GHS) Weil descent attack

Let $\mathbb{F}_{2^{ln}}$ be a degree-n extension of $\mathbb{F}_{2^l}$ and let E be an elliptic curve defined over $\mathbb{F}_{2^{ln}}$ given by the equation

$$E/\mathbb{F}_{2^{ln}} : y^2 + xy = x^3 + ax^2 + b \qquad a \in \mathbb{F}_{2^{ln}},\ b \in \mathbb{F}^*_{2^{ln}}. \tag{6.3}$$

The GHS Weil descent attack [65] consists of the following steps,

1. The Weil descent:

(a) Construct the Weil restriction $W_{E/\mathbb{F}_{2^l}}$ of scalars of E, which is an n-dimensional abelian variety over $\mathbb{F}_{2^l}$. One can construct this variety as follows. Let $\beta = \{\phi_1, \ldots, \phi_n\}$ be a basis of $\mathbb{F}_{2^{ln}}$ viewed as a vector space over $\mathbb{F}_{2^l}$. Then write a, b, x and y in terms of β,

$$a = \sum_{i=1}^{n} a_i\phi_i, \quad b = \sum_{i=1}^{n} b_i\phi_i, \quad x = \sum_{i=1}^{n} x_i\phi_i \text{ and } y = \sum_{i=1}^{n} y_i\phi_i. \tag{6.4}$$

Given that β is a linearly independent set, by substituting the equations (6.4) into the equation (6.3) we obtain an n-dimensional abelian variety A defined over $\mathbb{F}_{2^l}$. Moreover, the group law of A is similar to the elliptic curve E group law.

(b) Intersect A with $n-1$ hyperplanes (e.g. $x_1 = x_2 = \cdots = x_n = x$) to obtain a subvariety of A, and then use its linear independence property to obtain a curve H over $\mathbb{F}_{2^l}$.

2. Reduce the DLP on $E(\mathbb{F}_{2^{ln}})$ to the DLP on $J_H(\mathbb{F}_{2^l})$.

3. Solve the DLP on $J_H(\mathbb{F}_{2^l})$.

Let $\gamma \in \mathbb{F}_{2^{ln}}$, $\sigma\colon \mathbb{F}_{2^{ln}} \to \mathbb{F}_{2^{ln}}$ be the Frobenius automorphism defined as $\sigma(\gamma) = \gamma^{2^l}$, $\gamma_i = \sigma^i(\gamma)$ for all $i \in \{0, \ldots, n-1\}$ and

$$m = m(\gamma) = \dim(\mathrm{Span}_{\mathbb{F}_2}\{(1, \gamma_0^{1/2}), \ldots, (1, \gamma_{n-1}^{1/2})\}).$$

Finally, let us assume that

$$\text{either } n \text{ is odd or } m(b) = n \text{ or } \mathrm{Tr}_{\mathbb{F}_{2^{ln}}/\mathbb{F}_2}(a) = 0. \tag{6.5}$$

Then the GHS Weil descent attack constructs an explicit group homomorphism $\chi\colon E(\mathbb{F}_{2^{ln}}) \to J_H(\mathbb{F}_{2^l})$, where H is a hyperelliptic curve defined over $\mathbb{F}_{2^l}$ of genus $g = 2^{m-1}$ or $g = 2^{m-1} - 1$.

6.3.1 The generalized GHS (gGHS) Weil descent attack

In [80] Hess generalized the GHS restrictions (6.5) as follows. Let $\wp(x) = x^2 + x$ and $F = \mathbb{F}_{2^{ln}}(x)$, $\Delta = f\mathbb{F}_2[\sigma] + \wp(F)$ where $f = \gamma_1/x + \gamma_3 + x\gamma_2$ for $\gamma_1, \gamma_2, \gamma_3 \in \mathbb{F}_{2^{ln}}$ such that $\gamma_1\gamma_2 \neq 0$.

Given a polynomial $p = \sum_{i=0}^{d} p_i x^i \in \mathbb{F}_2[x]$ of degree d we write $p(\sigma)(x) = \sum_{i=0}^{d} p_i x^{2^{li}}$. For each element $\gamma \in \mathbb{F}_{2^{ln}}$, $\mathrm{Ord}_\gamma(x)$ is the unique monic polynomial

$p \in \mathbb{F}_2[x]$ of least degree such that $p(\sigma)(\gamma) = 0$. Furthermore, we define the m-degree $\mathrm{Ord}_{\gamma_1,\gamma_2,\gamma_3}$ as,

$$\mathrm{Ord}_{\gamma_1,\gamma_2,\gamma_3} = \begin{cases} \mathrm{lcm}(\mathrm{Ord}_{\gamma_1}, \mathrm{Ord}_{\gamma_2}) & \text{if } \mathrm{Tr}_{\mathbb{F}_{2^{ln}}/\mathbb{F}_2}(\gamma_3) = 0 \\ \mathrm{lcm}(\mathrm{Ord}_{\gamma_1}, \mathrm{Ord}_{\gamma_2}, x+1) & \text{otherwise.} \end{cases}$$

Then $\Delta/\wp(F) \cong \mathbb{F}_2[x]/\mathrm{Ord}_{\gamma_1,\gamma_2,\gamma_3}$ and the Frobenius automorphism σ of F with respect to $\mathbb{F}_{2^{ln}}$ extends to a Frobenius automorphism of a function field $C = F(\wp^{-1}(\Delta))$ with respect to $\mathbb{F}_{2^{ln}}$ if and only if,

$$\text{either } \mathrm{Tr}_{\mathbb{F}_{2^{ln}}/\mathbb{F}_2}(\gamma_3) = 0 \text{ or } \mathrm{Tr}_{\mathbb{F}_{2^{ln}}/\mathbb{F}_{2^l}}(\gamma_1) \neq 0 \text{ or } \mathrm{Tr}_{\mathbb{F}_{2^{ln}}/\mathbb{F}_{2^l}}(\gamma_2) \neq 0. \tag{6.6}$$

In addition, the genus of C is given by

$$g_C = 2^m - 2^{m-\deg(\mathrm{Ord}_{\gamma_1})} - 2^{m-\deg(\mathrm{Ord}_{\gamma_2})} + 1$$

and there exists a curve H with genus g_C that can be related to an elliptic curve $E/\mathbb{F}_{2^{ln}} : y^2 + xy = x^3 + ax^2 + b$ with $a = \gamma_3$ and $b = (\gamma_1\gamma_2)^2$.

6.3.2 Using isogenies to extend the attacks

Let E and E' be two ordinary elliptic curves defined over $\mathbb{F}_{2^{ln}}$ and given by the equation (6.3). A rational map $\Psi\colon E \to E'$ over $\mathbb{F}_{2^{ln}}$ is an element of the elliptic curve $E'(\mathbb{F}_{2^{ln}}(E))$. An isogeny $\Phi\colon E \to E'$ over $\mathbb{F}_{2^{ln}}$ is a non-constant rational map over $\mathbb{F}_{2^{ln}}$ and is also a group homomorphism from $E(\mathbb{F}_{2^{ln}})$ to $E'(\mathbb{F}_{2^{ln}})$. In that case, we say that E and E' are isogenous. It is known that E and E' are isogenous over $\mathbb{F}_{2^{ln}}$ if and only if $\#E(\mathbb{F}_{2^{ln}}) = \#E'(\mathbb{F}_{2^{ln}})$ [147].

An isogeny $\Phi\colon E \to E'$ induces a map $\Phi^*\colon \mathbb{F}_{2^{ln}}(E') \to \mathbb{F}_{2^{ln}}(E)$, called the pullback of Φ [57], which is necessarily injective,

$$\begin{array}{rcl} \Phi^*\colon \mathbb{F}_{2^{ln}}(E') & \to & \mathbb{F}_{2^{ln}}(E) \\ \theta & \to & \theta \circ \Phi. \end{array}$$

If $x \in E'(\mathbb{F}_{2^{ln}})$, we can pull back x along Φ, and obtain a divisor

$$D = \sum_{P \in \Phi^{-1}(x)} \nu_P(\Phi)(P).$$

The degree δ of Φ is defined by the integer $[\mathbb{F}_{2^{ln}}(E)\colon \Phi^*(\mathbb{F}_{2^{ln}}(E'))]$ and we say that Φ is a δ-isogeny.

The authors in [60] propose to extend the range of vulnerable curves against the GHS attack (and equivalently the gGHS attack) by finding an explicit representation for an isogeny $\Phi\colon E \to E'$ and determining if there exists at least one elliptic curve E' against which the attack is effective.

6.4 Analyzing the GLS elliptic curves

Let $\mathbb{F}_{2^{2n}}$ be a degree-2 extension of $\mathbb{F}_{2^n}$. Also, let $E/\mathbb{F}_{2^n}$ be an ordinary elliptic curve given by the equation

$$E/\mathbb{F}_{2^n} : y^2 + xy = x^3 + ax^2 + b \qquad a \in \mathbb{F}_{2^n},\ b \in \mathbb{F}_{2^n}^*, \tag{6.7}$$

with $\mathrm{Tr}(a) = 1$. We know that $\#E(\mathbb{F}_{2^n}) = q + 1 - t$, where t is the trace of E over $\mathbb{F}_{2^n}$. It follows that $\#E(\mathbb{F}_{2^{2n}}) = (q+1)^2 - t^2$. Let $a' \in \mathbb{F}_{2^{2n}}$ such that $\mathrm{Tr}(a') = 1$. Then we can construct the GLS curve,

$$E'/\mathbb{F}_{2^{2n}} : y^2 + xy = x^3 + a'x^2 + b. \tag{6.8}$$

Which is isomorphic to E over $\mathbb{F}_{2^{4n}}$ under the involutive isomorphism $\tau : E \to E'$. The GLS endomorphism can be constructed by applying τ with the Frobenius automorphism σ, defined as $(x, y) \mapsto (x^{2^n}, y^{2^n})$, as follows, $\psi = \tau\sigma\tau^{-1}$.

6.4.1 Applying the GHS Weil descent attack

The theoretical security of a given binary GLS curve $E/\mathbb{F}_{2^{2n}}$ depends basically on the complexity of solving the DLP on its group of points $E(\mathbb{F}_{2^{2n}})$. As discussed in the introduction of this chapter, the usual approach is to apply the Pollard Rho algorithm for elliptic curves, which runs in approximately $\sqrt{\frac{\pi 2^{2n}}{2}}$ operations [126].

However, after the publication of the GHS reduction, it is also necessary to check whether the complexities of solving the DLP on $J_H(\mathbb{F}_2)$, $J_H(\mathbb{F}_{2^2})$ or $J_H(\mathbb{F}_{2^n})$ are lower than solving it on $E(\mathbb{F}_{2^{2n}})$. If such is the case, the smallest complexity provides us the real security of the curve E.

Let us assume that the number of isogenous curves E' is smaller than the number of vulnerable isogeny classes, then the following steps describe a method for determining if a given GLS curve is vulnerable against the extended GHS attack:

1. **Setting the environment.** Let us have a GLS curve $E_{\tilde{a},\tilde{b}}/\mathbb{F}_{2^{2n}}$ given by the equation (6.8) but defined with the particular parameters $\tilde{a}$ and $\tilde{b}$. In the context of the GHS attack, the extension field $\mathbb{F}_{2^{2n}}$ can be seen as a degree-n extension of $\mathbb{F}_{2^2}$ or a degree-$2n$ extension of $\mathbb{F}_2$. For the sake of simplicity, we will represent the base field as $\mathbb{F}_{2^2}$. Nonetheless, the steps must be executed for both base representations.

2. **Checking the $\tilde{b}$ parameter.** We know that $(x^n+1)(\sigma) = x^{2^{2n}} + x = 0 \Leftrightarrow x^{2^{2n}} = x$. In addition, $\mathrm{Ord}_\gamma | (x^n+1)$. Given that the polynomial x^n+1 factorizes as $(x+1) \cdot f_i \cdot \ldots \cdot f_s$, let $d = \deg(f_i)$. Then, search a pair of polynomials $s_1 = (x+1)^{j_1} \cdot f_i^{j_2}$ and $s_2 = (x+1)^{j_3} \cdot f_i^{j_4}$, with positive integers j_i and find a representation of $\tilde{b}$ as $(\gamma_1\gamma_2)^2$, such that $\mathrm{Ord}_{\gamma_i} = s_i(\sigma)$ and $\mathrm{Ord}_{\gamma_1,\gamma_2,\tilde{a}}$ derive a small associated value g_C.

3. **Solving the DLP on a hyperelliptic curve.** If such minimum pair s_1, s_2 exists (i), apply the Weil descent on E to construct a hyperelliptic curve H. Check if the complexity of solving the DLP on $J_H(\mathbb{F}_{2^2})$ is smaller than solving it on $E(\mathbb{F}_{2^{2n}})$ (ii). If that is the case, the curve E is vulnerable against the GHS attack. If either (i) or (ii) is false, go to step 4.

4. **The extended GHS attack.** For each isogenous curve E' to E, perform the check (steps 2 and 3). If there is no vulnerable elliptic curve E' isogenous to E, then the curve E is not vulnerable against the extended GHS attack.

If the number of isogenous curves E' is greater than the number of vulnerable isogeny classes, a more efficient method to perform the vulnerability check is to list all vulnerable parameters $\hat{b}$ and store all of the related group orders $\#E_{a,\hat{b}}(\mathbb{F}_{2^{2n}})$ in a set L. The check consists in verifying whether $\#E_{a,b}(\mathbb{F}_{2^{2n}}) \in L$ [76].

