Centro de Investigación y de Estudios Avanzados del Instituto Politécnico Nacional

Unidad Zacatenco

Departamento de Computación

Análisis e implementación eficiente de protocolos criptográficos de llave pública

Tesis que presenta

José Eduardo Ochoa Jiménez

para obtener el grado de

Doctor en Ciencias en Computación

Director de tesis

Dr. Francisco José Rambó Rodríguez Henríquez

Ciudad de México **Febrero, 2019**

Center for Research and Advanced Studies of
National Polytechnic Institute

Zacatenco campus

Computer science department

Analysis and efficient implementation of public key cryptographic protocols

Submitted by

José Eduardo Ochoa Jiménez

for the degree of

Ph.D. in Computer Science

Advisor

Francisco José Rambó Rodríguez Henríquez, Ph.D.

Mexico City **February, 2019**

Dedication

To my family.

Acknowledgements

I thank my advisor Francisco Rodríguez-Henríquez for his guidance and support during my Ph.D. studies.

I also thank my friends from the cryptography laboratory for sharing their knowledge and for having shared with me hard and good times.

I thank the Consejo Nacional de Ciencia y Tecnología CONACyT for the scholarship provided to me during the period that I was a Ph.D. candidate at Cinvestav.

And finally, a special thanks to the Department staff who supported me during my Ph.D. research.

Resumen

Durante las últimas cuatro décadas, el paradigma de la criptografía de llave pública ha brindado soluciones elegantes a diversos problemas difíciles que surgen de aplicaciones contemporáneas de seguridad de la información. Ejemplos de estos problemas incluyen: autenticación de entidades, anonimato, no repudio, por nombrar sólo algunos. No obstante, la implementación eficiente de la criptografía de llave pública implica el cálculo de operaciones aritméticas no triviales sobre operandos extremadamente grandes. Por tal motivo, el objetivo principal de esta tesis es analizar cuidadosamente algunos de los protocolos criptográficos de llave pública más populares, con la finalidad de identificar las operaciones críticas que influyen significativamente en el costo computacional de dichos esquemas. Una vez identificadas estas operaciones, nuestro siguiente objetivo es proponer mejoras algorítmicas y/o de implementación que permitan reducir significativamente el tiempo de ejecución de estos protocolos, mientras se mantiene la seguridad en contra de ataques de canal lateral de los esquemas. Este trabajo de investigación examina tres subáreas diferentes de la criptografía de llave pública, es decir, esquemas basados en la factorización de números enteros, emparejamientos e isogenias. Las primeras dos subáreas se han utilizado e implementado intensamente en innumerables aplicaciones de seguridad de la información. Mientras que el último esquema es un candidato prometedor para realizar el intercambio de llaves secretas en un escenario post-cuántico, donde se supone que ya se encuentran disponibles computadoras cuánticas suficientemente poderosas. Nuestro estudio comienza realizando un análisis cuidadoso de la implementación eficiente de la aritmética entera y de campos finitos en las micro-arquitecturas de los procesadores más recientes. Este análisis nos permitió diseñar una biblioteca de software que es utilizada para implementar de manera segura el algoritmo de firma RSA. De la subárea de criptografía basada en emparejamientos, se aborda el problema general del "hashing" de tiempo constante hacia curvas elípticas, donde se proponen algoritmos prácticos, eficientes y seguros para realizar el "hashing" a los subgrupos de curva elíptica utilizados en este tipo de protocolos. Además, se diseñó una biblioteca de software que implementa dos protocolos de autenticación de dos factores, los cuales son seguros contra ataques simples de canal lateral. Por otra parte, proponemos el uso de emparejamientos sobre curvas elípticas con grado de encajamiento uno para implementar el protocolo de firma corta propuesto por Boneh, Lynn y Shacham (BLS). Nuestro esquema aprovecha el hecho de que la mejora algorítmica para el cálculo de logaritmos discretos, reportado recientemente por Kim y Barbulescu, no se aplica al escenario cuando el "Discrete Logarithm Problem" (DLP) se calcula en campos finitos de orden primo. En el caso de la criptografía basada en isogenias, se proponen diversas optimizaciones algorítmicas cuyo objetivo es mejorar el desempeño de las operaciones aritméticas de curvas elípticas y campos finitos. Estas optimizaciones producen una aceleración importante del tiempo de ejecución del protocolo "Supersingular Isogeny Diffie-Hellman" (SIDH). Finalmente, se presenta una nueva construcción del protocolo SIDH, utilizando isogenias cuyo grado no es una potencia de un primo, la cual permite conseguir una aceleración considerable en su cálculo.

Abstract

During the last four decades, the public-key cryptography paradigm has provided elegant solutions to several difficult problems that arise in contemporary information security applications. Examples of these problems include, entity authentication, anonymity, non-repudiation to name just a few. Nevertheless, the efficient implementation of public-key cryptography involves the computation of non-trivial arithmetic operations with extremely large operands. Accordingly, the main research goal of this thesis is to carefully analyze some of the most popular public key cryptographic protocols with the aim of identifying critical operations that significantly influence the whole computational cost of those schemes. Once that these operations were identified, our next objective was to propose algorithmic and/or implementation improvements that allow us to obtain significant speedups in the running time of those protocols, while keeping a sound security of those schemes against side-channel attacks. In this research work, we examine three different sub-areas of public-key cryptography, namely, Integer-factorization-based, pairing-based and isogeny-based cryptographic schemes. The first two sub-areas have been intensively used and deployed in innumerable information security applications. The last sub-area is a promising candidate for computing secret key-exchange in a post-quantum scenario, where it is assumed that powerful quantum computers are already available. Taking into account practical considerations, we started our study by performing a careful analysis of the efficient implementation of integer and finite field arithmetic over the newest desktop micro-architectures. In this study, different techniques for the efficient computation of modular multiplication were especially analyzed due to the large influence of this operation in the performance achieved by the cryptographic schemes studied in this work. This study allows us to design a software library used for implementing the RSA signature algorithm in a secure way. In the case of pairing-based cryptography, we discuss the general problem of constant-time hashing into elliptic curves and we propose practical, efficient, and secure algorithms for hashing values to elliptic curve subgroups used in pairing-based cryptographic protocols. Moreover, we design a software library that implements two pairing-based two-factor authentication protocols, which allows to thwart simple side-channel attacks. Then, we also propose the usage of pairings over elliptic curves with embedding degree one to implement the Boneh, Lynn and Shacham (BLS) short signature protocol. Our scheme takes advantage of the fact that the algorithmic improvement for computing discrete logarithms recently reported by Kim and Barbulescu, do not apply to the scenario when the Discrete Logarithm Problem (DLP) is computed on prime-order fields $\mathbb{F}_p$. In the case of isogeny-based cryptography, we proposed several algorithmic optimizations targeting both elliptic curve and finite field arithmetic operations. These optimizations yielded an important speedup in the runtime cost of the Supersingular Isogeny Diffie-Helmann (SIDH) protocol. Finally, we presented a new construction of the SIDH using non-prime power degree isogenies in the Bob's side, which allows us to achieve a considerable speedup in its computation.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

Nowadays we can find *cryptography* everywhere. An example of this are the security mechanisms that rely entirely on it, and which are a fundamental part of any computer system that seeks to provide a certain security level. For this reason, it is considered that the field of *modern cryptography* involves much more than just establishing a secure communication between two entities, which was the original purpose of *classical cryptography*. Nowadays, the cryptographic community also tries to solve a variety of problems including, message authentication, digital signatures, secret key exchange, entities authentication, electronic voting, or digital money. Bearing this in mind, we can define cryptography as "The scientific study of the techniques used to keep digital information, electronic transactions, and distributed calculations secure" [106].

The importance of cryptography lies in the fact that through it we can provide crucial security services such as: *authentication*, which allows to certify that an entity is who it claims to be, using mechanisms such as digital signatures or biometric features; *confidentiality*, which ensures that private information can be consulted or manipulated only by authorized entities; *integrity* that gives the certainty that a document has not been modified by unauthorized entities; *non-repudiation*, which provides protection to an entity in the case that another denies, subsequently, that a certain transaction was made; and *access control* that provides the ability of allowing or denying the usage of a certain resource to a particular entity.

Such security services are generally implemented by means of crypto-schemes. A familiar way of explaining a crypto-scheme is presented in the classical cryptographic scenario. In that scenario two entities, which in the field of cryptography have been traditionally named as Alice and Bob, wish to exchange messages in a secure way such that a third entity, who is known as Eve, is not able to understand its content. To do this, they use the following communication cryptographic scheme: Alice using an encryption algorithm and a secret key that was previously agreed with Bob, transforms the original message into an encrypted message incomprehensible for Eve, which is sent to Bob. On the other side, when Bob receives the encrypted message, he transforms it to the original message by using a decryption algorithm and the secret key agreed with Alice. This approach is called *symmetric cryptography* or *secret key cryptography*, because it is necessary that Alice and Bob agree in advance a shared secret, which is used to encrypt and decrypt messages.

Although the computation of this kind of cryptography is highly efficient, it has the disadvantage that if n entities wish to securely communicate then the system administrator has to provide $O(n^2)$ keys. This is because, the same secret key is used for both encrypting and decrypting messages, and therefore, each key must be different for each pair of entities. A non trivial problem left open in this paradigm is how to securely share the secret key among the participant entities.

The aforementioned secret key cryptography shortcomings may be solved using the so-called *asymmetric cryptography* paradigm, also known as *public key cryptography*. Unlike secret key cryptography, in this paradigm each entity is assigned with a pair of keys: a private key that is used to decrypt messages, which is only known by the owner entity; and the public key used to encrypt messages, which is diffused to all the others. This implies that a secure communication of a n-entity community can be achieved by using just $O(2n)$ keys. It is important to mention that these two keys keep a close relationship, since usually the public key is directly constructed from the private key. Due to the above, and considering the fact that all the entities know the public key of the other participants, it must be computationally intractable deduce the private key from the sole information of the public key.

The security of public key schemes is based on the computational intractability of certain hard mathematical problems. Chronologically speaking, public key cryptography was proposed in 1976 by Whitfield Diffie and Martin Hellman [57], who devised a novel scheme that allows two entities to agree a secret key using an insecure communication channel, without previously agreeing a secret. The security of this scheme lies on the difficulty of the Discrete Logarithm Problem (DLP) defined on finite fields. One year later, in 1977, Ron Rivest, Adi Shamir, and Leonard Adleman proposed the cryptographic scheme known as RSA [143], which can be used as both, encryption scheme and digital signature scheme. The RSA security is based on the difficulty of factoring large integers.

Before 1985, the public key schemes were based on elementary number theory, particularly they used the multiplicative group of integers modulo a large integer (in RSA) or a prime number (in the Diffie-Hellman scheme). In 1985 Neal Koblitz [112] and Victor Miller [127] independently proposed the usage of elliptic curves for cryptographic proposes. This gave birth to the field of *Elliptic Curve Cryptography* (ECC). Koblitz and Miller observed that when an elliptic curve is defined over a finite field, the points on the elliptic curve form an Abelian group, where the DLP results difficult to solve and which is even much more difficult than its analogue in finite fields using the same group and field order. Bearing this in mind, it is possible to offer the same security provided by the other existing public key schemes, but using smaller fields. This fact means that it is feasible to use smaller key lengths and also smaller bandwidth and memory for deploying crypto-systems on constrained devices.

Almost a decade later, Alfred J. Menezes, Tatsuaki Okamoto and Scott A. Vanstone [124], proposed in 1993 the usage of the Weil pairing over elliptic curves as an attack method that allows to solve the DLP in the group of points of a family of elliptic curves. This attack reduces the problem to compute the DLP over a group of points belonging to an elliptic curve to its analogue in finite fields. Around the year 2000, constructive cryptographic properties of pairings were proposed in the seminal works of Joux [99]; Sakai, Ohgishi and Kasahara [144]; and Boneh and Franklin [22]. This gave birth to the field of *pairing-based cryptography* (PBC) that bases its security on the difficulty of solving discrete logarithm problems in both, finite fields and elliptic curves. Bilinear pairings have been considered to be one of the most suitable mathematical tools to design secure and efficient crypto-schemes, in virtue of their powerful bilinearity property. This property is useful to solve in an elegant way the problem of the practical implementation of the so called *identity-based cryptography* (IBE) [22], which was theoretically proposed by Adi Shamir in 1984 [153]. The IBE paradigm consist of using a string associated to an entity, like a personal email address or phone number, as her public key.

In the event of the deploying of sufficiently powerful quantum computers, all the cryptographic schemes based on the DLP or the factorization of large integers would be deemed completely insecure. This hypothetical situation has prompted during the last decade the search for finding hard mathematical problems that would be presumably difficult to solve by a quantum attacker creating a sub-discipline coined as quantum-safe cryptography. In August 2015, the U.S. National Security Agency (NSA) released a major policy statement [11], where it was stated that transition towards quantum-safe cryptography should be performed

in the coming years. Reacting accordingly, the cryptographic community has proposed several candidate schemes. One of such proposals is based on the hardness of finding an isogeny map between two elliptic curves, a problem that provides the security guarantees of the so-called isogeny-based cryptography. Reportedly Couveignes made the first suggestions towards the usage of isogenies for cryptographic purposes in a seminar held in 1997, which he later reported in [53]. Jao and Venkatesan showed techniques to provide a public-key encryption system based on isogenies of Abelian varieties [93]. The first published work of a concrete isogeny-based cryptographic primitive was presented by Charles, Lauter and Goren in [34] where they introduced the hardness of path-finding in supersingular isogeny graphs and its application to the design of hash functions. It has since been used as an assumption for other cryptographic applications such as key-exchange and digital signature protocols. Stolbunov studied in [156] the hardness of finding isogenies between two ordinary elliptic curves defined over a finite field $\mathbb{F}_q$, with q a prime power. The author proposed to use this setting as the underlying hard problem for a Diffie-Hellman-like key exchange protocol. Nevertheless, Childs, Jao and Soukharev discovered in [39] a subexponential complexity quantum attack against Stolbunov's scheme. Finally, in 2011 Jao and De Feo proposed the Supersingular Isogeny-based Diffie-Hellman (SIDH) key exchange protocol [95], which is a promising candidate for quantum-safe cryptography.

1.1. Motivation

Although public key cryptography solves some problems found in secret key cryptography, it has the disadvantage that the operations involved in this kind of cryptography turns out to be very expensive in comparison with the operations required in secret key schemes. Because of this reason, it is important to analyze these schemes to determine which operations have a greater influence in their total computational cost. In addition to this need for acceleration, there exists also a great concern about the safety of implementations of cryptographic operations. This is because there are attacks whose objective is not to break the mathematical properties of a particular scheme, but rather to extract the secret key by analyzing the leaked information obtained from the device where such operation is executed. These techniques are generally known as side channel attacks. One avenue of attack using side-channel techniques is to take advantage of the fact that the computations that compose cryptographic operations vary from execution to execution. Hence, it is also considered of crucial importance to perform implementations of cryptographic schemes in constant-time. This countermeasure serves as a first line of defense against this type of attacks.

The main research purpose of this thesis is to closely examine cryptographic protocols ranging from those based on elementary number theory to those based on computation of isogenies of elliptic curves. This analysis focused on identifying the critical-performance operations that have a major influence in the whole computational cost of those protocols. Once that these operations have been identified, our objective is to propose algorithmic or implementation improvements that allow us to obtain significant speedups in their running time and, in addition, that allow us to generate secure implementations against side-channel attacks.

1.2. Outline

The remainder of this document is organized as follows. In Chapter 2, we provide a mathematical background where we present some helpful definitions and fundamental results on the underlying theory and concepts of this thesis.

This thesis is divided into three parts. In the firs part, denoted Integer-factorization based cryptography, we focused on the fast and secure software implementation of integer and finite

field arithmetic and the efficient and protected implementation of the RSA signature scheme.

Chapter 3 is dedicated to show the practical considerations of an efficient implementation of integer and finite field arithmetic over the newest desktop processors. Then, in Chapter 4 we shown the design of a software library used to implement the RSA signature scheme, using an integer and Residue Number System based arithmetic over desktop processors and GPU. The work in this chapter was realized along with Nareli C. Cortéz, Luis A. Rivera-Zamarripa and Francisco Rodríguez-Henríquez. A part of this work was published in [48].

The second part of this thesis entitled pairing-based cryptography, presents the analysis of hash functions into elliptic curves and a efficient and secure implementation of two authentication protocols and the BLS signature scheme.

In Chapter 5 we present a a brief mathematical background about bilinear pairings over elliptic curves. Then, in Chapter 6 we discussed the general problem of constant-time hashing into elliptic curves. We propose practical, efficient, and secure algorithms for hashing values to elliptic curve subgroups used in pairing-based cryptography protocols. This advances are a joint work with Mehdi Tibouchi and Francisco Rodríguez-Henríquez and were published in [135]. Through the Chapter 7, we proposed the design of a software library that implements two paring-based two-factor authentication protocols [148, 149] in a secure way, which thwart simple side-channel attacks. This work was published in [98] and was performed with Francisco Rodríguez-Henríquez. Finally, given that the security of the pairings on BN curves have been affected by the work realized by Kim and Barbulescu in [109], in Chapter 8 we propose the usage of pairings over elliptic curves with embedding degree one. This is because, the aforementioned improvements in algorithms for computing discrete logarithms do not apply to the DLP in prime-order fields $\mathbb{F}_p$ provided that the prime p does not have a special form. With this in mind, we present the design of a software library that implements the digital signature algorithm BLS, constructed over elliptic curves with embedding degree one. This work was performed with Francisco Rodríguez-Henríquez.

The third part of this thesis named isogeny-based cryptography is focused on the development of techniques to accelerate the SIDH protocol running time performance.

In Chapter 9 we provide some mathematical background that is used through this part of the thesis. Then, in Chapter 10 several algorithmic optimizations targeting both elliptic curve and finite field arithmetic operations are proposed, in order to accelerate the SIDH runtime performance. We accelerated the finite field operations, adapted the right-to-left Montgomery ladder variant presented in [138] to the context of the SIDH protocol and presented an improved formula for elliptic curve point tripling. This work was realized along with Armando Faz-Hernández, Julio López and Francisco Rodríguez-Henríquez and was published in [62]. Besides, in Chapter 11 we propose a new construction of the SIDH protocol using non-prime power degree isogenies in the Bob's side, that allow us to achieve a considerable speedup in the computation of the SIDH protocol. This work was performed with Daniel Cervantes-Vazquez and Francisco Rodríguez-Henríquez.

Finally, in Chapter 12, we conclude the thesis by listing more specifically our main contributions an we list the possible future works.

Chapter 2

Mathematical background

Most of cryptographic schemes are built on the foundations of algebra and number theory. For this reason, it results important to know the definitions and properties of some mathematical structures, which allow us to construct such cryptographic schemes. In this chapter, we present some definitions and properties of the mathematical structures used through this thesis such as groups, rings, fields and elliptic curves. For more details about the following content, we refer the reader to consult [86, 155, 66].

2.1. Groups

The main subject of this section is introduces the notion of group, as well as, showing some of its most important properties and features. We begin by defining a binary operation.

Definition 2.1 (Binary operation)**.** *A binary operation $\star$ on a set G is a function mapping $G \times G$ into G. We will denote the operation $\star(a, b)$ for $a, b \in G$ by $a \star b$. Moreover, we say that the binary operation $\star$*

- *is commutative if $a \star b = b \star a$ for all $a, b \in G$, and*
- *is associative if $(a \star b) \star c = a \star (b \star c)$ for all $a, b, c \in G$.*

Example 2.1. *The usual addition $+$ is a binary operation on the set $\mathbb{R}$, and the usual multiplication $\cdot$ is a different binary operation on $\mathbb{R}$.*

Example 2.2. *Considering the set $\mathbb{R}$ of real numbers. The usual subtraction $-$ is a binary operation non-commutative and non-associative because $7-(-7) \neq (-7)-7$ and $7-((-7)-7) \neq (7-(-7))-7$, respectively.*

In Example 2.1 the binary operations are defined for every pair (a, b) of elements in $\mathbb{R}$, however, sometimes a binary operation over a set G also provides a binary operation on a subset H of G.

Definition 2.2 (Induced operation)**.** *Let $\star$ be a binary operation on G and let H be a subset of G. The subset H is closed under $\star$ if for all $a, b \in H$ we also have $a \star b \in H$. In this case, the binary operation on H given by restricting $\star$ to H is the induced operation of $\star$ on H.*

Example 2.3. *The addition $+$ on the set $\mathbb{R}$ of real numbers does not induces a binary operation on the set $\mathbb{R}^*$ of non-zero real numbers because $7 \in \mathbb{R}^*$ and $-7 \in \mathbb{R}^*$, but $7+(-7) = 0$ and $0 \notin \mathbb{R}^*$. Thus, $\mathbb{R}^*$ is not closed under $\star$.*

Definition 2.3 (Group)**.** *A group $(G, \star)$, denoted as $\mathbb{G}$, is a set G closed under a binary operation $\star$, such that the following properties are satisfied:*

- *associativity: for all $a, b, c \in G$, we have $(a \star b) \star c = a \star (b \star c)$;*
- *identity: there exists an unique element $e \in G$ such that for all $a \in G$, $a \star e = e \star a = a$;*
- *inverse: for all $a \in G$ there exists an unique element $a' \in G$ such that $a \star a' = a' \star a = e$.*

Definition 2.4 (Abelian group)**.** *A group $\mathbb{G}$ is Abelian if its binary operation is commutative.*

When it is required to specify a group $\mathbb{G}$, one must determine both the underlying set G as well as the binary operation $\star$. However, generally the binary operation is implicit from the context, and by abuse of notation we often refers to $\mathbb{G} = G$ as the group. For example, when we discuss about the Abelian group $\mathbb{Z}_n$ for a positive integer n, it is understood that the binary operation is the addition, while when we talking about the Abelian group $\mathbb{Z}_n^*$ it is understood that the binary operation is the multiplication.

In addition, instead of using the symbol $\star$ for the binary operation, generally we use the regular addition " $+$ " and multiplication " $\cdot$ " symbols. If *an Abelian group is additively written*, then the identity element is denoted by $0_{\mathbb{G}}$ or just 0 if $\mathbb{G}$ is clear from the context; the inverse of a element $a \in \mathbb{G}$ is denoted by $-a$; and for $a, b \in \mathbb{G}$, $a - b$ is calculated as $a + (-b)$. Conversely, if *an Abelian group is multiplicatively written*, then the identity element is denoted by $1_{\mathbb{G}}$ or just 1 if $\mathbb{G}$ is clear from context; and the inverse of a element $a \in \mathbb{G}$ is denoted by a^{-1}.

Example 2.4. *The set of integers $\mathbb{Z}$ closed under the addition forms an Abelian group, with 0 being the identity, and $-a$ being the inverse of $a \in \mathbb{Z}$.*

Example 2.5. *The set $\mathbb{Z}_n^*$ of residue classes modulo a positive integer n, with $\gcd(a, n) = 1$ for all $a \in \mathbb{Z}_n^*$, forms an Abelian group. Where 1 is the identity, and a^{-1} is the multiplicative inverse of a modulo n.*

Definition 2.5 (Group order)**.** *A group $\mathbb{G}$ may be finite or infinite. If the group is finite its* order *is defined as the cardinality of the underlying set G, i.e. $|G| = n \in \mathbb{N}$; otherwise, we say that the group has infinite order.*

Definition 2.6 (Order of a group element)**.** *Let $\mathbb{G}$ be a group. The order of an element $a \in \mathbb{G}$, is defined as the smallest positive integer n such that $\star^n(a) = e_{\mathbb{G}}$ [1]. If such integer exists, we say that a has a finite order (or that a is a torsion n element); otherwise, the element a has infinite order.*

A common way to represent a finite group is using a Cayley's table, which describes the group structure by arranging all possible results of the binary operation over all elements in the group.