The extension field $\mathbb{F}_{2^{2n}}$ can also be represented as a degree-2 extension of $\mathbb{F}_{2^n}$. However, as analyzed in [76], in this setting the GHS attack generates hyperelliptic curves of genus 2 or 3. Solving the DLP on the Jacobian of these curves is not easier than solving it on $E(\mathbb{F}_{2^{2n}})$ with the Pollard Rho method.

The complexity of solving the DLP on $J_H(\mathbb{F}_{2^2})$ (or $J_H(\mathbb{F}_2)$) is determined by the genus of the curve H (see Section 6.2). In the GHS attack context, the genus of the constructed hyperelliptic curve H is given by the degree of the minimum pair of polynomials (s_1, s_2). For each extension degree n, these values are derived from the factors of the polynomial x^n+1. For this reason, in characteristic two, we have many extensions where the genus of H is large and consequently, the GHS attack is ineffective for any GLS curve defined over such extension fields.

To illustrate those cases, we present in Table 6.1 the costs of solving the DLP with the GHS/Enge-Gaudry approach and the Pollard Rho algorithm on binary GLS curves $E/\mathbb{F}_{2^{2n}}$ with $n \in [5, 257]$. We chose all the hidden constant factors in the Enge-Gaudry algorithm complexity to be one, and suppressed all fields whose genus of the generated curve H is higher than 10^6. In addition, the effort of finding a vulnerable curve against the GHS attack is not included in the cost for solving the DLP.

Table 6.1: Different binary GLS curves and their security. The smallest complexity is written in bold type

				Cost for solving the DLP	
Base field of E	Base field of H	Genus of H	$E(\mathbb{F}_{2^{2n}})$ order ($\approx$) (bits)	Pollard Rho algorithm on E (ceiling, bits)	Enge-Gaudry algorithm on H (ceiling, bits)
$\mathbb{F}_{2^{2\cdot 5}}$	$\mathbb{F}_2$ $\mathbb{F}_{2^2}$	32 15	9	**5**	17 16
$\mathbb{F}_{2^{2\cdot 7}}$	$\mathbb{F}_2$ $\mathbb{F}_{2^2}$	16 7	13	**7**	11 10
$\mathbb{F}_{2^{2\cdot 11}}$	$\mathbb{F}_2$ $\mathbb{F}_{2^2}$	2048 1023	21	**11**	207 207
$\mathbb{F}_{2^{2\cdot 13}}$	$\mathbb{F}_2$ $\mathbb{F}_{2^2}$	8192 4095	25	**13**	452 452
$\mathbb{F}_{2^{2\cdot 17}}$	$\mathbb{F}_2$ $\mathbb{F}_{2^2}$	512 255	33	**17**	93 93
$\mathbb{F}_{2^{2\cdot 19}}$	$\mathbb{F}_2$ $\mathbb{F}_{2^2}$	524288 262143	37	**19**	4401 4401
$\mathbb{F}_{2^{2\cdot 23}}$	$\mathbb{F}_2$ $\mathbb{F}_{2^2}$	4096 2047	45	**23**	307 306
$\mathbb{F}_{2^{2\cdot 31}}$	$\mathbb{F}_2$ $\mathbb{F}_{2^2}$	64 31	61	31	26 **26**
$\mathbb{F}_{2^{2\cdot 43}}$	$\mathbb{F}_2$ $\mathbb{F}_{2^2}$	32768 16383	85	**43**	974 974
$\mathbb{F}_{2^{2\cdot 73}}$	$\mathbb{F}_2$ $\mathbb{F}_{2^2}$	1024 511	145	**73**	139 139
$\mathbb{F}_{2^{2\cdot 89}}$	$\mathbb{F}_2$ $\mathbb{F}_{2^2}$	4096 2047	177	**89**	307 306
$\mathbb{F}_{2^{2\cdot 127}}$	$\mathbb{F}_2$ $\mathbb{F}_{2^2}$	256 127	253	127	62 **62**
$\mathbb{F}_{2^{2\cdot 151}}$	$\mathbb{F}_2$ $\mathbb{F}_{2^2}$	65536 32767	301	**151**	1424 1424
$\mathbb{F}_{2^{2\cdot 257}}$	$\mathbb{F}_2$ $\mathbb{F}_{2^2}$	131072 65535	513	**257**	2078 2078

6.4.2 A mechanism for finding vulnerable curves

In this part, we propose a mechanism for performing the step 3 check on the parameter b of a given GLS curve $E_{a,b}$. This mechanism is useful when the number of GHS vulnerable isogeny classes is greater than the number of isogenous curves to $E_{a,b}$. Similarly to the previous section, $\mathbb{F}_{2^{2n}}$ is a degree-n extension field of $\mathbb{F}_{2^2}$. However, the method can be easily adapted to any field representation.

Let $x^n + 1 = (x+1)f_1 \cdots f_s$, where each irreducible polynomial $f_i \in \mathbb{F}_2[x]$ has degree d and $f_i \neq f_j$ for $i \neq j$. Also, let $S = \{(x+1)^{j_1} f_i^{j_2}, (x+1)^{j_3} f_i^{j_4}\}_{j_i \in \{0,1\}}$ be a nonempty finite set where for each pair $(s_1, s_2) \in S$, $\deg(s_1) \geq \deg(s_2)$. Then let $B = \{b = (\gamma_1\gamma_2)^2 \colon \gamma_1, \gamma_2 \in \mathbb{F}^*_{2^{2n}},\ \exists (s_1, s_2) \in S | s_1(\sigma)(\gamma_1) = 0 \wedge s_2(\sigma)(\gamma_2) = 0\}$.

Let $f, g \in F_2[x]$ be two degree-d polynomials. Then we have the following theorems:

Theorem 12. *$(f \cdot g)(\sigma)(x) = (f(\sigma) \circ g(\sigma))(x) = (g(\sigma) \circ f(\sigma))(x)$.*

Proof. Let $q = 2^2$. The expression $(f \cdot g)(x)$ can be written as

$$(\sum_{i=0}^{d} f_i x^i)(\sum_{j=0}^{d} g_j x^j) = \sum_{i=0}^{d}\sum_{j=0}^{d} f_i g_j x^{i+j}.$$

Then,

$$\begin{aligned}(f \cdot g)(\sigma)(x) &= \sum_{i=0}^{d}\sum_{j=0}^{d} f_i g_j x^{q^{i+j}} = \sum_{i=0}^{d}\sum_{j=0}^{d} f_i \left(g_j x^{q^j}\right)^{q^i} = \sum_{i=0}^{d} f_i \sum_{j=0}^{d} \left(g_j x^{q^j}\right)^{q^i} \\ &= \sum_{i=0}^{d} f_i \left(\sum_{j=0}^{d} g_j x^{q^j}\right)^{q^i} = \left(f(\sigma) \circ g(\sigma)\right)(x).\end{aligned}$$

The proof of the case $(f \cdot g)(\sigma)(x) = (g(\sigma) \circ f(\sigma))(x)$ is similar. □

Theorem 13. *$f(\sigma)(x) \mid (f \cdot g)(\sigma)(x)$ over $\mathbb{F}_{2^{2n}}(\alpha_1, \ldots, \alpha_{q^d})$ where each α_i is a root of the polynomial $f(\sigma)$.*

Proof. Given that $p(\sigma)(x)$ has a zero at $x = 0$ for all polynomial $p \in \mathbb{F}_2[x]$, let α be a root of $f(\sigma)(x)$ in its splitting field. Then $g(\sigma)(f(\sigma)(\alpha)) = g(\sigma)(0) = 0$, i.e., α is also a root of $(f \cdot g)(\sigma)$. As a result, $f(\sigma) = \prod(x - \alpha_i)$ divides $(f \cdot g)(\sigma)$ over $\mathbb{F}_{2^{2n}}(\alpha_1, \ldots, \alpha_{q^d})$. □

Theorem 14. *$\forall \gamma \in \mathbb{F}_{2^{2n}}$, we have that $Ord_\gamma(\sigma)(x)$ splits on $\mathbb{F}_2$.*

Proof. From the the previous theorem, we have that for a polynomial $p(x) \in \mathbb{F}_2[x]$, every factor $\overline{p}(x)$ of $p(x)$ satisfies $\overline{p}(\sigma)(x)|p(\sigma)(x)$ over $\mathbb{F}_{2^{2n}}(\alpha_1, \ldots, \alpha_{\deg(\overline{p})})$ where each α_i is a root of $\overline{p}$. Then, since $\mathrm{Ord}_\gamma(x)|x^n+1$ we have that $\mathrm{Ord}_\gamma(\sigma)(x)|(x^{q^{2n}}+x)$ over $\mathbb{F}_{2^{2n}}(\alpha_1, \ldots, \alpha_{\deg(\mathrm{Ord}_\gamma)})$ where each α_i is a root of $\mathrm{Ord}_\gamma(\sigma)(x)$. Consequently, $\mathbb{F}_{2^{2n}} = \mathbb{F}_{2^{2n}}(\alpha_1, \ldots, \alpha_{\deg(\mathrm{Ord}_\gamma)})$ and $\mathrm{Ord}_\gamma(\sigma)(x)$ splits over $\mathbb{F}_2[x]$. □

For a given $b \in \mathbb{F}^*_{2^{2n}}$, we can determine whether b is in B as follows. For all $(s_1, s_2) = (\sum_j s_{1,j}x^j, \sum_j s_{2,j}x^j) \in S$, let $b_i(x) = s_i(\sigma)(b^{1/2}x) = \sum_j (b^{1/2}s_{i,j})x^{q^j}$ and let $\overline{s}_i(x) = x^{\deg(s_i)}s_i(\sigma)(\frac{1}{x})$. Then,

$$\begin{aligned}
\gcd(b_1(x), \overline{s}_2(x)) \neq 1 \quad &\Leftrightarrow \quad \exists \gamma \in \mathbb{F}^*_{2^{2n}} \text{ such that } s_1(\sigma)(b^{1/2}\gamma) = 0 \text{ and } s_2(\sigma)(\frac{1}{\lambda}) = 0 \\
&\Leftrightarrow \quad s_1(\sigma)(x) \text{ has a zero at } \gamma_1 = b^{1/2}\gamma \text{ and} \\
&\qquad\ s_2(\sigma)(x) \text{ has a zero at } \gamma_2 = \frac{1}{\gamma} \\
&\Leftrightarrow \quad b = (\gamma_1\gamma_2)^2 \text{ where } s_1(\sigma)(\gamma_1) = 0 \text{ and} \\
&\qquad\ s_2(\sigma)(\gamma_2) = 0 \\
&\Leftrightarrow \quad b = (\gamma_1\gamma_2)^2 \in B.
\end{aligned}$$

Let us now assume that S contains only the pairs of polynomials (s_1, s_2) that construct parameters b which are vulnerable against the gGHS attack. Then, for an arbitrary GLS curve E we have that,

$$E_{a,b} \text{ is vulnerable} \Leftrightarrow \exists (s_1, s_2) \in S \text{ such that } \gcd(b_1(x), \overline{s}_2(x)) \neq 1. \tag{6.9}$$

Complexity analysis. Let $s_1 = (x+1)f_i$ be the maximum degree polynomial of all pairs in S and the complexity of computing the greatest common divisor over elements in S be $O((q^{d+1})^2)$. Then the complexity for checking a parameter b with the above mechanism is $O(\#S(q^{d+1})^2)$. This complexity is an upper bound because in practice we see that a *gcd* in S requires a smaller number of operations.

At last, we summarize the mechanism in Algorithm 24. Note that, for determining whether a given b is a vulnerable parameter, the algorithm must be executed in all base field representations.

Algorithm 24 A mechanism for verifying the binary curve parameter b

Input: The element $b \in \mathbb{F}^*_{2^{2n}}$, the polynomial lists $b_1, \overline{s}_2$ obtained from the set S.
Output: *True* if the binary curve defined with a parameter b is vulnerable against the gGHS attack and *False* otherwise.

```
1: aux ← 1, $j \leftarrow 0$
2: while aux $= 1$ and $j < \#S$ do
3:     $j \leftarrow j + 1$
4:     aux ← $\gcd(b_1[j], \overline{s}_2[j])$
5: end while
6: if aux $\neq 1$ then
7:     return True
8: else
9:     return False
10: end if
```

6.5 A concrete attack on the GLS curve $E/\mathbb{F}_{2^{62}}$

In order to understand the practical implications of the GHS Weil descent algorithm over a binary GLS curve, we implemented a complete attack on a curve defined over the field $\mathbb{F}_{2^{31\cdot 2}}$. Such field was chosen for two reasons: (i) solving the DLP on the Jacobian of a hyperelliptic curve obtained by the GHS attack is easier then solving it on the elliptic curve (see Table 6.1); (ii) the small amount of resources required for solving the DLP in this curve allowed us to experiment with different approaches to the problem.