Example 2.6. *Given the group $\mathbb{Z}_7^*$, its corresponding Cayley's table is presented in Table 2.1. From the Cayley's table we can see that $\mathbb{Z}_7^* = \{1, 2, 3, 4, 5, 6\}$ and it has order 6. Moreover, we can find the order of any element following the Cayley's table: for the element $4 \in \mathbb{Z}_7^*$, for example, we compute $4^2 = 4 \cdot 4 = 2$ according to the Table 2.1, then we obtain $4^3 = 4^2 \cdot 4 = 2 \cdot 4 = 1$. At this point, we know that the element $4 \in \mathbb{Z}_7^*$ has order 3.*

[1] $\star^n(a)$ denotes the *n-th* application of the operation $\star$ to a group element a.

$\cdot$	1	2	3	4	5	6
1	1	2	3	4	5	6
2	2	4	6	1	3	5
3	3	6	2	5	1	4
4	4	1	5	2	6	3
5	5	3	1	6	4	2
6	6	5	4	3	2	1

Table 2.1: Cayley's table for $\mathbb{Z}_7^*$.

2.1.1. Subgroups

From this section onward, we will deal only with Abelian groups and, for simplicity, we will say group instead of Abelian group.

Definition 2.7 (Subgroup)**.** *If a subset H of a group $\mathbb{G}$ is closed under the binary operation of $\mathbb{G}$ and if H with the induced operation from $\mathbb{G}$ is itself a group, then H is a subgroup of $\mathbb{G}$. We shall let $H \subseteq \mathbb{G}$ denote that H is a subgroup of $\mathbb{G}$, and $H \subset \mathbb{G}$ shall denote that $H \subseteq \mathbb{G}$ but $H \neq \mathbb{G}$.*

Definition 2.8. *Let $\mathbb{G}$ be a group. The subgroup $H = \mathbb{G}$ is called improper subgroup of $\mathbb{G}$, and all other subgroups are said to be proper subgroups of $\mathbb{G}$. Furthermore, the subgroup $\{e\}$ containing only the identity element is called trivial subgroup, and all other subgroups are called non-trivial subgroups of $\mathbb{G}$.*

Theorem 2.1 ([66, Theorem 5.14])**.** *A subset H of a group $\mathbb{G}$ is a subgroup of $\mathbb{G}$ if and only if*

- *H is closed under the binary operation $\star_{\mathbb{G}}$,*
- *the identity element $e_{\mathbb{G}}$ is in H, and*
- *for all $a \in H$ it is true that $a^{-1} \in H$.*

Example 2.7. *For the group $\mathbb{Z}_9 = \{0, 1, 2, 3, 4, 5, 6, 7, 8\}$, the subset $H = \{0, 3, 6\}$ is a subgroup of $\mathbb{Z}_9$. The identity element 0 is in H, the additive inverse modulo 9 of elements in H are also in H, and the set H is closed under the modular addition.*

2.1.2. Cyclic groups

Let $\mathbb{G}$ be a group. From now on we will denote as a^n the application of the operation " $\cdot$ " $n-1$ times on an element $a \in \mathbb{G}$ (or na if the operation is " $+$ ").

Theorem 2.2 ([66, Theorem 5.17])**.** *Let $\mathbb{G}$ be a group and let $a \in \mathbb{G}$. Then*

$$H = \{a^n \mid n \in \mathbb{Z}\}$$

is a subgroup of $\mathbb{G}$ and is the smallest subgroup of $\mathbb{G}$ that contains a, that is, every subgroup containing a contains H.

Definition 2.9 (Cyclic subgroup)**.** *Let $\mathbb{G}$ be a group and let $a \in \mathbb{G}$. Then the subgroup $\{a^n \mid n \in \mathbb{Z}\}$ of $\mathbb{G}$ is called the cyclic subgroup of $\mathbb{G}$ generated by a, and denoted by $\langle a \rangle$.*

Definition 2.10 (Cyclic group)**.** *An element a of a group $\mathbb{G}$ generates $\mathbb{G}$ and is a generator for $\mathbb{G}$ if $\langle a \rangle = \mathbb{G}$. A group $\mathbb{G}$ is cyclic if there is some element $a \in \mathbb{G}$ that generates $\mathbb{G}$.*

Example 2.8. *The set $\mathbb{Z}_6$ closed under the operation of addition modulo 6 forms an Abelian group. The Cayley's table for this group is presented in Table 2.2. We can observe that the group $\mathbb{Z}_6$ is cyclic because $\mathbb{Z}_6 = \langle 1 \rangle = \langle 5 \rangle$, and moreover, it has the following cyclic subgroups:*

$$\begin{aligned} \langle 1 \rangle = \langle 5 \rangle &= \{0, 1, 2, 3, 4, 5\}, \\ \langle 2 \rangle = \langle 4 \rangle &= \{0, 2, 4\} \\ \langle 3 \rangle &= \{0, 3\} \end{aligned}$$

$+$	0	1	2	3	4	5
0	0	1	2	3	4	5
1	1	2	3	4	5	0
2	2	3	4	5	0	1
3	3	4	5	0	1	2
4	4	5	0	1	2	3
5	5	0	1	2	3	4

Table 2.2: Cayley's table for $\mathbb{Z}_6$.

Theorem 2.3 ([66, Theorem 6.1])**.** *Every cyclic group is Abelian.*

Theorem 2.4 ([66, Theorem 6.6])**.** *A subgroup of a cyclic group is cyclic.*

Theorem 2.5 ([66, Theorem 6.10])**.** *Let $\mathbb{G}$ be a cyclic group with generator a. If the order of $\mathbb{G}$ is infinite, then $\mathbb{G}$ is isomorphic to $(\mathbb{Z}, +)$. If $\mathbb{G}$ has finite order n, then $\mathbb{G}$ is isomorphic to $(\mathbb{Z}_n, +_n)$.*

2.1.3. Cosets

In this section, a generalization of the notion of congruence relation is presented. Given a group $\mathbb{G}$, and a subgroup H of $\mathbb{G}$. For all $a, b \in \mathbb{G}$, we write $a \equiv b \mod H$ if $a \star b' \in H$, where the element b' as before represents the inverse of b. This binary relation is an equivalence relation that partitions $\mathbb{G}$ into equivalences classes.

For an element $a \in \mathbb{G}$ we denote as $[a]_H$ the equivalence class containing a. This equivalence class is defined as:

$$[a]_H = a \star H = \{a \star h \mid h \in H\},$$

i.e. $x \in [a]_H \iff x \equiv a \mod H$. Those equivalence classes are called *cosets of H in $\mathbb{G}$*, and any member of a coset is called a *representative of the coset.*

Example 2.9. *Let $\mathbb{G} = \mathbb{Z}_{12}$ be an additive Abelian group and given the subgroup $H = 4\mathbb{Z}_{12} = \{0, 4, 8\}$ of $\mathbb{G}$, then the cosets of H in $\mathbb{G}$ are*

$$\begin{aligned} [0]_H &= \{0 + h \mid h \in H\} = \{0, 4, 8\}, \\ [1]_H &= \{1 + h \mid h \in H\} = \{1, 5, 9\}, \\ [2]_H &= \{2 + h \mid h \in H\} = \{2, 6, 10\}, \\ [3]_H &= \{3 + h \mid h \in H\} = \{3, 7, 11\}. \end{aligned}$$

Definition 2.11 (Quotient group)**.** *The set of all cosets of H in $\mathbb{G}$, denoted as $\mathbb{G}/H$, is called the* quotient group of $\mathbb{G}$ modulo H. *The binary operation of $\mathbb{G}/H$ is defined as $[a]_H \star [b]_H = [a \star b]_H$.*

Example 2.10. *Returning to the Example 2.9, the quotient group* $\mathbb{Z}_{12}/4\mathbb{Z}_{12}$ *has order* $|\mathbb{Z}_{12}|/|4\mathbb{Z}_{12}| = 12/3 = 4$ *and its Cayley table is presented in Table 2.3.*

$+$	$[0]_H$	$[1]_H$	$[2]_H$	$[3]_H$
$[0]_H$	$[0]_H$	$[1]_H$	$[2]_H$	$[3]_H$
$[1]_H$	$[1]_H$	$[2]_H$	$[3]_H$	$[0]_H$
$[2]_H$	$[2]_H$	$[3]_H$	$[0]_H$	$[1]_H$
$[3]_H$	$[3]_H$	$[0]_H$	$[1]_H$	$[2]_H$

Table 2.3: Cayley's table for $\mathbb{Z}_{12}/4\mathbb{Z}_{12}$.

In the table we can see that $\mathbb{Z}_{12}/4\mathbb{Z}_{12}$ *is a group with the same structure that the group* $\mathbb{Z}_4$.

From example 2.9 we can notice that every coset of a subgroup H in a group $\mathbb{G}$ has the same order than H. This fact is described in the following theorem.

Theorem 2.6 (Lagrange's theorem [66, Theorem 10.10])**.** *Let* $\mathbb{G}$ *be a finite Abelian group, and* H *a subgroup of* $\mathbb{G}$*. The order of* H *divides the order of* $\mathbb{G}$*.*

Corollary 2.1. *Given a finite group* $\mathbb{G}$ *and an element* $a \in \mathbb{G}$*, the order of* a *is a divisor of the order of* $\mathbb{G}$*.*

2.1.4. Group homomorphisms

In this section, we present maps that relate group structures. Such maps fulfill some properties and provide some applications, which allow to understand the structure of a certain group using features of another group.

Definition 2.12 (Group homomorphism)**.** *A group homomorphism is a map* ϕ *from an Abelian group* $\mathbb{G}$ *to an Abelian group* $\mathbb{G}'$ *such that*

$$\phi(a \star_{\mathbb{G}} b) = \phi(a) \star_{\mathbb{G}'} \phi(b)$$

for all $a, b \in \mathbb{G}$*.*

When we talk about a map $\phi : \mathbb{G} \to \mathbb{G}'$ we are interested in the set $\phi(\mathbb{G}) = \{\phi(a) \mid a \in \mathbb{G}\}$ that is called the *image of* ϕ; and the set of all elements of $\mathbb{G}$ that are mapped to the identity element $e_{\mathbb{G}'}$ known as the *kernel of* ϕ. Those two sets, generally, are denoted as $Img(\phi)$ and $Ker(\phi)$, respectively.