6.5.1 Building a vulnerable curve

Let $\mathbb{F}_{2^{2\cdot 31}}$ be an extension field of $\mathbb{F}_q$ with $q = 2^{2\cdot 31/n}$ where $n \in \{31, 62\}$. Then we can represent the field $\mathbb{F}_{2^{62}}$ as follows,

- $n = 62, q = 2$, $\mathbb{F}_{2^{62}} \cong \mathbb{F}_2[v]/f(v)$, with $f(v) = v^{62} + v^{29} + 1$.
- $n = 31, q = 2^2$, $\mathbb{F}_{2^2} \cong \mathbb{F}_2[z]/g(z)$, with $g(z) = z^2 + z + 1$.
  $\mathbb{F}_{2^{62}} \cong \mathbb{F}_{2^2}[u]/h(u)$, with $h(u) = u^{31} + u^3 + 1$

Also, let E be a binary GLS curve given by the following equation

$$E_{a,b}/\mathbb{F}_{2^{62}} : y^2 + xy = x^3 + ax^2 + b \qquad a \in \mathbb{F}_{2^{62}},\ b \in \mathbb{F}^*_{2^{31}}$$

Given that the parameter a can be chosen arbitrarily subject to the constraint $\mathrm{Tr}_{\mathbb{F}_{2^{62}}/\mathbb{F}_2}(a) = 1$, we chose $a = z^2$. The next step was to find a vulnerable parameter $b \in \mathbb{F}^*_{2^{31}}$ which defines a curve $E_{a,b}$ that is vulnerable against the gGHS attack. Moreover, to simulate a cryptographic environment, we must have $\#E_{a,b}(\mathbb{F}_{2^{62}}) = c \cdot r$, with small c and prime r.

Let $x^{31} + 1 = (x+1)f_1 \cdots f_6$. We have $\deg(f_i) = 5$. Then Tables 6.2 and 6.3 give us a list of polynomials that generate the vulnerable parameters $b = (\gamma_1\gamma_2)^2$.

Table 6.2: Polynomials Ord_{γ_i} which generate low-genus hyperelliptic curves for the case $n = 31,\ q = 2^2$

Ord_{γ_1}	Ord_{γ_2}	$\deg(\mathrm{Ord}_{\gamma_1})$	$\deg(\mathrm{Ord}_{\gamma_2})$	m	genus	E-G algorithm complexity
$(x+1)f_i$	$x+1$	6	1	6	32	26.46
f_i	$x+1$	5	1	6	31	25.93

Table 6.3: Polynomials Ord_{γ_i} which generate low-genus hyperelliptic curves for the case $n = 62,\ q = 2$

Ord_{γ_1}	Ord_{γ_2}	$\deg(\mathrm{Ord}_{\gamma_1})$	$\deg(\mathrm{Ord}_{\gamma_2})$	m	genus	E-G algorithm complexity
$(x+1)^2 f_i$	$x+1$	7	1	7	64	26.46

At first, we looked for vulnerable parameters $b \in \mathbb{F}^*_{2^{31}}$ by obtaining the roots of the polynomials listed in Tables 6.2 and 6.3. However, for all those b parameters, $\log_2 |r| < 52$. For that reason, we considered non-GLS vulnerable parameters $b \in \mathbb{F}^*_{2^{62}}$ for which $\log_2 |r| \geq 52$. As a result, 61 isogeny classes were found. Let L be the set of its group orders. Then, in a 20-core Intel Xeon E5-2658 2.40GHz, we executed for 70 hours an extensive search through all $b \in \mathbb{F}^*_{2^{31}}$ checking if $\#E_{a,b}(\mathbb{F}_{2^{62}}) \in L$. However, no isogenous curves were found and the extended GHS attack could not be carried out.

Next, under the setting ($n = 31,\ q = 2^2$), we chose the vulnerable parameter $b = u^{24} + u^{17} + u^{16} + u^{12} + u^5 + u^4 + u^3 + u + 1$, which allowed us to construct a group with order $\#E_{a,b}(\mathbb{F}_{2^{62}}) = 4611686014201959530$. The size of our subgroup of interest is of about 51 bits. In theory, solving the DLP on this subgroup through the

Pollard Rho method would take about 2^{26} steps, which is the same cost as solving the DLP with the GHS/Enge-Gaudry approach.

Finally, we created an order-r generator point $P \in E_{a,b}(\mathbb{F}_{2^{62}})$ with the Magma `Random` function:

$$\begin{aligned}P(x,y) =& (u^{30} + z^2u^{28} + zu^{27} + u^{26} + zu^{25} + zu^{24} + u^{23} + z^2u^{20} + u^{18} + zu^{17} \\ &+ zu^{16} + u^{15} + u^{12} + z^2u^{10} + zu^9 + z^2u^8 + u^7 + zu^6 + u^4 + u^2 \\ &+ z^2u + z, \\ &zu^{30} + z^2u^{29} + z^2u^{26} + z^2u^{25} + zu^{24} + u^{23} + z^2u^{22} + z^2u^{21} + zu^{20} \\ &+ u^{19} + zu^{18} + u^{17} + u^{15} + zu^{14} + zu^{13} + z^2u^{12} + z^2u^{10} + zu^9 + u^8 \\ &+ zu^7 + u^6 + u^2 + zu).\end{aligned}$$

The challenge Q was generated with the same function:

$$\begin{aligned}Q(x,y) =& (u^{29} + z^2u^{28} + u^{27} + u^{26} + z^2u^{25} + zu^{24} + u^{23} + zu^{22} + z^2u^{20} + z^2u^{17} \\ &+ z^2u^{16} + zu^{12} + u^{11} + zu^{10} + z^2u^9 + z^2u^8 + zu^7 + zu^6 + z^2u^5 + zu^4 + \\ &z^2u^2 + u + z^2, \\ &u^{30} + u^{29} + z^2u^{28} + u^{27} + zu^{26} + z^2u^{24} + zu^{22} + u^{21} + z^2u^{20} + z^2u^{19} \\ &+ zu^{18} + zu^{17} + zu^{15} + u^{14} + zu^{12} + z^2u^{11} + u^{10} + z^2u^9 + u^6 + u^5 \\ &+ z^2u^3 + z^2u^2 + z^2u + z).\end{aligned}$$

Then we constructed the following genus-32 hyperelliptic curve with the Weil descent method[1]:

$$\begin{aligned}H(\mathbb{F}_{2^2}):\ & y^2 + (z^2x^{32} + x^{16} + z^2x^8 + z^2x^2 + x)y = \\ & x^{65} + x^{64} + z^2x^{33} + zx^{32} + x^{17} + z^2x^{16} + x^8 + x^5 + x^4 + z^2x^3 + zx^2 + zx.\end{aligned}$$

The points P, Q were mapped to the $J_H(\mathbb{F}_{2^2})$, which generated the divisors D_P and D_Q.

6.5.2 Adapting the Enge-Gaudry Algorithm

To solve the DLP on $J_H(\mathbb{F}_q)$, with $q = 2^2$ and genus $g = 32$, we adapted the Enge-Gaudry algorithm by restricting the factor base size in order to balance the

[1]In this step, we used the function `WeilDescent` from Magma..

relation collection and the linear algebra phases. According to [46], we can balance the two phases by selecting the factor base degree as $m = \lceil \log_q L_{q^g}[\varrho] \rceil$ where $\varrho = \sqrt{\frac{1}{2} + \frac{1}{4\vartheta}} - \sqrt{\frac{1}{4\vartheta}}$ for some positive integer ϑ which complies with (i) $g \geq \vartheta \log q$ and (ii) $q \leq L_{q^g}[\frac{1}{\sqrt{\vartheta}}]$. Similarly to the Section 6.4, we chose the constant factors of the algorithm complexity to be one. For all values of ϑ that satisfy the restrictions (i) and (ii), we have $m = [4, 6]$.

However, in practice, we constructed the factor base dynamically. At first, we initialized our base $\mathcal{F}$ as an empty set and imposed a restriction so that $\mathcal{F}$ can contain polynomials up to degree m. Next, for each valid relation in the Enge-Gaudry algorithm, that is, when the polynomial U of a divisor $D = \mathrm{div}(U, V)$ is d-smooth, we included in $\mathcal{F}$ all irreducible factors of U which were not in $\mathcal{F}$. Finally, when the number of relations were equal to the number of factors in $\mathcal{F}$, we concluded the relations collection phase.

Experimentally, we saw that, at the end of the relations collection phase, just a portion of the irreducible polynomials of degree less or equal than m were included in $\mathcal{F}$. For that reason, in order to have approximately the same factor base size as if we had constructed a factor base with all irreducible polynomials of degree up to 6, we chose $m = 7$. The algorithm was executed within the Magma v2.20-2 system, in one core of a Intel Core i7-4700MQ 2.40GHz machine. The timings of each phase are presented below.

Table 6.4: Timings for the adapted Enge-Gaudry algorithm

Random walk initialization	3.00 s
Relations collection	284.52 s
Linear Algebra (Lanczos)	0.11 s

At the end of the relation collection phase, our factor basis had 1458 elements, which is 44.12% of the total number of irreducible polynomials of degree 7 and below. Although the algorithm phases were not balanced as expected, solving the linear algebra system was trivial, and we considered our degree selection satisfactory. Finally, the computed discrete logarithm is given as $\lambda = 2344450470259921$.

An analysis of the algorithm balance: the genus-32 case

In order to verify the theoretical balance of [46] in the context of the dynamic factor base construction, we executed the algorithm with different factor base degree limits. The results are presented in Table 6.5.

Table 6.5: Details of different Enge-Gaudry (E-G) algorithm settings ($g = 32$)

	Factor base maximum degree (d)							
	5	6	7	8	9	10	11	12
Relations collection phase								
Number of irreducible poly of degree $\leq d$ (α)	294	964	3304	11464	40584	145338	526638	1924378
Factor base size (β)	152	474	1458	4352	12980	34883	91793	214116
Ratio β/α	0.52	0.49	0.44	0.38	0.32	0.24	0.17	0.11
Theoretical cost (bits, ceiling) *	23	21	20	19	19	19	20	21
Average timing per relation (s)	20.25	1.47	0.20	0.05	0.02	0.01	0.01	0.01
Timing (s)	3078	646	284	220	252	413	909	2451
Original E-G timing estimation (s)	5953	1416	644	573	771	1744	4739	21168
Linear algebra phase								
Theoretical cost (bits, ceiling) *	17	21	24	27	30	33	35	38
Timing (s)	0.01	0.03	0.11	0.87	9.62	169	1288	6774

* The steps in the relations collection and the linear algebra phases have different costs. Since we do not have access to the Magma algebra system code, we could not give the exact timings of each step.

The theoretical costs for the relation collection phase were obtained by multiplying the inverse of the probability of having a d-smooth degree-32 polynomial by the factor base size. The linear algebra step theoretical cost was computed as the square of the factor base size multiplied by the average number of irreducible factors in each d-smooth degree-32 polynomial, which was calculated experimentally.

Here we can see that, regarding the theoretical costs, setting the factor base degree limit to 6 results in the most balanced implementation. However, the practical timings demonstrate against this assertion. This is because factorizing a degree-32 polynomial in $\mathbb{F}_{2^2}[x]$, which is the relations collection step[2], is more expensive than

[2]In fact, the cost of this step can be reduced by performing only a smoothness test instead of factorizing the polynomial. However, since we implemented the attack in Magma, we used the available function `Factorization`.

the linear algebra step.

On the other hand, if we consider practical timings, the degree-11 setting offers the most balanced version. However, it is clearly more important to have the lowest overall timings, which is achieved by the degree-8 setting. The results for the degree settings from 5 to 12 are presented in Figure 6.1.

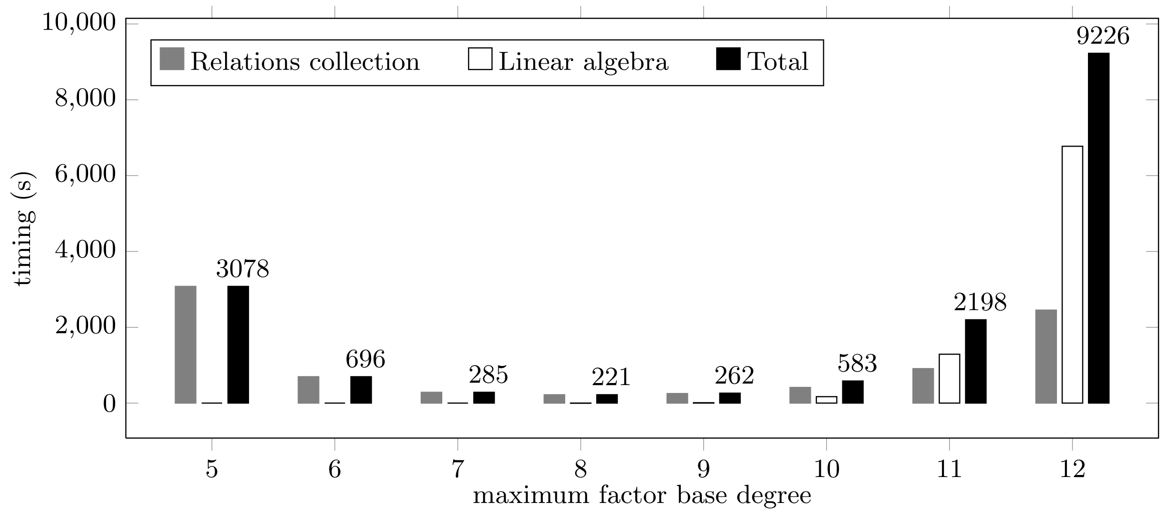

Figure 6.1: Timings for the Enge-Gaudry algorithm with dynamic factor base ($g = 32$)

The problem of balancing the Enge-Gaudry method with dynamic factor base is slightly different from the traditional algorithm. In the former, the cost of finding a valid relation and the ratio α/β (see Table 6.5) decreases as we increase the factor base degree limit. However, because of the larger number of irreducible polynomials, the probability of having relations with factors which are not included in our factor base increases. As a consequence, for each valid relation, more factors are added and the cost to achieve a matrix with the same number of columns and rows also increases. This effect is shown in the Figure 6.2.