Example 2.11. *Let* $\mathbb{Z}$ *a group under the regular addition and* $\mathbb{Z}_n$ *a group under the addition modulo an integer* n*. Define* $\phi : \mathbb{Z} \to \mathbb{Z}_n$ *as* $\phi(a) = [a]_n$*, where* $[\cdot]_n$ *denotes the operation* $a \bmod n$*. Then,* ϕ *is an homomorphism because*

$$\phi(a) +_{\mathbb{Z}_n} \phi(b) = [a]_n +_{\mathbb{Z}_n} [b]_n = [a+b]_n = \phi(a +_{\mathbb{Z}} b).$$

The kernel of this homomorphism is $n\mathbb{Z} = \{na \mid a \in \mathbb{Z}\}$ *and its image is the set* $\{\phi(a) \mid a \in \mathbb{Z}\}$*.*

Example 2.12. *Let* $\mathbb{G}$ *a additively written group and* m *an integer. The map* $\phi : \mathbb{G} \to \mathbb{G}$ *defined as* $\phi(a) = ma$ *is an homomorphism, since*

$$\phi(a+b) = m(a+b) = ma + mb = \phi(a) + \phi(b).$$

The image of ϕ *is the subgroup* $m\mathbb{G}$ *and its kernel is the subgroup* $\mathbb{G}\{m\}$*. This map is called the* m*-multiplication map on* $\mathbb{G}$ *(or the* m*-power map on* $\mathbb{G}$ *if the group is written multiplicatively).*

Theorem 2.7 ([66, Theorem 13.12]). *Let ϕ be a group homomorphism form the group $\mathbb{G}$ to the group $\mathbb{G}'$, then*

- *If $e_{\mathbb{G}}$ is the identity element of $\mathbb{G}$, then $\phi(e_{\mathbb{G}}) = e_{\mathbb{G}'}$,*
- *if $a \in \mathbb{G}$, then $\phi(-a) = -\phi(a)$ if the groups were additively written and $\phi(a^{-1}) = \phi(a)^{-1}$ if were multiplicatively written,*
- *if $a \in \mathbb{G}$, then $\phi(na) = n\phi(a)$ if the groups were additively written and $\phi(a^n) = \phi(a)^n$ if were multiplicatively written,*
- *if H is a subgroup of $\mathbb{G}$, then $\phi(H)$ is a subgroup of $\mathbb{G}'$.*

The group homomorphisms may be classified according to the way that its domain and image are mapped to each other, this classification is summarized in the Definition 2.13.

Definition 2.13. *Let $\phi : \mathbb{G} \to \mathbb{G}'$ be a group homomorphism. We say that ϕ is a group:*

- *monomorphism if ϕ is injective;*
- *epimorphism if ϕ is surjective;*
- *isomorphism if ϕ is bijective, and also we say that $\mathbb{G}$ is isomorphic to $\mathbb{G}'$;*
- *automorphism if ϕ is a isomorphism and $\mathbb{G} = \mathbb{G}'$.*

Theorem 2.8. *If ϕ is a group isomorphism of $\mathbb{G}$ with $\mathbb{G}'$, then the inverse map ϕ^{-1} is a group isomorphism of $\mathbb{G}'$ with $\mathbb{G}$.*

Example 2.13. *In the Example 2.10 we can see that the group $\mathbb{Z}_{12}/4\mathbb{Z}_{12}$ is isomorphic to $\mathbb{Z}_4$, which is denoted as $\mathbb{Z}_{12}/4\mathbb{Z}_{12} \cong \mathbb{Z}_4$.*

2.1.5. Discrete Logarithm Problem (DLP)

The discrete logarithm problem is one of the most used mathematical problems in asymmetric cryptography. This problem must be hard in well-chosen groups, so that secure-enough cryptosystems can be build.

Definition 2.14 (Discrete Logarithm Problem (DLP)). *Given an Abelian group $\mathbb{G} = (G, \cdot)$, an element $a \in \mathbb{G}$ and a generator g of $\mathbb{G}$. The discrete logarithm problem consist in finding a solution for the equation $g^x = a \in \mathbb{G}$ knowing the value of g and a.*

2.2. Rings

In this section we introduce the the notion of a ring, which is basically an algebraic structure with addition and multiplication operations. In particular we are interested in the definition of a commutative ring with unity.

Definition 2.15 (Commutative ring with unity). *A commutative ring with unity is conformed by a set R, and the binary operations of addition " $+$ " and multiplication " $\cdot$ " on R. Where such operations satisfy the following properties:*

- *the set R under the addition forms an Abelian group, whose identity is 0_R;*
- *operation of multiplication is associative. i.e for all $a, b, c \in R$ it holds that $a(bc) = (ab)c$;*

- *multiplication distributes over the addition. i.e. for all* $a, b, c \in R$ *we have that* $a(b+c) = ab + ac$ *and* $(b + c)a = ba + ca$*;*
- *there exist a multiplicative identity. i.e. there is an element* $1_R \in R$ *such that* $1_R a = a = a1_R$ *for all* $a \in R$*;*
- *multiplication is commutative. i.e. for all* $a, b \in R$ *we have that* $ab = ba$*.*

If the last two properties are not satisfied, then we have the definition of a *ring*. Since we will not work with general rings, we will say simply ring instead of commutative ring with unit, and a ring will be denoted by R.

Example 2.14. *The sets of integers* $\mathbb{Z}$*, real numbers* $\mathbb{R}$ *and rational numbers* $\mathbb{Q}$ *are rings under the usual rules of multiplication and addition on each set.*

Example 2.15. *The cyclic group* $\mathbb{Z}_n$ *under the addition modulo* n *and the multiplication modulo* n *forms a ring.*

Definition 2.16 (Characteristic of R)**.** *Let* R *be a ring. If there exists a positive integer* $n \neq 0$ *such that* $n1_R = \sum_{i=0}^{n-1} 1_R = 0_R$*, then the least positive integer that satisfy such condition is called the characteristic of* R*. If such integer* n *does not exists for* R*, we say that the characteristic of* R *is zero.*

Example 2.16. *The ring* $\mathbb{Z}$ *has characteristic zero, and the ring* $\mathbb{Z}_n$ *has characteristic* n*.*

Definition 2.17 (Unit)**.** *Let* R *be a ring. A unit is an element* $a \in R$ *such that* $a \mid 1_R$ *(that means that* a *divides* 1_R*), i.e.* $ab = 1_R$ *for an element* $b \in R$*. The element* b *is unique and is called the multiplicative inverse of* a*.*

A ring R is an Abelian group with respect to the binary operation of addition, and a subgroup of such group is called *subgroup of the additive group of* R. Moreover, the set of units, denoted as R^*, is closed under the multiplication, and is an Abelian group with respect to the multiplication operation called *the multiplicative group of units of* R*.*

Example 2.17. *For the ring* $\mathbb{Z}_7$ *its multiplicative group of units is defined as* $\mathbb{Z}_7^* = \{1, 2, 3, 4, 5, 6\}$*, and the subgroup of the additive group is* $\mathbb{Z}_7 = \{0, 1, 2, 3, 4, 5, 6\}$*.*

Example 2.18. *The ring* $\mathbb{Z}$ *only has two units* ± 1

2.2.1. Subrings, ideals and quotient rings

In this section, the definitions and notions of the concepts of subrings, ideals, principal ideals and quotient rings are presented.

Definition 2.18 (Subring)**.** *Let* R *be a ring. A subset* S *of* R *is called a subring if satisfies the following properties*

- $1_R \in S$*,*
- S *is closed under multiplication, and*
- S *is an additive subgroup of* R*.*

From the above definition, we can observe that the subring S is also a ring where 0_R is the additive identity of S and 1_R is the multiplicative identity of S. We may also call R an *extension ring* of S.

Example 2.19. $\mathbb{Q}$ *is a subring of* $\mathbb{R}$*.*

Definition 2.19 (Ideal)**.** *Let R be a ring. An ideal I of R is an additive subgroup of R such that $ar \in I$ for all $a \in I$ and $r \in R$.*

Theorem 2.9. *Let R be a ring and let $a \in R$, then $aR = \{ar \mid r \in R\}$ is an ideal of R and it is called the ideal of R generated by a.*

Example 2.20. *For the ring $\mathbb{Z}_n$ the set $m\mathbb{Z}$, for each $m \in \mathbb{Z}$, is an additive subgroup of the ring $\mathbb{Z}_n$ and an ideal of this ring. An ideal of this form is called a principal ideal.*

Let I be an ideal of a ring R. And given the set of equivalence classes with respect to the congruence relation $a \equiv b \mod I$, where $a - b \in I$. It forms a ring R/I, if we define the multiplication and addition in R/I in terms of multiplication or addition of coset representatives. Such ring is called a *quotient ring* and the elements of R/I are called residue classes.

Example 2.21. *The ring $\mathbb{Z}_n$ is precisely the quotient ring $\mathbb{Z}/n\mathbb{Z}$ for $n \geq 1$.*

2.2.2. Ring homomorphisms

Definition 2.20 (Ring homomorphism)**.** *A ring homomorphism is a map ϕ from a ring R to a ring R' if*

- *ϕ is a group homomorphism with respect to the underlying additive groups of R and R',*
- *for all $a, b \in R$ it holds that $\phi(ab) = \phi(a)\phi(b)$, and*
- *$\phi(1_R) = 1_{R'}$.*

2.3. Fields

In the same way as ring, a field is an algebraic structure conformed by a set and two binary operations. Formally, a field is defined as follows:

Definition 2.21 (Field)**.** *A field $\mathbb{F}$ is a ring where every non-zero element of $\mathbb{F}$ has a multiplicative inverse. The subgroup of the additive group of the field is the set $\mathbb{F}$; and the subgroup $\mathbb{F} - \{0\}$ is the multiplicative group of units of $\mathbb{F}$. Moreover, the characteristic of $\mathbb{F}$ (see 2.16) is either zero or a prime number.*

For a field $\mathbb{F}$, we say that $\mathbb{F}$ is a infinite field if its characteristic is zero; otherwise we say that $\mathbb{F}$ is a finite field.

Theorem 2.10. *The ring $\mathbb{Z}/p\mathbb{Z}$ of residue classes of integers modulo the principal ideal generated by a prime p, is a field.*

Example 2.22. *Given the ring $\mathbb{Z}/5\mathbb{Z} = \{0, 1, 2, 3, 4\}$ under the binary operations of addition and multiplication modulo 5. We can see that the additive Abelian group of $\mathbb{Z}/5\mathbb{Z}$ has order 5 and the multiplicative group of units of $\mathbb{Z}/5\mathbb{Z}$ has order 4. The Table 2.4 shows the Cayley's table for both groups.*

In addition, the multiplication operation is associative, distributive over the addition, and commutative. Moreover, every non-zero element in $\mathbb{Z}/5\mathbb{Z}$ has an multiplicative inverse. Therefore, the ring $\mathbb{Z}/5\mathbb{Z}$ is a finite field with prime characteristic 5, and is denoted as $\mathbb{F}_5$.

+	0	1	2	3	4
0	0	1	2	3	4
1	1	2	3	4	0
2	2	3	4	0	1
3	3	4	0	1	2
4	4	0	1	2	3

·	1	2	3	4
1	1	2	3	4
2	2	4	1	3
3	3	1	4	2
4	4	3	2	1

Table 2.4: Cayley's tables for $\mathbb{F}_5$.

Theorem 2.11. *Every finite field $\mathbb{F}$ must have order p^n, where p is prime, n is a positive integer, and p is the characteristic of $\mathbb{F}$.*

Theorem 2.12. *For every finite field $\mathbb{F}_p$ the multiplicative group $\mathbb{F}_p^*$ of nonzero elements of $\mathbb{F}_p$ is cyclic.*

Definition 2.22 (Primitive element)**.** *A generator of the cyclic group $\mathbb{F}_p^*$ is called a primitive element of $\mathbb{F}_q$.*

The following theorems present very useful facts about number theory. The first is due to Fermat and the second is a generalization of Fermat's little theorem.

Theorem 2.13 (Fermat's little theorem)**.** *For an element $a \in \mathbb{Z}$ and a prime number p such that p does not divides a. It holds that $a^{p-1} \equiv 1 \mod p$.*

Theorem 2.14 (Euler's theorem)**.** *For an element $a \in \mathbb{Z}$ relative prime to $n \in \mathbb{Z}$, i.e. with $\gcd(a, n) = 1$. It holds that $a^{\varphi(n)} \equiv 1 \mod n$, where $\varphi(n)$ is Euler's totient function.*

2.3.1. Field extensions

Let $\mathbb{F}$ be a field. A subset K of F that is itself a field under the operations of $\mathbb{F}$ will be called a *subfield* of $\mathbb{F}$. In this context, $\mathbb{F}$ is called an *extension field* of K. In the same manner that for groups, if $K \neq \mathbb{F}$ we say that K is a proper subfield of $\mathbb{F}$.

Definition 2.23 (Prime field)**.** *A field containing no proper subfields is called a prime field.*

Theorem 2.15. *Let $\mathbb{F}_q$ be the finite field with $q = p^n$ elements. Then every subfield of $\mathbb{F}_q$ has order p^m, where m is a positive divisor of n. Conversely, if m is a positive divisor of n, then there is exactly one subfield of $\mathbb{F}_q$ with p^m elements. Then, the field $\mathbb{F}_q$ is an extension field of every subfield of $\mathbb{F}_q$.*

In order to exemplify the concept of extension field, first it is necessary to recall some concepts as polynomial ring and irreducible polynomial.

Definition 2.24 (Polynomial ring)**.** *The set of polynomials in the variable x with coefficients in a ring R under the operations of addition and multiplication of polynomials, is called the polynomial ring over R and denoted by $R[x]$.*

Theorem 2.16. *Let R be a ring. Then*

- *$R[x]$ is commutative if and only if R is commutative.*
- *$R[x]$ is a ring with identity if and only if R has identity.*
- *$R[x]$ is a finite field if and only if R is a finite field.*

In the rest of this section we will deal exclusively with polynomials rings with coefficients in a field $\mathbb{F}$.

Definition 2.25 (Irreducible polynomial)**.** *A polynomial $f \in \mathbb{F}[x]$ is said to be irreducible in $\mathbb{F}[x]$ if f has positive degree and $f = gh$ with $g, h \in \mathbb{F}[x]$ implies that either g or h is the zero polynomial or a polynomial of degree zero.*

Example 2.23. *Compute the irreducible polynomials over $\mathbb{F}_2$ of degree 4. There are $2^4 = 16$ polynomials in $\mathbb{F}[x]$ of degree 4. The irreducible polynomials of degree 4 are those without a divisor of degree 1 or 2. Therefore, if we compute all products $(a_0 + a_1x + a_2x^2 + x^3)(b_0 + x)$ and $(a_0 + a_1x + x^2)(b_0 + b_1x + x^2)$ with $a_i, b_j \in \mathbb{F}_2$, we obtain all reducible polynomials over $\mathbb{F}_2$ of degree 4. Then, by removing these reducible polynomials from the set of polynomials of degree 4, we obtain the irreducible polynomials x^4+x+1, x^4+x^3+1 and $x^4+x^3+x^2+x+1$.*

Theorem 2.17. *For $f \in \mathbb{F}[x]$, the quotient ring $\mathbb{F}[x]/(f(x))$ is a field if and only if f is irreducible in $\mathbb{F}[x]$.*

Example 2.24. *Let $f(x) = x^2 + x + 1$ be an irreducible polynomial in $\mathbb{F}_2[x]$. Then, the quotient ring $\mathbb{F}_2[x]/(f(x))$ with four elements $\{0, 1, x, y = x + 1\}$ is a field according to the Theorem 2.17. Table 2.5 shows the Cayley's tables for the addition and multiplication of polynomials modulo $f(x)$.*

$+$	0	1	x	y
0	0	1	x	y
1	1	0	y	x
x	x	y	0	1
y	y	x	1	0

$\cdot$	0	1	x	y
0	0	0	0	0
1	0	1	x	y
x	0	x	y	1
y	0	y	1	x

Table 2.5: Cayley's tables for $\mathbb{F}_2[x]/(f(x))$ with $f(x) = x^2 + x + 1 \in \mathbb{F}_2$.

From the Example 2.24 we can observe that the field $\mathbb{F}_{2^2} = \mathbb{F}_2[x]/(f(x))$ contains the finite field $\mathbb{F}_2$, i.e. the set $\{0, 1\} \subset \{0, 1, x, x + 1\}$. Therefore, $\mathbb{F}_2$ is a proper field of $\mathbb{F}_{2^2}$, and then $\mathbb{F}_{2^2}$ is an extension field of $\mathbb{F}_2$. Moreover, the finite field $\mathbb{F}_2$ is a *prime field*, since it containing no proper subfields.

Definition 2.26 (Algebraic closure of a finite field)**.** *Let p be a prime number, the algebraic closure of the finite field $\mathbb{F}_p$, denoted as $\bar{\mathbb{F}}_p$, is the infinite set of all field extensions. That is, The algebraic closure of $\mathbb{F}_p$ is the union $\bigcup_{i=1}^{\infty} \mathbb{F}_{p^i}$.*

2.4. Elliptic curves

In this section we introduce the definition of elliptic curves and some basic concepts related to them, which will be used through this thesis.

Definition 2.27 (Elliptic curve)**.** *An elliptic curve is defined by the affine Weierstrass equation*

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

where $a_1, \ldots, a_6 \in \bar{\mathbb{F}}$. If the coefficients $a_1, \ldots, a_6$ belongs to a finite field $\mathbb{F}$, then we say E is defined over $\mathbb{F}$, and write this as $E/\mathbb{F}$, the same goes for any extension field of $\mathbb{F}$.

If the characteristic of the field is different than 2 or 3, it is possible to simplify the equation given by the the Weierstrass using the following change of variables [82]

$$(x, y) \to \left(\frac{x - 3a_1^2 - 12a_2}{36}, \frac{y - 3a_1x}{216} - \frac{a_1^3 + 4a_1a_2 - 12a_3}{24} \right).$$

Which transforms Equation (2.1) into the following simplified equation called the short Weierstrass equation

$$E : y^2 = x^3 + ax + b, \tag{2.2}$$

where $a, b \in \mathbb{F}$ and whose discriminant is $\Delta = 4a^3 + 27b^2$. It is necessary that $\Delta \neq 0$ in order that $E/\mathbb{F}$ be an elliptic curve. This condition means that the polynomial $x^3 + ax + b$ has no double root, *i.e.* that this polynomial and its derivative polynomial are relative prime.

Remark 2.1. *There exist other elliptic curve models such as the Montgomery and Edwards forms, however we use the Weierstrass model to introduce the theory about elliptic curves. Although, in Chapter 9 we describe and use these other curve models.*

Definition 2.28 (j-invariant)**.** *Given an elliptic curve $E : y^2 = x^3 + ax + b$, the j-invariant of E, denoted as $j(E)$, determines the isomorphism class of E and is defined as*

$$j = -1728\frac{(4a)^3}{\Delta},$$

where $\Delta = -16(4a^3 + 27b^2)$ is the curve discriminant.

Definition 2.29 (Points on the elliptic curve)**.** *Let $\bar{\mathbb{F}}$ be the algebraic closure of the field $\mathbb{F}$ and given an elliptic curve $E/\bar{\mathbb{F}}$ defined using the Equation* (2.2)*. The set of of the points in $E/\bar{\mathbb{F}}$ is defined as*

$$E(\bar{\mathbb{F}}) = \{(x, y) \mid x, y \in \bar{\mathbb{F}}, y^2 - x^3 - ax - b = 0\}$$

Definition 2.30 ($\mathbb{F}$-rational points)**.** *For any extension field $\mathbb{F}$, the set of $\mathbb{F}$-rational points of the elliptic curve $E/\mathbb{F}$ are defined as*

$$E(\mathbb{F}) = \{(x, y) \mid x, y \in \mathbb{F}, y^2 - x^3 - ax - b = 0\}$$

An important observation is that, the set $E(\mathbb{F})$ together with the point at infinity, denoted as $\mathcal{O}$, form an Abelian group additively written. From now on, when we will use $E(\mathbb{F})$ we refer to the Abelian group and not just to the set of $\mathbb{F}$-rational points of $E/\mathbb{F}$.

In an informal way we can place the the point at infinity at the lower and upper end of the axis of the ordinates, in such a way that the vertical line to the point P in Figure 2.1b intersects the point $\mathcal{O}$.

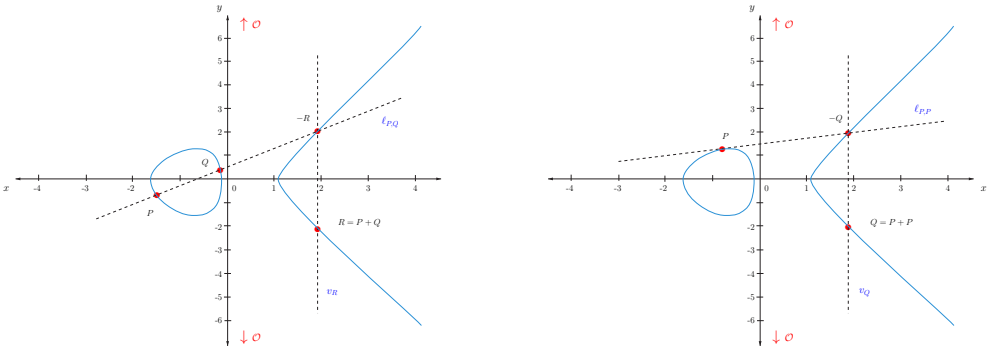
2.4.1. The group law

In this section we introduce the elliptic curve group law, and for this purpose it is convenient to view pictures of elliptic curves over $\mathbb{R}$ since are especially instructive.

The operations over points on the elliptic curve $E/\mathbb{F}$ are frequently described geometrically as: given two points $P, Q \in E(\mathbb{F})$, the addition $R = P + Q$ is computed drawing a straight line $\ell_{P,Q}$ through the points P and Q. This line intersects the elliptic curve in a third point corresponding to the point $-R$, then R is obtained by reflecting $-R$ over the abscissas axis (line v_R) as shown in Figure 2.1a; on the other hand, the doubling $Q = [2]P$ is computed drawing a tangent line $\ell_{P,P}$ to the point P, which intersects the curve in a second point corresponding to $-Q$, then in order to obtain the point Q we reflect $-Q$ over the abscissas axis (line v_Q) as shown in Figure 2.1b.

The algebraic formulas for the group law can be derived from the above description and they are presented in the following points:

- Identity: the identity element is the point at infinity $\mathcal{O}$, and for all $P \in E(\mathbb{F})$ holds that $P + \mathcal{O} = \mathcal{O} + P = P$.



(a) Point addition. **(b)** Point doubling.

Figure 2.1: Point addition and point doubling computed geometricaly over $\mathbb{R}$.

- Inverse: if $P = (x, y) \in E(\mathbb{F})$, then $(x, y) + (x, -y) = \mathcal{O}$. The point $(x, -y)$ is denoted as $-P$ and is called the negative of P. Notice that $-P$ is a point in $E(\mathbb{F})$ in the same way that $\pm\mathcal{O}$.
- Point addition: for $P = (x_P, y_P) \in E(\mathbb{F})$ and $Q = (x_Q, y_Q) \in E(\mathbb{F})$, with $P \neq \pm Q$. The addition $P + Q = (x, y)$, is computes as

$$x = \left(\frac{y_Q - y_P}{x_Q - x_P}\right)^2 - x_P - x_Q \quad \text{and} \quad y = \left(\frac{y_Q - y_P}{x_Q - x_P}\right)(x_P - x) - y_P.$$

- Point doubling: for $P = (x_P, y_P) \in E(\mathbb{F})$, where $P \neq -P$. The doubling $[2]P = (x, y)$, is computed as

$$x = \left(\frac{3x_P^2 + a}{2y_P}\right)^2 - 2x_P \quad \text{and} \quad y = \left(\frac{3x_P^2 + a}{2y_P}\right)(x_P - x) - y_P.$$

From the point addition and doubling operations we can define an extra operation called *scalar multiplication.*

Definition 2.31 (Scalar multiplication)**.** *Given a scalar $k \in \mathbb{Z}$ and a point $P \in E(\mathbb{F})$, the scalar multiplication denoted as $[k]P$ consist in compute*

$$[k]P = \underbrace{P + P + ... + P}_{k-1 \text{ additions}}.$$

There exist many algorithms to compute this operation, the most intuitive algorithm consist in apply the Horner's rule. Where we process the scalar k in its binary representation bit by bit, and doubling in each step and adding if the i-th bit is one. In Chapter 5 and Chapter 9 we will show different approaches to compute this operation efficiently.