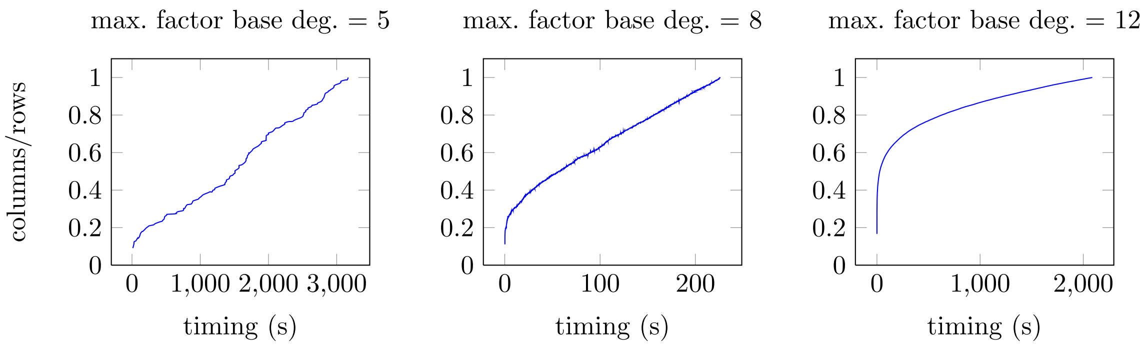

Figure 6.2: The ratio of the matrix columns (polynomials in the factor base) and rows (valid relations) per time. The relation collections phase ends when the ratio is equal to one ($g = 32$).

One possible solution for achieving a balanced algorithm is to restrict the size of the dynamic factor base. Ultimately, although unbalanced, constructing our factor base dynamically was useful in our context, since it allowed us to conclude the relations collection phase more efficiently when compared with the original Enge-Gaudry algorithm (see Table 6.5).

An analysis of the algorithm balance: the genus-45 case

We also analyzed the balance between the relations collection and the linear algebra phases of the dynamic-base Enge-Gaudry algorithm over a Jacobian of a hyperelliptic curve of genus 45 defined over $\mathbb{F}_{2^2}$. The subgroup of interest is of size $r = 2934347646102267239451433$ of approximately 81 bits.

After performing the theoretical balancing computations presented at the beginning of this section, we saw that our factor base should be composed of irreducible polynomials of degree up to $m = [5, 8]$. For that reason, we used this range as a reference for our factor base limit selection. The results are presented below.

Compared with the genus-32 case, we had a large number of factors per relation. As a result, more irreducible polynomials were added to the factor base, and consequently the relations collection phase became more costly. In addition, the ratios α/β were greater than the ones presented in the genus-32 example (see Table 6.5).

The most efficient configuration ($d = 10$) was unbalanced, the relations collection was about 36 times slower than the linear algebra phase. However, the genus-45 example provided a more balanced Enge-Gaudry algorithm, since the best setting for the genus-32 curve was unbalanced by a factor of 253. One possible reason is

Table 6.6: Details of different Enge-Gaudry algorithm settings ($g = 45$)

	Factor base maximum degree (d)						
	7	8	9	10	11	12	13
Relations collection phase							
Number of irreducible poly of degree $\leq d$ (α)	3304	11464	40584	145338	526638	1924378	7086598
Factor base size (β)	1626	5227	16808	52366	158226	460240	1268615
Ratio β/α	0.49	0.46	0.41	0.36	0.30	0.24	0.18
Theoretical cost (bits, ceiling) *	27	26	25	25	25	25	26
Average timing per relation (s)	29.45	4.60	1.05	0.29	0.12	0.07	0.09
Timing (s)	47895	24067	17621	15204	18909	32630	107902
Original E-G timing estimation (s)	97319	52780	42532	42148	62670	136631	602361
Linear algebra phase							
Theoretical cost (bits, ceiling) *	25	28	31	34	37	40	43
Timing (s)	0.62	3.79	39	421	4804	48661	417920

* The steps in the relations collection and the linear algebra phases have different costs. Since we do not have access to the Magma algebra system code, we could not give the exact timings of each step.

that, here, each linear algebra steps computed over operands of about 81 bits, which are 30 bits longer than the operands processed in the genus-32 linear algebra steps.

We expect that, for curves with larger genus, with respectively larger subgroups, a fully balanced configuration can be found. The results for each setting in the 45-genus example is shown in Figure 6.3.

In Figure 6.4, we show the progression of the ratio

$$\frac{\text{number of valid relations}}{\text{factor base size}}$$

during the relations collection phase. Similarly to the genus-32 case, for bigger d values, the rate of the factor base growth stalled the progress of the relations collection algorithm. Again, one potential solution to this issue is to impose limits on the factor base size.

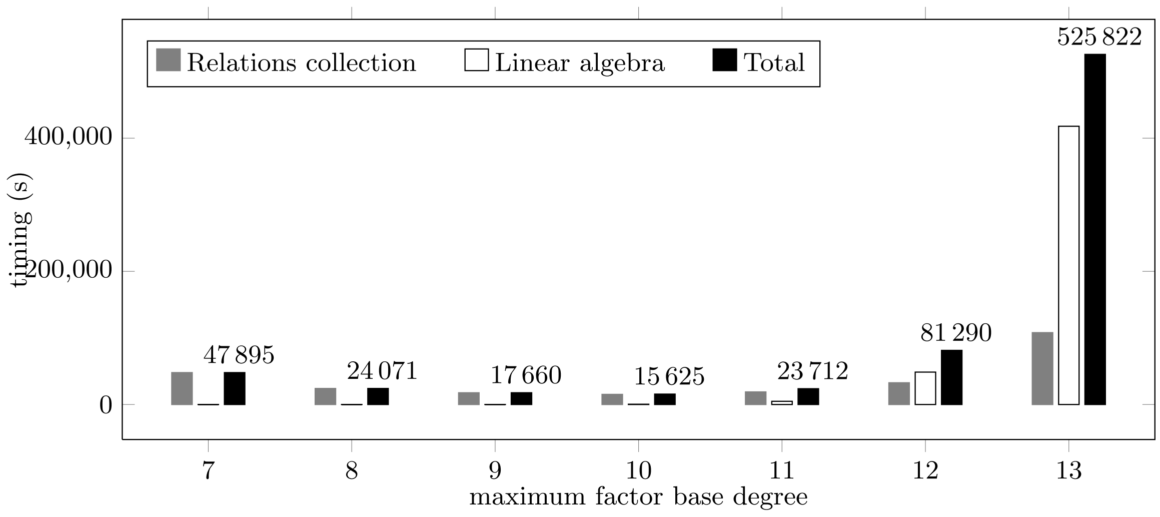

Figure 6.3: Timings for the Enge-Gaudry algorithm with dynamic factor base ($g = 45$)

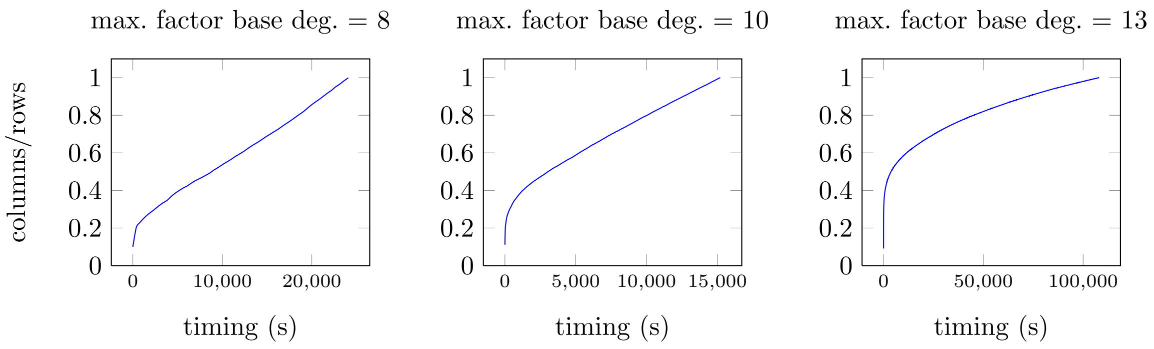

Figure 6.4: The ratio of the matrix columns (polynomials in the factor base) and rows (valid relations) per time. The relation collections phase ends when the ratio is equal to one ($g = 45$)

The challenge for obtaining an optimal relations collection phase is to find a balance between the average timing per relation and the factor base growth rate. The goal is to have a graph which, after the initial vertical rise, directs toward the ratio one as a linear function, such as the $d = 8, 10$ cases.

6.5.3 The Pollard Rho method

In order to verify the correspondence between the theoretical complexities and the practical results, we implemented the Pollard rho method which, for GLS curves $E/\mathbb{F}_{2^{2\cdot 31}}$, requires about 29.65 times the amount of work to solve the DLP with the GHS/Enge-Gaudry approach (see Table 6.1).

Our Pollard rho random walk implementation was based on the r-adding walk (with $r = 100$) method proposed in [149] and on the Floyd's cycle-finding technique. The algorithm was also implemented on Magma and executed in eight cores of a Intel Xeon E5-2650 2.60GHz machine.

After computing 68880848 points, we found the discrete logarithm in 63.18 CPU hours. Each step of the algorithm, which includes two modular integer addition and one point addition, took about 0.026 seconds. Comparing the practical experiments of the Pollard rho and the GHS/Enge-Gaudry algorithms, we have that the latter is about 6329.23 times faster. Note that, both implementations could be possibly improved by implementing them in C and exploiting the computing resources of our particular architetcure.

6.6 Summary

Here, we presented an implementation of the GHS attack against a binary GLS curve defined over $\mathbb{F}_{2^{2\cdot 31}}$. The DLP was solved in the Jacobian of a genus-32 hyperelliptic curve over $\mathbb{F}_{2^2}$ with a modified version of the Enge-Gaudry algorithm. In this version, we used a dynamic factor base in the relations collection phase. As a result, this phase proved to be more than twice as fast as the factor base proposed in the original algorithm.

Moreover, we realized experiments to understand the dynamic factor base mechanism with different configurations. In order to strengthen our conclusion, we also provided experiments in the Jacobian of a genus-45 binary hyperelliptic curve.

Part III

Conclusion

7 | Final Discussions

In the subsequent sections we present a general analysis of the work presented in this thesis. We start by enumerating our main contributions and then examine the impact of them in the development of cryptography primitives constructed over small-characteristic fields. Finally, we consider the problems and challenges left for future research and the open possibilities for investigation in the technological and algorithmic areas.

7.1 Contributions

In the following paragraphs, we concisely specify our contributions in the area of *high-speed elliptic curve cryptography*,

- A new system of projective coordinates, denominated *lambda coordinates*, which provided state-of-the-art formulas for computing the point arithmetic in binary elliptic curves. In addition, its form $(x, \lambda = x + \frac{y}{x})$, is particularly appropriate for scalar multiplication methods based on the point halving operation, since it bypasses the point coordinates transformation overhead imposed by this operation.

- Design of efficient base ($\mathbb{F}_q$) and quadratic ($\mathbb{F}_{q^2}$) field arithmetic. More precisely, the implementation of fast arithmetic for the case $q = 2^{127}$ aimed for high-end desktops. This implementation exploits the 128-bit SSE/AVX vector set of instructions and the 64-bit carry-less multiplier, both technologies which are ubiquitous on current desktop platforms. Our efficient arithmetic, along with the lambda coordinate system formulas, supported the speed-record software implementation of a 128-bit secure scalar multiplication in a binary GLS curve.

- A novel right-to-left halve-and-add Montgomery ladder algorithm. For the first time, the point halving operation could be applied efficiently in a scalar multiplication algorithm based on the Montgomery ladder. In addition, the algorithm precomputes the points to be added throughout the main iteration. As a consequence, a major speed-up is expected in the fixed-point scenario.

- The first timing-resistant scalar multiplication algorithm designed for four-core platforms. To achieve this, we combined the double-and-add left-to-right (without precomputation) and the halve-and-add right-to-left (with precomputation) Montgomery ladder point multiplication approaches with the endomorphism provided by the GLS binary curves. As a result, we improved by 50% the one-core 128-bit secure Montgomery double-and-add algorithm implementation.

- A regular τ-adic recoding based on the work in [91]. This method was a necessary step for the implementation of the first timing-attack resistant scalar multiplication in Koblitz curves, which surpassed by 26% the fastest protected Montgomery ladder point multiplication implementation on Koblitz curves [25].

- The construction of a new family of Koblitz curves defined over a prime extension of the field $\mathbb{F}_4$. The arithmetic in quadratic fields is well-suited for implementation on concurrent computing architectures, such as the current desktops and some mobile devices. As a result, we achieved a speed record in protected 128-bit software scalar multiplication implementation on Koblitz curves.

- Finally, we developed base arithmetic fully based on redundant trinomials [45]. Our scalar multiplication in Koblitz curves over $\mathbb{F}_4$ was implemented using the trinomial $g(x) = x^{192} + x^{19} + 1$, which was carefully selected in order to provide efficient timings for the main base arithmetic operations: multiplication and squaring.

Next, we list our contributions in the area of the *discrete logarithm problem*,

- A concrete analysis of the impact of the recent discrete logarithm problem advances [86, 68, 13, 70, 89] on the pairing-friendly fields $\mathbb{F}_{3^{6\cdot 509}}$ and $\mathbb{F}_{3^{6\cdot 1429}}$. Our study was important to understand the computing costs for solving the DLP on fields of cryptographic importance. Also, it assisted the community in deciding whether those fields are secure to be employed in practical protocols.

- The practical application of the new methods for solving the DLP over small-characteristic fields to the cases $\mathbb{F}_{3^{6\cdot 137}}$ and $\mathbb{F}_{3^{6\cdot 163}}$. The field $\mathbb{F}_{3^{6\cdot 163}}$ was previously considered for implementing pairing-based protocols [26, 59]. Our computations were temporarily established as a record in breaking cryptographic-relevant fields.

- An implementation of the GHS attack against a GLS binary elliptic curve defined over $\mathbb{F}_{2^{2\cdot 31}}$ with the Magma algebraic system. The implementation showed us that, in practice, the GHS attack is more efficient than the Pollard Rho approach, even though both have approximately the same theoretical cost.

- The analysis of the cost of the Enge-Gaudry algorithm [46] for solving the discrete logarithm problem with a dynamic factor base. In this work, we realized experiments to determine the best factor base configuration in order to balance the relation collection and the linear algebra phases. Moreover, we showed how this optimal setting differs from the theoretical balance, which are based on asymptotic cost estimates.

7.1.1 Publications

The work presented in this document were published in the following papers,

- T. Oliveira, J. López, D. F. Aranha and F. Rodríguez-Henríquez. Lambda Coordinates for Binary Elliptic Curves. In *Cryptographic Hardware and Embedded System - CHES 2013*, volume 8086 of LNCS, pages 311 - 330. Springer Berlin Heidelberg, 2013.

- G. Adj, A. Menezes, T. Oliveira and F. Rodríguez-Henríquez. Weakness of $\mathbb{F}_{3^{6\cdot 509}}$ for Discrete Logarithm Cryptography. In *Pairing-Based Cryptography - Pairing 2013*, volume 8365 of LNCS, pages 20 - 44. Springer International Publishing, 2014.

- T. Oliveira, J. López, D. F. Aranha and F. Rodríguez-Henríquez. Two is the fastest prime: lambda coordinates for binary elliptic curves. *Journal of Cryptographic Engineering*, 4(1):3 - 17, 2014.

- T. Oliveira, J. López, D. F. Aranha and F. Rodríguez-Henríquez. Fast Point Multiplication Algorithms for Binary Elliptic Curves with and without Precomputation. In *Selected Areas in Cryptography - SAC 2014*, volume 8781 of LNCS, pages 324 - 344. Springer International Publishing, 2014.

- G. Adj, A. Menezes, T. Oliveira and F. Rodríguez-Henríquez. Computing Discrete Logarithms in $\mathbb{F}_{3^{6\cdot137}}$ and $\mathbb{F}_{3^{6\cdot163}}$ Using Magma. In *International Workshop on the Arithmetic of Finite Fields - WAIFI 2014*, volume 9061 of LNCS, pages 3 - 22. Springer International Publishing, 2014.
- G. Adj, A. Menezes, T. Oliveira and F. Rodríguez-Henríquez. Weakness of $\mathbb{F}_{3^{6\cdot1429}}$ and $\mathbb{F}_{2^{4\cdot3041}}$ for discrete logarithm cryptography. *Finite Fields and Their Applications*, 32:148 - 170, 2015.
- T. Oliveira and J. Chi. Attacking a Binary GLS Elliptic Curve with Magma. In *Progress in Cryptology - LATINCRYPT 2015*, volume 9230 of LNCS, pages 308 - 326. Springer International Publishing, 2015.

7.2 Advances

Recently, a number of new approaches for solving the discrete logarithm problem in elliptic curves (ECDLP) were proposed. Those advances, concisely discussed in [58], brought doubt and mistrust to the practical use of binary curves in the academic community [135]. However, the same authors in [58] concluded their survey with the following statement,

> *The current situation, not at all definitive, is that there is no consensus whether there is a subexponential algorithm for ECDLP in characteristic 2.*

Considering this current scenario, our work proposed, through new technological and algorithmic ideas, alternative methods to compute, in software, efficient and secure scalar multiplications on binary curves. Our result showed that the binary curves are highly competitive with the mainstream approaches on prime curves.

In the discrete logarithm problem (DLP) area, our theoretical analysis of the impact of the recently proposed algorithms to solve the DLP on small-characteristic fields in pairing-based fields helped to clarify the range of fields which are suitable for the secure instantitation of pairing-based protocols. Moreover, our implementation of the attacks against fields that were previously proposed in the literature depicted the potential of the recent approaches.

Finally, our work on the DLP on binary GLS curves provided a better comprehension of the practical implications of the gGHS attack on this novel family of curves. In addition, we presented, for the first time, a concrete analysis of the effect of the dynamic factor base to the Enge-Gaudry algorithm for solving the DLP on a hyperelliptic curve.

7.3 Future work

7.3.1 Open questions

In this section, we list the open problems related to our work. The problems are not presented in any order of significance, but grouped by the two main subjects of our thesis.

Are there more efficient approaches to compute the inversion operation in binary fields? After the publication, in 1988, of the Itoh-Tsujii algorithm [84] for inversion in binary fields[1], there wasn't any major advances to improve the efficiency of this operation. The continuous decrease of the carry-less multiplier latency and throughput emphasizes even more the high cost of the inversion function. For instance, in the Haswell architecture, the relation between the inversion and multiplication costs (in clock cycles) is more than thirty (see Section 4.2.5).

An efficient inversion operation would open more possibilities in the implementation of the scalar multiplication algorithm. For example, one could choose to represent the points with affine coordinates, whose arithmetic are based on inversions and, consequently, reduce the required amount of memory throughout the main iteration of the algorithm.

Is the Karatsuba method faster than the schoolbook algorithm? Asymptotically, the Karatsuba technique computational complexity of $O(n^{log_2 3})$ is better than the $O(n^2)$ complexity of the schoolbook approach. However, in practice, the cost of pre- and post-processing the operands is higher in the Karatsuba algorithm. In addition, the cost of multiplying two 64-bit operands is approaching the cost of the basic logical operations, which are used in the peripheral phases of the Karatsuba method. The evolution of these instructions costs is shown in Table 7.1.

As a result, when multiplying polynomials stored in a few words (i.e. two or three), the schoolbook algorithm may be faster. Let us analyze the two-word case. For the Karatsuba method we need three multiplications and twelve logical instructions, while the schoolbook algorithm requires four multiplications and five logical instructions. If we have a scenario where one 64-bit carry-less multiplication costs less than seven logical operations, then the schoolbook method is more efficient.

[1]As the title of the article suggests, the method was devised for finite fields elements represented by a normal basis, where the squaring operation is computed by low-cost circular shifts. In the case of the polynomial representation, it is too much expensive to compute the algorithm through multiple squarings. As a consequence, precomputed tables are required (see Section 3.1.3).

Table 7.1: Comparison of the costs (in clock cycles) of the carry-less multiplier with the logical operations (computed with 128-bit SSE/AVX instructions) in different computer architectures

Architecture	Carry-less multiplier		Logical operations	
	Latency	Throughput	Latency	Throughput
Westmere, 2010	14	8	1	0.33
Sandy Bridge, 2011	14	8	1	0.33
Ivy Bridge, 2012	14	8	1	0.33
Haswell, 2013	7	2	1	0.33
Broadwell, 2014	5	1	1	1
Skylake, 2015	7	1	1	0.33

The costs of the Westmere, Sandy Bridge, Ivy Bridge and Haswell architectures were taken from [130]. For the Broadwell and Skylake machines, the timings were based on [127].

Considering only the latency, we have that the Broadwell architecture fulfills these requirements. In practice, experiments must be carried out, since the instruction throughput is also a determining factor.

Is the shift-and-add method for computing the modular reduction faster than its alternative, the mul-and-add technique? This is another problem related to the decrease of the latency and throughput of the carry-less multiplication instruction. However, it is more complex to analyze, since it depends on many factors such as the size of the field and the form of the irreducible polynomial which is used to construct the field.

If the answer to this problem is affirmative, an interesting result follows: there would be no need to search for irreducible trinomials or pentanomials (or redundant trinomials) anymore. The only requirement (for the sake of efficiency) is that one is given a degree-d irreducible polynomial $f(x) = x^d + x^a + \ldots + 1$ with $(d - a) > W$, where W is the architecture word size.

Do other families of binary elliptic curves with efficiently-computable endomorphisms exist? When compared with prime elliptic curves, there are a few families of binary curves which are provided with efficient endomorphisms [99, 76]. If more such families were found, we would have more options to provide an efficient point multiplication implementation. Besides, we could combine the families to extend the dimension of the GLV decomposition, as done with prime curves [29, 50, 43] and, consequently, accelerate the scalar multiplication computation.

Is it possible to compute the point halving operation in projective coordinates? Computing a timing-resistant right-to-left halve-and-add scalar multiplication is not feasible in practice. This is because the right-to-left approach implies a higher overhead in order to protect the point multiplication. More precisely, we must apply the linear pass function twice on the accumulator points (see Sections 3.3 and 4.2.4). When the NAF-width w value is big (e.g. four or five), this overhead becomes too high.

However, to perform the left-to-right approach efficiently, we need that the accumulator point be represented in projective coordinates throughout the main point multiplication loop. If we could devise a method to perform the point halving in projective coordinates which was faster than the projective point doubling operation, we could improve the efficiency of the protected scalar multiplication in binary curves.

Can we apply the GLV decomposition method with the Montgomery ladder algorithm? In Section 3.4, we presented a method combining the GLV approach with the Montgomery ladder in order to compute the scalar multiplication in the multi-core scenario. However, for a one-core architecture, it is not known how to maintain the Montgomery ladder point difference together with the applications of a non-trivial endomorphism.

If the same speed-ups obtained in the GLV approach were achieved in the Montgomery ladder algorithm, we would have the fastest timing-resistant scalar multiplication approach.

Is it faster to solve the DLP over small-characteristic fields with FPGAs? It would be useful to analyze the performance of FPGAs to perform the first descent phases (i.e. continuous-fraction and classical) of the DLP algorithms. Those phases consist of multiple instances of simple functions, such as the polynomial multiplication and smoothness testing, which are fully independent. As a result, the implementation of these algorithms in FPGAs could be faster than executing them in desktop machines that are embedded with just four to sixteen cores.

Can we use GPUs to accelerate the relations collection phase of the Enge-Gaudry algorithm? The relations collection phase is one of the most crucial steps of the Enge-Gaudry method for solving the DLP in hyperelliptic curves. If we improve the efficiency for collecting a relation, we could reduce the factor base degree and, consequently, would have a smaller set of equations to be solved in the subsequent linear algebra step.

The challenge is to devise an implementation for adding divisors which is suitable for efficient implementation in many GPU threads. Also, this implementation must require a minimum amount of memory, in order to extend the GPU parallelism.

Is there a polynomial-time algorithm to solve the DLP over small-characteristic fields? After the recent outstanding advances in solving the DLP over small-characteristic fields, which included a quasi-polynomial algorithm [13], the expectations for finding a polynomial-time approach in the near future are high. However, one questions whether another framework will be required for such an achievement. In practice, we must devise an efficient descent method which outpaces the one based on the complex Gröbner basis algorithm for solving bilinear equations.

7.3.2 Further possibilities

Next, we present different subjects that could complement and advance the work developed in this thesis.

Implementation of the GLS and Koblitz curves scalar multiplication in different architectures and scenarios. In this thesis, we presented point multiplication implementations focused on high-end desktop architectures. However, it is worthwhile to analyze the efficiency of those curves in constrained platforms, where resources such as memory and power are scarce, the instruction set is simpler and the register size is smaller. On the other hand, architectures that provides a higher level of parallelism, such as GPUs, could also be explored.

In addition, the fixed-point setting should also be analyzed, since it is part of elliptic curve-based digital signatures protocols [132].

Instruction-level parallelism in the point multiplication implementations. The opportunities for applying the 256-bit AVX instruction set and its future 512-bit extension (AVX-512) to scalar multiplication implementations could be studied. The larger 256-bit registers could be used to generate an efficient one-step shift-and-add modular reduction. Also, the same techniques used to implement a 128-bit base arithmetic could be applied to higher security levels by using larger vectorized registers.

Alternative methods to avoid side-channel attacks based on the CPU cache. As we increase the NAF w-width, we also increase the pre- and post-computation overhead. At the same time, we save point additions in the scalar multiplication main loop. The bound for selecting the value of w lies on the balance between these two phases[2]. For timing attack-resistant implementations, the w value also determines the not-negligible linear pass function cost, which increases with the precomputed points.

An alternative method that mitigates the impact of the CPU cache timing-attack protection would allow the expansion of the NAF-width w and the efficiency improvement of the scalar multiplication.

The impact of the side-channel attacks in multi-core architectures. The multiple cores currently available in desktops and mobile devices can be used to accelerate the scalar multiplication computations (see Sections 3.3.5 and 3.4). However, there are few studies [154] on the vulnerability of cryptographic implementations in these architectures.

Binary elliptic curves with higher security levels. According to a recent NSA statement on cryptographic algorithms, documents classified as "TOP SECRET", which is the maximum U.S. government security level, should be signed with 384-bit elliptic curves (192-bit security level) [133]. More precisely, with the standard NIST P-384. As a result, we must expect, in the near future, a greater demand for scalar multiplication implementations with more than 128 bits of security.