2.4.1.1. Projective coordinates

In previous paragraphs we present the affine formulas for point addition and point doubling. Those formulas require a field inversion that can be computed with the extended Euclidean algorithm, however, this operation is considerably more costly that field multiplication. An option to perform the addition and doubling in an efficient way is using the projective version of the corresponding formulas, where it is possible to avoid the inversion at the price of computing more products.

Let $\mathbb{F}$ an finite field, and let c, d two positive integers. We can define a equivalence relation in the set $\mathbb{F}^3 \setminus \{(0,0,0)\}$ as: $(X_1, Y_1, Z_1) \sim (X_2, Y_2, Z_2)$ if $X_1 = \lambda^c X_2$, $Y_1 = \lambda^d Y_2$ and $Z_1 = \lambda Z_2$ for some $\lambda \in \mathbb{F}^*$.

The equivalence class that contains $(X, Y, Z) \in \mathbb{F}^3 \setminus \{(0,0,0)\}$ is:

$$(X : Y : Z) = \{(\lambda^c X, \lambda^d Y, \lambda Z) \mid \lambda \in \mathbb{F}^*\},$$

where $(X : Y : Z)$ is a class of projective points, while (X, Y, Z) denotes a representative element of $(X : Y : Z)$. The set of all projective points is denoted as $\mathbb{P}(\mathbb{F})$. Particularly, if $Z = 1$, we have that $(X/Z^c, Y/Z^d, 1)$ is a representative of the point $(X : Y : Z)$, which is the unique with coordinate $Z = 1$. Therefore, we have a one to one correspondence between the set of projective points:

$$\mathbb{P}(\mathbb{F})^* = \{(X : Y : Z) \mid X, Y, Z \in \mathbb{F},\ Z \neq 0\}$$

and the set of affine points:

$$\mathbb{A}(\mathbb{F})^* = \{(x, y) \mid x, y \in \mathbb{F}\}.$$

The set of projective points

$$\mathbb{P}(\mathbb{F})^0 = \{(X : Y : Z) \mid X, Y, Z \in \mathbb{F},\ Z = 0\}$$

is called line to the infinity because the points do not correspond to any affine points.

The projective form of the short equation of an elliptic curve 2.2, can be obtained by substituting $x = X/Z^c$, $y = Y/Z^d$, and doing the reduction of denominators. The points $\mathbb{P}(\mathbb{F})^*$ satisfy the projective equation, while the set of points $\mathbb{P}(\mathbb{F})^0$ correspond to the infinity point $\mathcal{O}$.

Jacobian coordinates

In Jacobian coordinates the integers c and d take the values 2 and 3, respectively. Then, the projective point (X, Y, Z), with $Z \neq 0$, corresponds to the affine point $(X/Z^2, Y/Z^3)$. In this coordinates the projective equation of the elliptic curve E is:

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

The point at the infinity $\mathcal{O}$ is the corresponding to $(1 : 1 : 0)$, while the negative point of $(X : Y : Z)$ is $(X : -Y : Z)$.

The Jacobian coordinates are the most frequently used formulas for practical implementations, because the operations over the points of an elliptic curve are more efficient using this system of coordinates. The most efficient way to perform a point addition is using mixed coordinates, that means, taking a point in affine coordinates with $Z = 1$ and the other in Jacobians. While the best manner to compute a point doubling is using just Jacobian coordinates. In the next we present the formulas for this operations.

Let $P = (X_1 : Y_1 : Z_1)$ be a point with $Z_1 \neq 0$ and $Q = (X_2 : Y_2 : 1)$ a point such that $P \neq \pm Q$, we have that the addition $R = P + Q = (X_3 : Y_3 : Z_3)$ can be computed using the following formulas [82]:

$$\begin{aligned} X_3 &= (Y_2 Z_1^3 - Y_1)^2 - (X_2 Z_1^2 - X_1)(X_1 + X_2 Z_1^2) \\ Y_3 &= (Y_2 Z_1^3 - Y_1)(X_1 (X_2 Z_1^2 - X_1)^2 - X_3) - Y_1 (X_2 Z_1^2 - X_1)^3 \\ Z_3 &= (X_2 Z_1^2 - X_1) Z_1 \end{aligned}$$

This equations have a cost of 3S and 8M, where S represents the cost of a field squaring and M denotes the cost of a field multiplication. On the other hand, the point doubling of P is

computed through the following equations:

$$\begin{aligned} X_2 &= (3X_1^2 + aZ_1^4)^2 - 8X_1Y_1^2 \\ Y_2 &= (3X_1^2 + aZ_1^4)(4X_1Y_1^2 - X_3) - 8Y_1^4 \\ Z_2 &= 2Y_1Z_1 \end{aligned}$$

This equations have a cost of 6S and 3M, although, when $a = 0$ this operation has a cost of 4S and 3M.

In Table 2.6 we compare the cost of the point addition and doubling using affine formulas for a general Weierstrass curve versus these operations using jacobian coordinates. From the table we can conclude that if the cost of an finite field inversion is more than 8 field multiplications, then the computations in jacobian coordinates allow a better performance.

Coordinates	**Point addition**	**Point doubling**
Affine	2M + 1S + I	2M + 1S + I
Mixed jacobian	8M + 3S	3M + 6S

Table 2.6: Cost comparison between affine and mixed jacobian coordinates for point addition and doubling.

2.4.2. Elliptic curves over finite fields

Let p a prime number and let $q = p^n$, where n is a positive integer. Given a finite field $\mathbb{F}_q$ with characteristic p, the $\mathbb{F}_q$-rational points in the elliptic curve forms the group $E(\mathbb{F}_q)$, such that for every affine point $P = (x_P, y_P) \in E(\mathbb{F}_q)$ the values x_P and y_P are elements in $\mathbb{F}_q$.

Definition 2.32 (Cardinality of the group $E(\mathbb{F}_q)$)**.** *The cardinality of the group $E(\mathbb{F}_q)$, denoted as $\#E(\mathbb{F}_q)$, is the number of points that satisfy the elliptic curve equation and whose coordinates belongs to $\mathbb{F}_q$.*

Given that Equation (2.2) has at most two solutions for each $x \in \mathbb{F}_q$, we know that $\#E(\mathbb{F}_q) \in [1, 2q + 1]$. However, the Hasse boundary establish more precise bounds for $\#E(\mathbb{F}_q)$:

Theorem 2.18. *(Hasse boundary). Let E an elliptic curve defined over $\mathbb{F}_q$, the interval*

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

is called the Hasse boundary.

Alternatively, we can write the cardinality of $E(\mathbb{F}_q)$ as

$$\#E(\mathbb{F}_q) = q + 1 - t,$$

where t represents the *trace* of an elliptic curve E over the finite field $\mathbb{F}_q$. Since the trace is bounded by $2\sqrt{q} \leq t \leq 2\sqrt{q}$ and it is relatively smaller than q we can say that $\#E(\mathbb{F}_q) \approx q$.

Elliptic curves can be divided into two families: supersingular and ordinary elliptic curves. The following definition highlights the difference between these two families:

Definition 2.33 (Supersingular and ordinary elliptic curve)**.** *Given an elliptic curve $E/\mathbb{F}_q$ with $q = p^n$ and $\#E(\mathbb{F}_q) = q + 1 - t$, we say that $E/\mathbb{F}_q$ is supersingular [164] if p divides t, that means, E is supersingular iff $t \equiv 0 \mod p$, which is true iff $\#E(\mathbb{F}_q) \equiv 1 \mod p$; otherwise, the curve is called ordinary.*

Definition 2.34 (Torsion points)**.** *Let $E/\mathbb{F}_p$ an elliptic curve and $\bar{\mathbb{F}}_p$ the algebraic closure of $\mathbb{F}_p$. For any positive integer r, we define the set of r-torsion points of $E(\bar{\mathbb{F}}_p)$, denoted as $E(\bar{\mathbb{F}}_p)[r]$, as the set*

$$E(\bar{\mathbb{F}}_p)[r] = \{P \in E(\bar{\mathbb{F}}_p) | rP = \mathcal{O}\}.$$

Let n be a positive integer, the set of $\mathbb{F}_{p^n}$-rational points of r-torsion for $\mathbb{F}_p \subseteq \mathbb{F}_{p^n} \subset \bar{\mathbb{F}}_p$, denoted as $E(\mathbb{F}_{p^n})[r]$, is:

$$E(\mathbb{F}_{p^n})[r] = \{P \in E(\mathbb{F}_{p^n}) | rP = \mathcal{O}\}.$$

Theorem 2.19 (Structure of the r-torsion group)**.** *Let $E/\bar{\mathbb{F}}_p$ be an elliptic curve defined over a finite field of characteristic p. For any non-zero integer r relative prime to p, the r-torsion group of E is isomorphic to the direct product $\mathbb{Z}/r\mathbb{Z} \times \mathbb{Z}/r\mathbb{Z}$. On the other hand, if $r = p^e$ for an integer e, then either $E[r] \cong \{\mathcal{O}\}$ or $E[r] \cong \mathbb{Z}/r\mathbb{Z}$.*

Corollary 2.2 (Cardinality of the r-torsion group)**.** *For any non-zero integer r, the r-torsion group of an elliptic curve contains r^2 points.*

Definition 2.35. *(Embedding degree). For two prime numbers p and r, given the finite field $\mathbb{F}_p$ and considering the elliptic curve $E/\mathbb{F}_p$ such that $r \mid \#E(\mathbb{F}_p)$. Let k a positive integer, we say that k is the embedding degree of $E/\mathbb{F}_p$ with respect to p and r, if k is the smaller positive integer that satisfies $r|p^k - 1$.*

Proposition 2.1. *Let r be an integer that divides the cardinality of an elliptic curve over a finite field $\mathbb{F}_q$ and such that r does not divides $q - 1$. The r-torsion group is included in the set of points of the elliptic curve whose coordinates belong to the extension of degree k of $\mathbb{F}_q$ if and only if r divides $q^k - 1$.*

2.4.2.1. Twist of a curve

In this section we give the definition and a theorem that explicitly describes the equations of the possible twisted elliptic curves.

Definition 2.36 (Twisted elliptic curve)**.** *Let E and E' be two elliptic curves, we say that E' is the twist of E, if and only if E and E' have the same j-invariant and are isomorphic over the algebraic closure of a finite field $\mathbb{F}$.*

Particularly, given an elliptic curve $E/\mathbb{F}_p$ with embedding degree k, if the finite field $E(\mathbb{F}_p)$ has a subgroup of prime order r, Hess *et al.* [85] show that, there exist a twist curve E' of E, defined over $\mathbb{F}_{p^{k/d}}$, where $d|k$, with $r|\#E'(\mathbb{F}_{p^{k/d}})$, such that there exist an isomorphism:

$$\Psi_d : E'(\mathbb{F}_{p^{k/d}}) \to E(\mathbb{F}_{p^k}),$$

where the integer d is the degree of the twist curve E'.