For binary curves, we have different options for implementing an efficient high-security level point multiplication. For 256 bits of security, one could select a GLS curve defined over $\mathbb{F}_{2^{2\cdot 251}}$ or $\mathbb{F}_{2^{2\cdot 257}}$ or a Koblitz curve defined over the field $\mathbb{F}_{2^{571}}$. Both curves require cost estimations, new arithmetic techniques and a crafty implementation in order to be considered proper to be used in practice.

Implementation of algorithms for computing Gröbner basis. The Gröbner basis descent is one of the most crucial phases of the small-characteristic field DLP algorithms. This is because it determines the smoothness bound that the previous descent phases must reach. Presently, the Magma algebra system [31] holds one of the most popular [3, 70] state-of-the-art implementation of the Faugère's F4 algorithm [47], which is used to compute the Gröbner basis.

[2]This statement applies only to architectures that do not have any memory restraints, such as high-end desktops. For constrained devices, one also must care about the required amount of memory when pre- or post-processing the points.

Given that the Magma code is not public, we do not know whether optimizations for a particular small-characteristic field could be applied. As a result, an efficient open-source implementation of the F4 (or F5 [48]) algorithm would allow the general use of a Gröbner basis computation technique and possibly extend the reach of the recent small-characteristic DLP solver methods.

Solving the DLP on higher extensions of pairing-related fields The analysis presented in [4, 3] showed that solving the DLP on large-extension fields such as $\mathbb{F}_{3^{6 \cdot 1429}}$ and $\mathbb{F}_{2^{4 \cdot 3041}}$ is infeasible in practice. Although theoretically broken, those fields are constructed with a large prime extension, which makes the descent phase too costly and still dependent of the expensive QPA step. Devising methods to reduce the cost of the DLP on those fields can possibly give insights for further asymptotic reductions in the general algorithm for solving the DLP on small-characteristic fields.

One of the last remaining fields that was considered for pairing-based cryptography but hasn't been practically broken yet is $\mathbb{F}_{2^{12 \cdot 439}}$, which is the embedded field of a genus two supersingular curve [34]. Here, we could analyze the applicability of the new methods [71, 89] to improve the efficiency of the DLP algorithms for small-characteristic fields.

Solving the DLP on a weak GLS curve defined over $\mathbb{F}_{2^{2 \cdot 127}}$. According to the authors in [76], solving the DLP over a GLS curve $E/\mathbb{F}_{2^{2 \cdot 127}}$ defined with vulnerable parameters against the generalized/extended GHS attack (see Section 6.3) has a cost of 52 bits. Roughly, a current high-end desktop can process about 2^{44} relation generation steps in one month[3].

As a consequence, we would need about $2^8 = 256$ machines to compute the relations collection phase in one month. This number can be reduced if we consider optimizations in the divisor arithmetic functions, the CPU/GPU parallelism (see Section 7.3.1) or relaxing the deadline by two or more months.

Solving the DLP on such GLS elliptic curves would show, in practice, the potential of the GHS attack against a curve that provides about 128 bits of security with the traditional Pollard rho method. Also, it would give us the expertise for implementing the Enge-Gaudry algorithm in a large scale.

[3]In small-characteristic field DLP algorithms, the relations collection step is less costly when compared with the hyperelliptic DLP solvers. Experimentally, in an Intel Core i7-4700MQ machine, we verified that a desktop can process 2^{46} steps of the former algorithm in one month. Considering that the latter algorithm step is four times more expensive, we estimated the cost as 2^{44}.

Binary elliptic curves defined over $\mathbb{F}_8$. Defining a binary elliptic curve over $\mathbb{F}_8$ would allow us to develop fast arithmetic, which can take profit from the internal parallelism of the current commercial desktops. However, it is necessary to perform an analysis of its security against attacks based on the Weil restriction. As stated by the authors in [109], the field $\mathbb{F}_{2^3}$ is "partially weak", which means that "only a non-negligible proportion of all elliptic curves over [this field] can be solved significantly faster that it takes Pollard's rho method".

As a result, one could determine this non-negligible proportion and find secure fields of cryptographic interest for implementing efficient scalar multiplication algorithms.

Practical generalized GHS attack Given that the curve C over k generated by the generalized GHS method is in general not hyperelliptic, the effectiveness of this attack depends on that the addition operation in $J_C(k)$ is similar to this operation in hyperelliptic curves and that the cost of solving the DLP in $J_C(k)$ is the same of solving it in a hyperelliptic curve with the Enge-Gaudry algorithm [110].

An interesting line of research would be to analyze the feasibility of the gGHS attack against elliptic curves currently used in real-world protocols, verifying how do the above assumptions hold in practice.

Bibliography

[1] G. Adj, A. Menezes, T. Oliveira, and F. Rodríguez-Henríquez. Weakness of $\mathbb{F}_{3^{6 \cdot 1429}}$ and $\mathbb{F}_{2^{4 \cdot 3041}}$ for Discrete Logarithm Cryptography. Cryptology ePrint Archive, Report 2013/737, 2013. http://eprint.iacr.org/.

[2] G. Adj, A. Menezes, T. Oliveira, and F. Rodríguez-Henríquez. Weakness of $\mathbb{F}_{3^{6 \cdot 509}}$ for Discrete Logarithm Cryptography. In *Pairing-Based Cryptography - Pairing 2013*, volume 8365 of *LNCS*, pages 20 – 44. Springer International Publishing, 2014.

[3] G. Adj, A. Menezes, T. Oliveira, and F. Rodríguez-Henríquez. Computing Discrete Logarithms in $\mathbb{F}_{3^{6 \cdot 137}}$ and $\mathbb{F}_{3^{6 \cdot 163}}$ Using Magma. In *Arithmetic of Finite Fields*, volume 9061 of *LNCS*, pages 3 – 22. Springer International Publishing, 2015.

[4] G. Adj, A. Menezes, T. Oliveira, and F. Rodríguez-Henríquez. Weakness of $\mathbb{F}_{3^{6 \cdot 1429}}$ and $\mathbb{F}_{2^{4 \cdot 3041}}$ for discrete logarithm cryptography. *Finite Fields and Their Applications*, 32:148 – 170, 2015.

[5] G. B. Agnew, R. C. Mullin, and S. A. Vanstone. An implementation of elliptic curve cryptosystems over $\mathbb{F}_{2^{155}}$. *IEEE Selected Areas in Communications*, 11(5):804–813, 1993.

[6] O. Ahmadi, D. Hankerson, and A. Menezes. Software Implementation of Arithmetic in $\mathbb{F}_{3^m}$. In *Arithmetic of Finite Fields*, volume 4547 of *LNCS*, pages 85–102. Springer Berlin Heidelberg, 2007.

[7] O. Ahmadi, D. Hankerson, and F. Rodríguez-Henríquez. Parallel Formulations of Scalar Multiplication on Koblitz Curves. *Journal of Universal Computer Science*, 14(3):481–504, 2008.

[8] E. Al-Daoud, R. Mahmod, M. Rushdan, and A. Kilicman. A new addition formula for elliptic curves over $GF(2^n)$. *IEEE Transactions on Computers*, 51(8):972–975, 2002.

[9] D. F. Aranha, A. Faz-Hernández, J. López, and F. Rodríguez-Henríquez. Faster Implementation of Scalar Multiplication on Koblitz Curves. In *Progress in Cryptology - LATINCRYPT 2012*, volume 7533 of *LNCS*, pages 177–193. Springer Berlin Heidelberg, 2012.

[10] D. F. Aranha, J. López, and D. Hankerson. Efficient Software Implementation of Binary Field Arithmetic Using Vector Instruction Sets. In *Progress in Cryptology - LATINCRYPT 2010*, volume 6212 of *LNCS*, pages 144–161. Springer Berlin Heidelberg, 2010.

[11] R. M. Avanzi, M. Ciet, and F. Sica. Faster Scalar Multiplication on Koblitz Curves Combining Point Halving with the Frobenius Endomorphism. In *Public Key Cryptography - PKC 2004*, volume 2947 of *LNCS*, pages 28–40. Springer Berlin Heidelberg, 2004.

[12] R. Barbulescu, C. Bouvier, J. Detrey, P. Gaudry, H. Jeljeli, E. Thomé, M. Videau, and P. Zimmermann. Discrete logarithm in GF(2^{809}) with FFS. Cryptology ePrint Archive, Report 2013/197, 2013. http://eprint.iacr.org/.

[13] R. Barbulescu, P. Gaudry, A. Joux, and E. Thomé. A Heuristic Quasi-Polynomial Algorithm for Discrete Logarithm in Finite Fields of Small Characteristic. In *Advances in Cryptology - EUROCRYPT 2014*, volume 8441 of *LNCS*, pages 1–16. Springer Berlin Heidelberg, 2014.

[14] P. S. L. M. Barreto, S. D. Galbraith, C. hÉigeartaigh, and M. Scott. Efficient pairing computation on supersingular Abelian varieties. *Designs, Codes and Cryptography*, 42(3):239–271, 2007.

[15] P. S. L. M. Barreto, H. Y. Kim, B. Lynn, and M. Scott. Efficient Algorithms for Pairing-Based Cryptosystems. In *Advances in Cryptology - CRYPTO 2002*, volume 2442 of *LNCS*, pages 354–369. Springer Berlin Heidelberg, 2002.

[16] D. J. Bernstein. Curve25519: New Diffie-Hellman Speed Records. In *Public Key Cryptography - PKC 2006*, volume 3958 of *LNCS*, pages 207–228. Springer Berlin Heidelberg, 2006.

[17] D. J. Bernstein, C. Chuengsatiansup, T. Lange, and P. Schwabe. Kummer strikes back: new DH speed records. Cryptology ePrint Archive, Report 2014/134, 2014. http://eprint.iacr.org/.

[18] D. J. Bernstein and T. L. (editors). eBACS: ECRYPT Benchmarking of Cryptographic Systems, Accessed: May 12 2014. http://bench.cr.yp.to.

[19] D. J. Bernstein and T. Lange. Explicit-Formulas Database. http://www.hyperelliptic.org/EFD.

[20] D. J. Bernstein and T. Lange. Security dangers of the NIST curves, 2013. Invited talk, International State of the Art Cryptography Workshop.

[21] D. J. Bernstein, T. Lange, and R. Farashahi. Binary Edwards Curves. In *Cryptographic Hardware and Embedded Systems - CHES 2008*, volume 5154 of *LNCS*, pages 244–265. Springer Berlin Heidelberg, 2008.

[22] J. L. Beuchat, J. Detrey, N. Estibals, E. Okamoto, and F. Rodríguez-Henríquez. Fast Architectures for the η_T Pairing over Small-Characteristic Supersingular Elliptic Curves. *IEEE Transactions on Computers*, 60(2):266–281, 2011.

[23] J.-L. Beuchat, J. Detrey, N. Estibals, E. Okamoto, and F. Rodríguez Henríquez. Fast Architectures for the η_T Pairing over Small-Characteristic Supersingular Elliptic Curves. *IEEE Transactions on Computers*, 60(2):266 – 281, 2011.

[24] J. L. Beuchat, E. López-Trejo, L. Martínez-Ramos, S. Mitsunari, and F. Rodríguez-Henríquez. Multi-core Implementation of the Tate Pairing over Supersingular Elliptic Curves. In *Cryptology and Network Security*, volume 5888 of *LNCS*, pages 413–432. Springer Berlin Heidelberg, 2009.

[25] M. Bluhm and S. Gueron. Fast Software Implementation of Binary Elliptic Curve Cryptography. Cryptology ePrint Archive, Report 2013/741, 2013. http://eprint.iacr.org/.

[26] D. Boneh, B. Lynn, and H. Shacham. Short Signatures from the Weil Pairing. *Journal of Cryptology*, 17(4):297–319, 2004.

[27] J. W. Bos, C. Costello, H. Hisil, and K. Lauter. Two is Greater than One. Cryptology ePrint Archive, Report 2012/670, 2012. http://eprint.iacr.org/.

[28] J. W. Bos, C. Costello, H. Hisil, and K. Lauter. Fast Cryptography in Genus 2. In *Advances in Cryptology - EUROCRYPT 2013*, volume 7881 of *LNCS*, pages 194 – 210. Springer Berlin Heidelberg, 2013.

[29] J. W. Bos, C. Costello, H. Hisil, and K. Lauter. High-Performance Scalar Multiplication Using 8-Dimensional GLV/GLS Decomposition. In *Cryptographic Hardware and Embedded Systems - CHES 2013*, volume 8086 of *LNCS*, pages 331 – 348. Springer Berlin Heidelberg, 2013.

[30] J. W. Bos, T. Kleinjung, R. Niederhagen, and P. Schwabe. ECC2K-130 on Cell CPUs. In *Progress in Cryptology - AFRICACRYPT 2010*, volume 6055 of *LNCS*, pages 225–242. Springer Berlin Heidelberg, 2010.

[31] W. Bosma, J. Cannon, and C. Playoust. The Magma algebra system. I. The user language. *Journal of Symbolic Computation*, 24(3 - 4):235 – 265, 1997.

[32] W. Bosma and H. Lenstra. Complete Systems of Two Addition Laws for Elliptic Curves. *Journal of Number Theory*, 53(2):229 – 240, 1995.

[33] R. Brent and P. Zimmerman. Algorithms for finding almost irreducible and almost primitive trinomials. Proceedings of a conference in honour of Professor H.C. Williams, 2003. http://web.comlab.ox.ac.uk/oucl/work/richard.brent/pd/rpb212.pdf.