Theorem 2.20. *Let E be an elliptic curve defined by the short Weierstrass of Equation (2.2) over an extension $\mathbb{F}_q$ of a finite field $\mathbb{F}_p$, for a prime number p, k a positive integer such that $q = p^k$. According to the value of k, the potential degrees for a twist curve are $d \in \{2, 3, 4, 6\}$.*

The explicit isomorphism $\Psi_d : E' \to E$ are presented below:

- $d = 2$. Let $\nu \in \mathbb{F}_{p^{k/2}}$ be such that the polynomial $X^2 - \nu$ is reducible over $\mathbb{F}_{p^{k/2}}$. The equation of E' over $\mathbb{F}_{p^{k/2}}$ is $E' : \nu y^2 = x^3 + ax + b$. The isomorphism Ψ_2 is given by

$$\begin{aligned} \Psi_2 : E'(\mathbb{F}_{p^{k/2}}) &\to E(\mathbb{F}_{p^k}) \\ (x, y) &\mapsto (x, y\nu^{1/2}). \end{aligned}$$

- $d = 3$. The elliptic curve E admits a twist of degree 3 if and only if $a = 0$. Let $\nu \in \mathbb{F}_{p^{k/3}}$ be such that the polynomial $X^3 - \nu$ is irreducible over $\mathbb{F}_{p^{k/3}}$. The equation of E' is $y^2 = x^3 + \frac{b}{\nu}$. The isomorphism Ψ_3 is given by

$$\begin{aligned} \Psi_3 : E'(\mathbb{F}_{p^{k/3}}) &\rightarrow E(\mathbb{F}_{p^k}) \\ (x, y) &\mapsto (x\nu^{1/3}, y\nu^{1/2}). \end{aligned}$$

- $d = 4$. The elliptic curve E admits a twist of degree 4 if and only if $b = 0$. Let $\nu \in \mathbb{F}_{p^{k/4}}$ be such that the polynomial $X^4 - \nu$ is irreducible over $\mathbb{F}_{p^{k/4}}$. The equation of E' is $y^2 = x^3 + \frac{a}{\nu}x$. The isomorphism Ψ_4 is given by

$$\begin{aligned} \Psi_4 : E'(\mathbb{F}_{p^{k/4}}) &\rightarrow E(\mathbb{F}_{p^k}) \\ (x, y) &\mapsto (x\nu^{1/2}, y\nu^{3/4}). \end{aligned}$$

- $d = 6$. The elliptic curve E admits a twist of degree 6 if and only if $a = 0$. Let $\nu \in \mathbb{F}_{p^{k/6}}$ be such that the polynomial $X^6 - \nu$ is irreducible over $\mathbb{F}_{p^{k/6}}$. The equation of E' is $y^2 = x^3 + \frac{b}{\nu}$. The isomorphism Ψ_6 is given by

$$\begin{aligned} \Psi_6 : E'(\mathbb{F}_{p^{k/6}}) &\rightarrow E(\mathbb{F}_{p^k}) \\ (x, y) &\mapsto (x\nu^{1/3}, y\nu^{1/2}). \end{aligned}$$

Part I

Integer-factorization-based cryptography

Chapter 3

Integer and finite field arithmetic

A fundamental part of most cryptographic schemes is the set of operations on the ring of integers $\mathbb{Z}$. Because, once it is possible to compute operations as addition, subtraction, multiplication, and squaring in the integers; we can be able to build more complex mathematical structures. For instance, in the case that we need to work with a cryptographic scheme based on elliptic curves over a finite field, it is necessary first to build the finite field whose main operations rely on integers, and then define the operations that compose the elliptic curve structure over such field. Given that the associated cost of the operations in finite fields and, therefore, the cost of the operations in elliptic curves are noticeably influenced by the cost of the used algorithms for the underlying integer arithmetic. It results extremely important that integer arithmetic be performed as efficiently as possible, in order to have an efficient implementation of the cryptographic scheme.

The present chapter is dedicated to describe the practical considerations of an efficient implementation of integer and finite field arithmetic. At first, we will explain how large integers can be represented internally on computers. Then we describe the main algorithms used to compute arithmetic operations on those large integers. Finally, we inspect the algorithms used in the finite field operations and its practical considerations of implementation.

3.1. Representation of large integers

The largest quantity that is possible to manipulate directly in a computer using the processor instruction set, depends on the size in bits of the processor registers. Consequently the size of a register, called a *word*, is one of the main characteristics of a processor to take into account when we talking about implementation of arithmetic on large integers, because the amount of work that the processor will perform to process such integers depends on its size in words. In present-day computers commonly we found micro-architectures using registers with words of $w = \{32, 64\}$ bits, although, there may also be micro-architectures with 8- or 16-bit words like that found in micro-controllers or smart cards.

Large integers refers to integers that need hundreds or thousands of bits to be represented, which cannot be manipulated by a contemporary computer directly. We say that a single precision or a 1-*word integer* is an integer that needs only one word to be represented, and therefore, it can be manipulated using the processor instruction set of a computer. *i.e.* for a w-bit micro-architecture, a 1-word integer a is an integer such that $0 \leq a < 2^w$. On the other hand, an integer that fits in more of one word is called a multi-precision integer. And then, if such integer needs n words to be represented, we say that it is an n-*word integer*. In order to represent a multi-precision integer a, we must first define a *radix* $r \geq 2$. This

value is commonly selected as $r = 2^w$, where as before w represent the bit-size of a processor register. Using this value r, we can write every integer $a \geq 0$ in a unique way as the sum

$$a = \sum_{i=0}^{n-1} a_i r^i = a_{n-1} r^{n-1} + \cdots + a_1 r + a_0, \tag{3.1}$$

where $0 \leq a_i < r$, the coefficient a_{n-1} has the largest index i for which $a_{n-1} > 0$, and $n = \lceil \lceil \log_2(a) \rceil / r \rceil$. The sum in Equation (3.1) corresponds to the representation of an integer a in radix-r, and we will denoted as $(a_{n-1}, \ldots, a_0)_r$.

3.2. Arithmetic instructions in processors

In order to reduce the latency of integer arithmetic operations, trough this chapter we took advantage of the `ADCX/ADOX` and `MULX` instructions. Which were specially designed for speeding-up integer multi-precision arithmetic operations, and that are available in the newest Intel and AMD micro-architectures.

Starting from the Intel Haswell micro-architecture, the instruction `MULX` was introduced as a part of the Bit Manipulation Instruction set (BMI2) [46]. `MULX` is an extension of the traditional 64-bit multiplication instruction `MUL`, that computes the multiplication of two unsigned 64-bit operands without affecting the arithmetic flags. This feature allows to combine `MULX` with addition instructions without affecting the carry chain state. Additionally, `MULX` uses a three-operand code that allows the programmer to choose the registers that will be used for storing the upper and lower part of the output product. Thus allowing to preserve the data stored in the input registers, and allowing to reduce the number of `MOV` instructions used to move the operands to the input registers and the result to the desired output registers.

On the other hand, the set of instructions `ADX` was introduced from the Intel Broadwell micro-architecture and includes the `ADCX/ADOX` instructions [76]. Which are extensions of the traditional 64-bit addition instructions `ADD/ADC`, that were designed for handling two independent carry chains. These new instructions compute an unsigned 64-bit integer addition without modifying the carry flag (`CF`) and the overflow flag (`OF`), respectively. In this way, the additions can be computed using as carry the value stored in the `CF` and the `OF` flags, allowing that `ADCX` and `ADOX` instructions can be executed concurrently.

In the remaining of this chapter we will study the combined usage of these novel instructions, in order to get an efficient implementation of the arithmetic operations on the integers and finite fields.

3.2.1. AVX2 instruction set

We also take advantage of the `AVX2` instruction set introduced in the Intel Haswell micro-architecture [47]. `AVX2` is an extension from `AVX`, which allows to compute Single Instruction Multiple Data (SIMD) operations using 256-bit vector registers. This instruction set provides operations supporting integer arithmetic and other useful computations, that are able to compute up to four simultaneous 64-bit operations over the values stored in the vector registers. In terms of performance, we will expect a speedup factor of four from the simultaneous execution of 64-bit operations. Nevertheless, this acceleration can be attained only for some instructions, because some factors like the execution latency and throughput, and the number of execution units available in the target micro-architecture reduce this acceleration. For example, in this chapter the expected acceleration is limited by the size of the `AVX2` multiplier, among other factors.

For the main purposes of this chapter, we mainly benefit from the `AVX2` instructions detailed in the following: `mm256_mul_epu32` that is able to compute four products of $32 \times$

32 bits, storing the four 64-bit results on a 256-bit vector register; `mm256_add_epi32` and `mm256_sub_epi32` that compute eight simultaneous 32-bit additions/subtractions, without handling the input/output carry and borrow, respectively; `mm256_slli_epi32` and `mm256_srli_epi32` that compute eight 32-bit logical shifts using the same fixed shift displacement for every word stored in the vector register; `mm256_shuffle_epi32` that shuffles 32-bit values of the source vector in the destination vector at the locations selected by a control operand; `mm256_xor_si256` and `mm256_and_si256` that compute the `XOR`/`AND` of two 256-bit vector registers; `mm256_cmpgt_epi32` that returns a vector with the values $2^{32}-1$ and zero depending if the comparison of the 32-bit integers in the vector register is true or not.

3.3. Integer arithmetic

In this section we present how the algorithms used to compute the operations of addition, subtraction, multiplication and modular reduction over the integers, can be efficiently performed taking advantage of the instructions presented in the previous section. Besides, through this section we also detail some practical considerations to take into account when a fast and secure implementation of these operations is desirable.

3.3.1. Addition and subtraction

Integer addition and subtraction of two multi-precision integers a and b can be performed through Algorithms 1 and 2, respectively. In both algorithms, it is possible to fix to n the number of words that the operands can have. This implies that, for adding or subtracting integers with lengths less than n, we must first pad the integers with as many zeros as necessary so that they have length n. The objective of fixing the number of words used to represent an integer, is to be able of unroll the main loops in both algorithms in order to obtain a constant-time implementation with a faster performance.

Algorithm 1 Integer addition

Input: Two n-word integers $a = (a_{n-1}, \ldots, a_0)_r$ and $b = (b_{n-1}, \ldots, b_0)_r$.
Output: The $(n+1)$-word integer $c = (c_n, \ldots, c_0)_r$ such that $c = a + b$.

1: carry $\leftarrow 0$
2: **for** $i = 0$ **to** $n - 1$ **do**
3: $\quad d \leftarrow a_i + b_i + \text{carry}$
4: $\quad c_i \leftarrow d \mod r$ $\quad \triangleright\ 0 \leq c_i < r$
5: $\quad \text{carry} \leftarrow \lfloor d/r \rfloor$ $\quad \triangleright$ carry $= 0$ or 1
6: **end for**
7: $c_n \leftarrow$ carry
8: **return** c

Considering that, the operations in the lines inside the loops in both algorithms can be performed using a single assembly instruction, when the instructions `ADD/ADC` or `ADCX/ADOX` and `SUB/SBB` are available in the target processor. The Algorithms 1 and 2 can be implemented at a cost of $(n+1)$-word additions and $(n+1)$-word subtractions, respectively.

Nevertheless, for the subtraction $c = a - b$ when $a < b$, the $(n+1)$-th word of c stores -1 which means that the result corresponds to the two's complement representation of $a - b$. In order to obtain $|a - b|$ we can conditionally compute $a - b$ or $b - a$ depending if $a \geq b$ or $b \geq a$, respectively. However, generally it is desirable to compute this operation in constant time in order to thwart side-channel attacks. For this purpose, we can avoid the conditional execution by computing unconditionally the subtraction $c = a - b$ followed of the Algorithm 2 using as inputs $a_i = (\sim c_n)$ `&` c_i and $b_i = c_n$ `&` c_i for $0 \leq i < n$. At the end the sign of

Algorithm 2 Integer subtraction

Input: Two n-word integers $a = (a_{n-1}, \ldots, a_0)_r$ and $b = (b_{n-1}, \ldots, b_0)_r$.
Output: The $(n+1)$-word integer $c = (c_n, \ldots, c_0)_r$ such that $c = a - b$.

1: borrow $\leftarrow 0$
2: **for** $i = 0$ **to** $n - 1$ **do**
3: $\quad d \leftarrow a_i - b_i + \text{borrow}$
4: $\quad c_i \leftarrow d \mod r$ ▷ $0 \leq c_i < r$
5: $\quad \text{borrow} \leftarrow \lfloor d/r \rfloor$ ▷ borrow $= 0$ or -1
6: **end for**
7: $c_n \leftarrow$ borrow
8: **return** c

the result is given by the value of the $(n + 1)$-th word of c. This procedure has a cost of $2(n + 1)$-word additions, $2n$ logical `AND`s and one logical `NOT` instructions.