[34] S. Chatterjee, D. Hankerson, and A. Menezes. On the Efficiency and Security of Pairing-Based Protocols in the Type 1 and Type 4 Settings. In *Arithmetic of Finite Fields*, volume 6087 of *LNCS*, pages 114–134. Springer Berlin Heidelberg, 2010.

[35] S. Chatterjee, K. Karabina, and A. Menezes. A New Protocol for the Nearby Friend Problem. In *Cryptography and Coding*, volume 5921 of *LNCS*, pages 236–251. Springer Berlin Heidelberg, 2009.

[36] J.-J. Chi and T. Oliveira. Attacking a Binary GLS Elliptic Curve with Magma. In *Progress in Cryptology - LATINCRYPT 2015*, volume 9230 of *LNCS*, pages 308 – 326. Springer International Publishing, 2015.

[37] 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*, 7(4):385–434, 1986.

[38] N. M. Clift. Calculating optimal addition chains. *Computing*, 91(3):265 – 284, 2011.

[39] H. Cohen, G. Frey, R. Avanzi, C. Doche, T. Lange, K. Nguyen, and F. Vercauteren. *Handbook of Elliptic and Hyperelliptic Curve Cryptography.* Chapman & Hall/CRC, Second edition, 2012.

[40] D. Coppersmith. Fast evaluation of logarithms in fields of characteristic two. *IEEE Transactions on Information Theory*, 30(4):587–594, 1984.

[41] D. Coppersmith. Solving Homogeneous Linear Equations Over $GF(2)$ via Block Wiedemann Algorithm. *AMS Mathematics of Computation*, 62(205):333–350, 1994.

[42] C. Costello, H. Hisil, and B. Smith. Faster Compact Diffie-Hellman: Endomorphisms on the x-line. In *Advances in Cryptology - EUROCRYPT 2014*, volume 8441 of *LNCS*, pages 183 – 200. Springer Berlin Heidelberg, 2014.

[43] C. Costello and P. Longa. FourQ: four-dimensional decompositions on a Q-curve over the Mersenne prime. Cryptology ePrint Archive, Report 2015/565, 2015. http://eprint.iacr.org/.

[44] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.2, 2008. http://www.ietf.org/rfc/rfc5246.txt.

[45] C. Doche. Redundant Trinomials for Finite Fields of Characteristic 2. In *Information Security and Privacy*, volume 3574 of *LNCS*, pages 122 – 133. Springer Berlin Heidelberg, 2005.

[46] A. Enge and P. Gaudry. A general framework for subexponential discrete logarithm algorithms. *Acta Arithmetica*, 102:83–103, 2002.

[47] J. C. Faugère. A new efficient algorithm for computing Gröbner bases (F4) . *Journal of Pure and Applied Algebra* , 139(1–3):61–88, 1999.

[48] J. C. Faugère. A New Efficient Algorithm for Computing Gröbner Bases Without Reduction to Zero (F5). In *Proceedings of the 2002 International Symposium on Symbolic and Algebraic Computation*, ISSAC '02, pages 75–83. ACM, 2002.

[49] J.-C. Faugère, L. Perret, C. Petit, and G. Renault. Improving the Complexity of Index Calculus Algorithms in Elliptic Curves over Binary Fields. In *Advances in Cryptology - EUROCRYPT 2012*, volume 7237 of *LNCS*, pages 27–44. Springer Berlin Heidelberg, 2012.

[50] A. Faz-Hernández, P. Longa, and A. H. Sánchez. Efficient and Secure Algorithms for GLV-Based Scalar Multiplication and Their Implementation on GLV-GLS Curves. In *Topics in Cryptology - CT-RSA 2014*, volume 8366 of *LNCS*, pages 1–27. Springer International Publishing, 2014.

[51] M. Firasta, M. Buxton, P. Jinbo, K. Nasri, and S. Kuo. Intel AVX: New Frontiers in Performance Improvements and Energy Efficiency. Technical report, Intel Corporation, May 2008. http://software.intel.com.

[52] A. Fog. Instruction Tables: List of Instruction Latencies, Throughputs and Micro-operation Breakdowns for Intel, AMD and VIA CPUs, Accessed: May 14 2014. http://www.agner.org/optimize/instruction_tables.pdf.

[53] K. Fong, D. Hankerson, J. López, and A. Menezes. Field inversion and point halving revisited. *IEEE Transactions on Computers*, 53(8):1047–1059, 2004.

[54] G. Frey. How to disguise an elliptic curve, 1998. http://www.cacr.math.uwaterloo.ca/conferences/1998/ecc98/frey.ps.

[55] G. Frey and H. G. Rück. A Remark Concerning m-Divisibility and the Discrete Logarithm in the Divisor Class Group of Curves. *AMS Mathematics of Computation*, 62(206):865 – 874, 1994.

[56] S. D. Galbraith. Supersingular Curves in Cryptography. In *Advances in Cryptology - ASIACRYPT 2001*, volume 2248 of *LNCS*, pages 495–513. Springer Berlin Heidelberg, 2001.

[57] S. D. Galbraith. *Mathematics of Public Key Cryptography.* Cambridge University Press, First edition, 2012.

[58] S. D. Galbraith and P. Gaudry. Recent progress on the elliptic curve discrete logarithm problem. Cryptology ePrint Archive, Report 2015/1022, 2015. http://eprint.iacr.org/.

[59] S. D. Galbraith, K. Harrison, and D. Soldera. Implementing the Tate Pairing. In *Algorithmic Number Theory*, volume 2369 of *LNCS*, pages 324–337. Springer Berlin Heidelberg, 2002.

[60] S. D. Galbraith, F. Hess, and N. P. Smart. Extending the GHS Weil Descent Attack. In *Advances in Cryptology - EUROCRYPT 2002*, volume 2332 of *LNCS*, pages 29–44. Springer Berlin Heidelberg, 2002.

[61] S. D. Galbraith, X. Lin, and M. Scott. Endomorphisms for Faster Elliptic Curve Cryptography on a Large Class of Curves. *Journal of Cryptology*, 24(3):446–469, 2011.

[62] S. D. Galbraith and N. P. Smart. A Cryptographic Application of Weil Descent. In *Cryptography and Coding*, volume 1746 of *LNCS*, pages 191–200. Springer Berlin Heidelberg, 1999.

[63] R. P. Gallant, R. J. Lambert, and S. A. Vanstone. Faster Point Multiplication on Elliptic Curves with Efficient Endomorphisms. In *Advances in Cryptology - CRYPTO 2001*, volume 2139 of *LNCS*, pages 190–200. Springer Berlin Heidelberg, 2001.

[64] P. Gaudry. An Algorithm for Solving the Discrete Log Problem on Hyperelliptic Curves. In *Advances in Cryptology - EUROCRYPT 2000*, volume 1807 of *LNCS*, pages 19–34. Springer Berlin Heidelberg, 2000.

[65] P. Gaudry, F. Hess, and N. P. Smart. Constructive and destructive facets of Weil descent on elliptic curves. *Journal of Cryptology*, 15(1):19–46, 2002.

[66] P. Gaudry, E. Thomé, N. Thériault, and C. Diem. A double large prime variation for small genus hyperelliptic index calculus. *Mathematics of Computation*, 76:475–492, 2007.

[67] F. Göloğlu, R. Granger, G. McGuire, and J. Zumbrägel. On the Function Field Sieve and the Impact of Higher Splitting Probabilities. In *Advances in Cryptology - CRYPTO 2013*, volume 8043 of *LNCS*, pages 109–128. Springer Berlin Heidelberg, 2013.

[68] F. Göloğlu, R. Granger, G. McGuire, and J. Zumbrägel. Solving a 6120-bit DLP on a Desktop Computer. In *Selected Areas in Cryptography - SAC 2013*, volume 8282 of *LNCS*, pages 136–152. Springer Berlin Heidelberg, 2014.

[69] R. Granger, T. Kleinjung, and J. Zumbrägel. Breaking '128-bit Secure' Supersingular Binary Curves (or how to solve discrete logarithms in $\mathbb{F}_{2^{4\cdot 1223}}$ and $\mathbb{F}_{2^{12\cdot 367}}$). Cryptology ePrint Archive, Report 2014/119, 2014. http://eprint.iacr.org/.

[70] R. Granger, T. Kleinjung, and J. Zumbrägel. Breaking '128-bit secure' supersingular binary curves (or how to solve discrete logarithms in $\mathcal{F}_{2^{4\cdot 1223}}$ and $\mathcal{F}_{2^{12\cdot 367}}$). In *Advances in Cryptology - CRYPTO 2014*, volume 8617 of *LNCS*, pages 126–145. Springer, 2014.

[71] R. Granger, T. Kleinjung, and J. Zumbrägel. On the Powers of 2. Cryptology ePrint Archive, Report 2014/300, 2014. http://eprint.iacr.org/.

[72] R. Granger, D. Page, and M. Stam. Hardware and software normal basis arithmetic for pairing-based cryptography in characteristic three. *IEEE Transactions on Computers*, 54(7):852–860, 2005.

[73] R. Granger, D. Page, and M. Stam. On Small Characteristic Algebraic Tori in Pairing-Based Cryptography. *LMS Journal of Computation and Mathematics*, 9:64–85, 2006.

[74] M. Hamburg. Fast and compact elliptic-curve cryptography. Cryptology ePrint Archive, Report 2012/309, 2012. http://eprint.iacr.org/.

[75] D. Hankerson, J. L. Hernandez, and A. Menezes. Software Implementation of Elliptic Curve Cryptography over Binary Fields. In *Cryptographic Hardware and Embedded Systems - CHES 2000*, volume 1965 of *LNCS*, pages 1–24. Springer Berlin Heidelberg, 2000.

[76] D. Hankerson, K. Karabina, and A. Menezes. Analyzing the Galbraith-Lin-Scott Point Multiplication Method for Elliptic Curves over Binary Fields. *IEEE Transactions on Computers*, 58(10):1411–1420, 2009.

[77] D. Hankerson, A. Menezes, and M. Scott. Software implementation of pairings. *Identity-Based Cryptography*, 2:188 – 206, 2008.

[78] D. Hankerson, A. Menezes, and S. Vanstone. *Guide to Elliptic Curve Cryptography.* Springer-Verlag New York, Inc., 2003.

[79] T. Hayashi, T. Shimoyama, N. Shinohara, and T. Takagi. Breaking Pairing-Based Cryptosystems Using ηT Pairing over $GF(3^{97})$. In *Advances in Cryptology - ASIACRYPT 2012*, volume 7658 of *LNCS*, pages 43 – 60. Springer Berlin Heidelberg, 2012.

[80] F. Hess. Generalising the GHS Attack on the Elliptic Curve Discrete Logarithm Problem. *LMS Computation and Mathematics*, 7:167–192, 2004.

[81] Z. Hu, P. Longa, and M. Xu. Implementing the 4-dimensional GLV method on GLS elliptic curves with j-invariant 0. *Designs, Codes and Cryptography*, 63(3):331–343, 2012.

[82] Y.-J. Huang, C. Petit, N. Shinohara, and T. Takagi. Improvement of Faugère et al.'s Method to Solve ECDLP. In *Advances in Information and Computer Security - IWSEC 2013*, volume 8231 of *LNCS*, pages 115–132. Springer Berlin Heidelberg, 2013.

[83] R. M. I. Blake, R. Fuji-Hara and S. Vanstone. Computing logarithms in finite fields of characteristic two. *SIAM Journal on Algebraic and Discrete Methods*, 5(2):276–285, 1984.

[84] T. Itoh and S. Tsujii. A fast algorithm for computing multiplicative inverses in GF(2^m) using normal bases. *Information and Computation*, 78(3):171–177, 1988.

[85] M. Jacobson, A. Menezes, and A. Stein. Solving Elliptic Curve Discrete Logarithm Problems Using Weil Descent. Cryptology ePrint Archive, Report 2001/041, 2001. http://eprint.iacr.org/.

[86] A. Joux. A new index calculus algorithm with complexity $L(1/4 + o(1))$ in very small characteristic. Cryptology ePrint Archive, Report 2013/095, 2013. http://eprint.iacr.org/.

[87] A. Joux. Discrete logarithm in $GF(2^{6168})$, 2013. Number Theory List (NMBRTHRY@LISTSERV.NODAK.EDU).

[88] A. Joux and R. Lercier. The Function Field Sieve in the Medium Prime Case. In *Advances in Cryptology - EUROCRYPT 2006*, volume 4004 of *LNCS*, pages 254–270. Springer Berlin Heidelberg, 2006.

[89] A. Joux and C. Pierrot. Improving the Polynomial time Precomputation of Frobenius Representation Discrete Logarithm Algorithms. In *Advances in Cryptology - ASIACRYPT 2014*, volume 8873 of *LNCS*, pages 378–397. Springer Berlin Heidelberg, 2014.

[90] A. Joux and V. Vitse. Cover and Decomposition Index Calculus on Elliptic Curves Made Practical. In *Advances in Cryptology - EUROCRYPT 2012*, volume 7237 of *LNCS*, pages 9–26. Springer Berlin Heidelberg, 2012.

[91] M. Joye and M. Tunstall. Exponent Recoding and Regular Exponentiation Algorithms. In *Progress in Cryptology - AFRICACRYPT 2009*, volume 5580 of *LNCS*, pages 334–349. Springer Berlin Heidelberg, 2009.

[92] D. Kim and S. Lim. Integer Decomposition for Fast Scalar Multiplication on Elliptic Curves. In *Selected Areas in Cryptography*, volume 2595 of *LNCS*, pages 13–20. Springer Berlin Heidelberg, 2003.

[93] K. H. Kim and S. I. Kim. A New Method for Speeding Up Arithmetic on Elliptic Curves over Binary Fields. Cryptology ePrint Archive, Report 2007/181, 2007. http://eprint.iacr.org/.

[94] B. King. An Improved Implementation of Elliptic Curves over $GF(2^n)$ when Using Projective Point Arithmetic. In *Selected Areas in Cryptography*, volume 2259 of *LNCS*, pages 134–150. Springer Berlin Heidelberg, 2001.

[95] E. Knudsen. Elliptic Scalar Multiplication Using Point Halving. In *Advances in Cryptology - ASIACRYPT 99*, volume 1716 of *LNCS*, pages 135–149. Springer Berlin Heidelberg, 1999.

[96] D. E. Knuth. *The Art of Computer Programming: Seminumerical Algorithms*, volume 2. Addison-Wesley Longman Publishing Co., Inc., Third edition, 1997.

[97] A. H. Koblitz, N. Koblitz, and A. Menezes. Elliptic curve cryptography: The serpentine course of a paradigm shift . *Journal of Number Theory* , 131(5):781 – 814, 2011.

[98] N. Koblitz. Elliptic Curve Cryptosystems. *Mathematics of Computation*, 48(177):203–209, 1987.

[99] N. Koblitz. CM-Curves with Good Cryptographic Properties. In *Advances in Cryptology - CRYPTO 91*, volume 576 of *LNCS*, pages 279–287. Springer Berlin Heidelberg, 1992.

[100] T. Lange. A note on López-Dahab coordinates. Cryptology ePrint Archive, Report 2004/323, 2006. http://eprint.iacr.org/.

[101] C. H. Lim and H. S. Hwang. Speeding Up Elliptic Scalar Multiplication with Precomputation. In *Information Security and Cryptology - ICISC 99*, volume 1787 of *LNCS*, pages 102–119. Springer Berlin Heidelberg, 2000.

[102] P. Longa and F. Sica. Four-Dimensional Gallant-Lambert-Vanstone Scalar Multiplication. In *Advances in Cryptology - ASIACRYPT 2012*, volume 7658 of *LNCS*, pages 718–739. Springer Berlin Heidelberg, 2012.

[103] P. Longa and F. Sica. Four-Dimensional Gallant-Lambert-Vanstone Scalar Multiplication. *Journal of Cryptology*, 27(2):248–283, 2014.

[104] J. López and R. Dahab. Fast Multiplication on Elliptic Curves over GF(2^m) without Precomputation. In *Cryptographic Hardware and Embedded Systems, CHES 99*, volume 1717 of *LNCS*, pages 316 – 327. Springer Berlin Heidelberg, 1999.

[105] J. López and R. Dahab. Improved Algorithms for Elliptic Curve Arithmetic in GF(2^n). In *Selected Areas in Cryptography*, volume 1556 of *LNCS*, pages 201–212. Springer Berlin Heidelberg, 1999.

[106] J. López and R. Dahab. An overview of elliptic curve cryptography. Technical Report IC-00-10, Institute of Computing, University of Campinas, 2000. http://www.ic.unicamp.br/reltech/2000/00-10.pdf.

[107] J. López and R. Dahab. New Point Compression Algorithms for Binary Curves. In *IEEE Information Theory Workshop, 2006*, pages 126–130, 2006.

[108] A. Menezes and M. Qu. Analysis of the Weil Descent Attack of Gaudry, Hess and Smart. In *Topics in Cryptology - CT-RSA 2001*, volume 2020 of *LNCS*, pages 308–318. Springer Berlin Heidelberg, 2001.

[109] A. Menezes and E. Teske. Cryptographic Implications of Hess' Generalized GHS Attack. Cryptology ePrint Archive, Report 2004/235, 2004. http://eprint.iacr.org/.

[110] A. Menezes and E. Teske. Cryptographic implications of Hess' generalized GHS attack. *Applicable Algebra in Engineering, Communication and Computing*, 16(6):439 – 460, 2005.

[111] A. Menezes, E. Teske, and A. Weng. Weak Fields for ECC. In *Topics in Cryptology - CT-RSA 2004*, volume 2964 of *LNCS*, pages 366–386. Springer Berlin Heidelberg, 2004.

[112] A. J. Menezes, T. Okamoto, and S. A. Vanstone. Reducing elliptic curve logarithms to logarithms in a finite field. *IEEE Transactions on Information Theory*, 39(5):1639–1646, 1993.

[113] V. S. Miller. Use of Elliptic Curves in Cryptography. In *Advances in Cryptology - CRYPTO 85*, volume 218 of *LNCS*, pages 417–426. Springer Berlin Heidelberg, 1986.

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

[115] K. i. Nagao. Decomposition Attack for the Jacobian of a Hyperelliptic Curve over an Extension Field. In *Algorithmic Number Theory*, volume 6197 of *LNCS*, pages 285–300. Springer Berlin Heidelberg, 2010.

[116] C. Nègre and J.-M. Robert. Impact of Optimized Field Operations AB,AC and AB + CD in Scalar Multiplication over Binary Elliptic Curve. In *Progress in Cryptology - AFRICACRYPT 2013*, volume 7918 of *LNCS*, pages 279–296. Springer Berlin Heidelberg, 2013.

[117] K. Okeya, T. Takagi, and C. Vuillaume. Efficient Representations on Koblitz Curves with Resistance to Side Channel Attacks. In *Information Security and Privacy*, volume 3574 of *LNCS*, pages 218–229. Springer Berlin Heidelberg, 2005.

[118] T. Oliveira, D. F. Aranha, J. López, and F. Rodríguez-Henríquez. Fast point multiplication algorithms for binary elliptic curves with and without precomputation. In *Selected Areas in Cryptography - SAC 2014*, volume 8781 of *LNCS*, pages 324–344. Springer International Publishing, 2014.

[119] T. Oliveira, J. López, D. F. Aranha, and F. Rodríguez-Henríquez. Lambda Coordinates for Binary Elliptic Curves. In *Cryptographic Hardware and Embedded Systems - CHES 2013*, volume 8086 of *LNCS*, pages 311–330. Springer Berlin Heidelberg, 2013.

[120] T. Oliveira, J. López, D. F. Aranha, and F. Rodríguez-Henríquez. Two is the fastest prime: lambda coordinates for binary elliptic curves. *Journal of Cryptographic Engineering*, 4(1):3–17, 2014.

[121] D. Page. Theoretical Use of Cache Memory as a Cryptanalytic Side-Channel. Cryptology ePrint Archive, Report 2002/169, 2002. http://eprint.iacr.org/.

[122] D. Page, N. P. Smart, and F. Vercauteren. A comparison of MNT curves and supersingular curves. *Applicable Algebra in Engineering, Communication and Computing*, 17(5):379–392, 2006.

[123] G. Paoloni. How to Benchmark Code Execution Times on Intel IA-32 and IA-64 Instruction Set Architectures. Technical report, Intel Corporation, 2010.

[124] Y. H. Park, S. Jeong, C. Kim, and J. Lim. An Alternate Decomposition of an Integer for Faster Point Multiplication on Certain Elliptic Curves. In *Public Key Cryptography*, volume 2274 of *LNCS*, pages 323–334. Springer Berlin Heidelberg, 2002.

[125] C. Petit and J.-J. Quisquater. On Polynomial Systems Arising from a Weil Descent. In *Advances in Cryptology - ASIACRYPT 2012*, volume 7658 of *LNCS*, pages 451–466. Springer, 2012.

[126] J. Pollard. Monte Carlo methods for Index Computation (mod p). *Mathematics of Computation*, 32:918–924, 1978.

[127] InstLatX64. x86, x64 Instruction Latency, Memory Latency and CPUID dumps, Accessed: 27 Nov 2015. http://instlatx64.atw.hu/.

[128] Intel Corporation. *Intel Architecture Instruction Set Extensions Programming Reference*, 2012. Reference Number: 319433-014. http://software.intel.com.

[129] Intel Corporation. Intel 64 and IA-32 Architectures Software Developer's Manual, 2015. 325462-056US.

[130] Intel Corporation. Intel Intrinsics Guide, Accessed: 27 Nov 2015. https://software.intel.com/sites/landingpage/IntrinsicsGuide/.

[131] National Institute of Standards and Technology. Recommended Elliptic Curves for Federal Government Use, 1999. http://csrc.nist.gov/csrc/fedstandards.html.

[132] National Institute of Standards and Technology. *FIPS PUB 186-4: Digital Signature Standard (DSS).* U.S. Department of Commerce, 2013.

[133] National Security Agency. Cryptography Today, August 2015. https://www.nsa.gov/ia/programs/suiteb_cryptography/.

[134] The Cunningham Project. http://homes.cerias.purdue.edu/ssw/cun/.

[135] The Internet Engineering Task Force. Crypto Forum Research Group Discussion Archive, 2015. http://www.ietf.org/mail-archive/web/cfrg/current/maillist.html.

[136] F. Rodríguez-Henríquez, G. Morales-Luna, and J. López. Low-Complexity Bit-Parallel Square Root Computation over $GF(2^m)$ for All Trinomials. *IEEE Transactions on Computers*, 57(4):472–480, 2008.

[137] P. Sarkar and S. Singh. A New Method for Decomposition in the Jacobian of Small Genus Hyperelliptic Curves. Cryptology ePrint Archive, Report 2014/815, 2014. http://eprint.iacr.org/.

[138] P. Sarkar and S. Singh. A Simple Method for Obtaining Relations Among Factor Basis Elements for Special Hyperelliptic Curves. Cryptology ePrint Archive, Report 2015/179, 2015. http://eprint.iacr.org/.

[139] R. Schroeppel. Cryptographic elliptic curve apparatus and method, 2000. US patent 2002/6490352 B1.

[140] R. Schroeppel. Elliptic curve point halving wins big. 2nd Midwest Arithmetical Geometry in Cryptography Workshop, 2000.

[141] R. Schroeppel. Automatically solving equations in finite fields, 2002. US patent 2002/0055962 A1.

[142] M. Scott. Optimal Irreducible Polynomials for GF(2^m) Arithmetic. Cryptology ePrint Archive, Report 2007/192, 2007. http://eprint.iacr.org/.

[143] M. Shantz and E. Teske. Solving the Elliptic Curve Discrete Logarithm Problem Using Semaev Polynomials, Weil Descent and Gröbner Basis Methods - an Experimental Study. Cryptology ePrint Archive, Report 2013/596, 2013. http://eprint.iacr.org/.

[144] N. Shinohara, T. Shimoyama, T. Hayashi, and T. Takagi. Key Length Estimation of Pairing-Based Cryptosystems Using η_T Pairing. In *Information Security Practice and Experience*, volume 7232 of *LNCS*, pages 228–244. Springer Berlin Heidelberg, 2012.

[145] J. A. Solinas. Efficient Arithmetic on Koblitz Curves. *Designs, Codes and Cryptography*, 19(2 - 3):195 – 249, 2000.

[146] D. Stebila. Elliptic Curve Algorithm Integration in the Secure Shell Transport Layer, 2009. http://www.ietf.org/rfc/rfc5656.txt.

[147] J. Tate. Endomorphisms of abelian varieties over finite fields. *Inventiones Mathematicae*, 2(2):134–144, 1966.

[148] J. Taverne, A. Faz-Hernández, D. F. Aranha, F. Rodríguez-Henríquez, D. Hankerson, and J. López. Speeding scalar multiplication over binary elliptic curves using the new carry-less multiplication instruction. *Journal of Cryptographic Engineering*, 1(3):187–199, 2011.

[149] E. Teske. Speeding up Pollard's rho method for computing discrete logarithms. In *Algorithmic Number Theory*, volume 1423 of *LNCS*, pages 541–554. Springer Berlin Heidelberg, 1998.

[150] W. R. Trost and G. Xu. On the Optimal Pre-Computation of Window τNAF for Koblitz Curves. Cryptology ePrint Archive, Report 2014/664, 2014. http://eprint.iacr.org/.

[151] Y. Tsunoo, E. Tsujihara, K. Minematsu, and H. Miyauchi. Cryptanalysis of Block Ciphers Implemented on Computers with Cache. In *International Symposium on Information Theory and Its Applications*, pages 803 – 806. IEEE Information Theory Society, 2002.

[152] L. C. Washington. *Elliptic Curves: Number Theory and Cryptography*. Chapman & Hall/CRC, Second edition, 2008.

[153] A. Weimerskirch and C. Paar. Generalizations of the Karatsuba Algorithm for Efficient Implementations. Cryptology ePrint Archive, Report 2006/224, 2006. `http://eprint.iacr.org/`.

[154] M. Weiß, B. Weggenmann, M. August, and G. Sigl. On Cache Timing Attacks Considering Multi-Core Aspects in Virtualized Embedded Systems. In *6th International Conference - INTRUST 2014*, volume 9473 of *LNCS*. Springer, 2015. To be published.

[155] E. Wenger and P. Wolfger. Solving the Discrete Logarithm of a 113-Bit Koblitz Curve with an FPGA Cluster. In *Selected Areas in Cryptography*, volume 8781 of *LNCS*, pages 363–379. Springer Berlin Heidelberg, 2014.

[156] D. Wiedemann. Solving sparse linear equations over finite fields. *IEEE Transactions on Information Theory*, 32(1):54–62, 1986.