3.3.2. Multiplication

Multiplication is a very important operation in any arithmetic system. This is due to the fact that the multiplication significantly influences the cost of some arithmetic operations such as modular reduction, exponentiation, among others computations whose algorithms depend to a large extent on the cost of the multiplication method that is used.

The most intuitive way to compute an integer multiplication is using the schoolbook multiplication method shown in Algorithm 3, which has a quadratic complexity with respect to the word-size of the operands. In the same manner that for the addition and subtraction algorithms, we can fix the word-size of the operands to n and take advantage of the assembly instructions `MULX`, `ADD/ADC` or `ADCX/ADOX`. By doing so, a naive implementation of the Algorithm 3 would have a cost of n^2 multiplications and $4n^2$ additions.

Algorithm 3 Schoolbook method for integer multiplication

Input: Two integers $a = (a_{n-1}, \ldots, a_0)_r$ and $b = (b_{m-1}, \ldots, b_0)_r$.
Output: The $(n + m)$-word integer $c = (c_{n+m-1}, \ldots, c_0)_r$ such that $c = a \cdot b$.

1: Set c equal to zero. ▷ For the step 5
2: **for** $i = 0$ **to** $n - 1$ **do**
3: $\quad e \leftarrow 0$
4: $\quad$ **for** $j = 0$ **to** $m - 1$ **do**
5: $\quad\quad d \leftarrow a_i \cdot b_j + c_{i+j} + e$ ▷ $0 \leq d < r^2$
6: $\quad\quad c_{i+j} \leftarrow d \mod r$ ▷ Lower part of d
7: $\quad\quad e \leftarrow \lfloor d/r \rfloor$ ▷ Highest part of d
8: $\quad$ **end for**
9: $\quad c_{m+i} \leftarrow e$
10: **end for**
11: **return** c

We can observe that the efficiency of this method mainly depends on the selection of the partial products and the way that they are added. Commonly, integer multiplication can be carried out following one of these strategies: the product scanning strategy in which all the partial products used to obtain the final result of one column are computed and added; and the operand scanning strategy where the multiplicand operand is multiplied by each word of the multiplier, and after performing all multiplications, the partial products are properly shifted and added to obtain the output product.

When the product scanning strategy is used two drawbacks arise: first, it is necessary to have enough processor registers to maintain all the partial products corresponding to a determined column; and given that multiplications involved in the result of a column do not share operands, then more `MOV` instructions are needed to place the operands in the input registers used for `MULX`. For instance, multiplying two 3-word integers $a = (a_2, a_1, a_0)_r$ and $b = (b_2, b_1, b_0)_r$ using this technique require to compute the value of the column c_2 pointed out in Figure 3.1. This computation can be done using the code shown in Listing 3.1 taking into account that the results of the columns c_0-c_4 are stored in the registers `r8-r12`, respectively.

```
1  //c[rdi] = a[rsi] * b[rcx]
2  // rdx = 8(rcx)
3
4  mulx  8(rsi), rax, rbx
5  mov    (rcx), rdx
6  mulx 16(rsi), r13, r14
7  mov  16(rcx), rdx
8  mulx   (rsi), r15, rbp
9  add      rax, r10
10 adc      r14, r11
11 adc        0, r12
12 add      r13, r10
13 adc      rbx, r11
14 adc        0, r12
15 add      r15, r10
16 adc      rbp, r11
17 adc        0, r12
```

Listing 3.1: Product scanning strategy for multiplication.

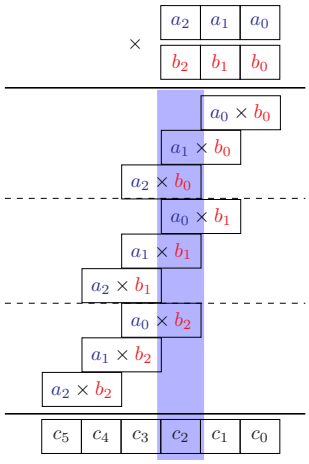


Figure 3.1: Schoolbook 3-word integer multiplication (product scanning strategy).

From Listing 3.1 we can observe that the cost of compute the column c_2 is three `MULX`, two `MOV` and nine `ADD/ADC` instructions. Besides, it is not possible to take advantage of the `ADCX/ADOX` instructions concurrency given the dependency between the addition operations.

On the other hand, when the operand scanning strategy is used those disadvantages disappear. Because the partial products can be added as they are computed, taking full advantage of `ADCX/ADOX` instructions as shown Listing 3.2. And moreover, given that a word of the operand keeps fixed during the computation of a row, as shown Figure 3.2, it is possible to save the `MOV` instructions needed to place the operands in the registers used by `MULX`. Following this strategy the computation of the second row has a cost of three `MULX`, one `MOV`, one `XOR` and seven `ADCX/ADOX` instructions.

```
1  //c[rdi] = a[rsi] * b[rcx]
2
3  xor       r12, r12
4  mov    8(rcx), rdx
5  mulx    (rsi), rax, rbx
6  adcx       rax, r9
7  adox       rbx, r10
8  mulx   8(rsi), rax, rbx
9  adcx       rax, r10
10 adox       rbx, r11
11 mulx 16(rsi), rax, rbx
12 adcx       rax, r11
13 adox       rbx, r12
14 adc          0, r12
```

Listing 3.2: Operand scanning strategy for multiplication.

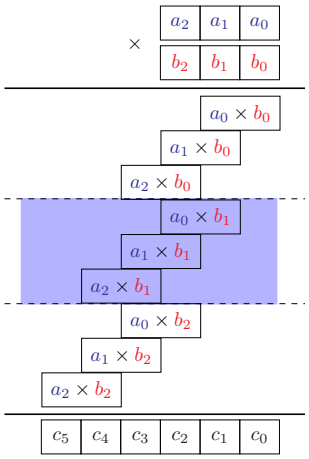


Figure 3.2: Schoolbook 3-word integer multiplication (operand scanning strategy).

Nevertheless, both strategies are limited by the number of general purpose registers available in the target architecture, since these registers are used to maintain the multiplication result and its partial products. For this reason and Algorithm 3 complexity the schoolbook multiplication method is useful just for operands with a small word-size.

3.3.2.1. Karatsuba method for multiplication

The Karatsuba multiplication method [105], was proposed by Anatolii Karatsuba in 1963 for multi-digit numbers or polynomials multiplication. Using a divide and conquer strategy, this method can be recursively used in order to significantly reduce the number of multiplications required in a multiplication, at cost of increase in the number of needed additions. The binary splitting of an n-word multiplication is performed writing the operands as $a = a_L + a_H x$ and $b = b_L + b_H x$ where $x = r^{n/2}$ and a_L, a_H, b_L, and b_H are $\frac{n}{2}$-word integers. Then, the values c_L, c_M and c_H are computed as

$$c_L = a_L \cdot b_L, \qquad c_M = (a_L + a_H) \cdot (b_L + b_H), \qquad c_H = a_H \cdot b_H.$$

And finally, the multiplication $c = a \cdot b$ is calculated as

$$c = c_L + (c_M - c_L - c_H)x + c_H x^2, \tag{3.2}$$

at a cost of three $\frac{n}{2}$-word multiplications (instead of the four multiplications that would be required when the schoolbook method is used), two $\frac{n}{2}$-word additions, one n-word addition and two n-word subtractions. However, computing the value c_M could require an $(\frac{n}{2}+1)$-word multiplication, which can be performed as

$$\begin{aligned} c_M &= (a_L + a_H) \cdot (b_L + b_H) \\ &= (d_L + d_H x) \cdot (e_L + e_H x) \\ &= (d_L \cdot e_L) + (d_L \cdot e_H)x + (e_L \cdot d_H)x + (d_H \cdot e_H)x^2 \end{aligned}$$

where d_H and e_H store the carry generated by $a_L + a_H$ and $b_L + b_H$ respectively. In this way, the multiplication is carry out computing the product $d_L \cdot e_L$ and adding $d_L x$, $e_L x$, and x^2 to it conditionally, depending on the value of d_H and e_H.

Given that the complexity of this method is $O(n^{\log_2 3}) \approx O(n^{1.58})$ when the word-size n is even, it results very useful to multiply large integers. A technique that is generally used to compute large integers multiplication consist of combining the schoolbook and Karatsuba multiplication methods, in §4 we will presented an analysis and implementation of this technique.

3.3.3. Squaring

Similarly to integer multiplication, squaring is a very important operation within an arithmetic system. This importance can be seen mainly in the modular exponentiation methods, which will be addressed latter in this chapter. However, unlike multiplication, this operation can be performed in a more efficient way, given that consists of multiplying the input operand by itself. This fact, allows us to save some partial products during the squaring computation.

The schoolbook multiplication method can be used to perform a squaring, but taking advantage of repeated partial products found during the computation. Such method is shown in Algorithm 4 and has a complexity of $O(\frac{n^2+n}{2})$. Using the assembly instructions described in previous paragraphs, a naive implementation should have a cost of $\frac{n^2-n}{2}$ word multiplications and n word squarings. However, since there is no squaring instruction in the processors, the associated cost to the Algorithm 4 is given using only multiplications and additions. Thus, the cost of the algorithm is $\frac{n^2+n}{2}$ word multiplications and $\frac{9n^2-5n}{2}$ word additions.

Algorithm 4 Schoolbook method for integer squaring

Input: An n-word integer $a = (a_{n-1}, \ldots, a_0)_r$.
Output: The $(2n)$-word integer $c = (c_{2n-1}, \ldots, c_0)_r$ such that $c = a^2$.

1: Set c equal to zero.
2: **for** $i = 0$ **to** $n - 1$ **do**
3: $\quad d \leftarrow a^2 + c_{2i}$ ▷ $0 \leq d < r^2$
4: $\quad c_{2i} \leftarrow d \mod r$
5: $\quad e \leftarrow \lfloor d/r \rfloor$ ▷ $0 \leq e < r$
6: $\quad$ **for** $j = i + 1$ **to** $n - 1$ **do**
7: $\quad\quad d \leftarrow 2(a_i \cdot a_j) + c_{i+j} + e$ ▷ $0 \leq d \leq 2r(r-1)$
8: $\quad\quad c_{i+j} \leftarrow d \mod r$
9: $\quad\quad e \leftarrow \lfloor d/r \rfloor$ ▷ $0 \leq e \leq 2(r-1)$
10: $\quad$ **end for**
11: $\quad c_{n+i} \leftarrow e$
12: **end for**
13: **return** c

The efficiency of Algorithm 4 depends on the way that the partial products are computed and added, just like in the schoolbook multiplication algorithm. So, we can use the product or operand scanning strategies for the integer squaring. Although, we found the same pros and cons as those that arose for the schoolbook integer multiplication. Therefore, we used the operand scanning strategy where the partial products can be added as they are computed. In Figure 3.3 we can observe the example of the squaring of a 3-word integer $a = (a_2, a_1, a_0)_r$ using this strategy, on it, the red rectangles point out that the product must be multiplied by 2 before we add it.

```
1  //c[rdi] = a[rsi] * a[rsi]
2
3  xor       rax, rax
4  mov     (rsi), rdx
5  mulx    (rsi), r8, r9
6  mulx  8(rsi), rax, r10
7  adox      rax, rax
8  adox      r10, r10
9  adcx      rax, r9
10 adc         0, r10
11 mulx 16(rsi), rax, r11
12 adox      rax, rax
13 adox      r11, r11
14 adcx      rax, r10
15 adc         0, r11
```

Listing 3.3: Operand scanning strategy for squaring.

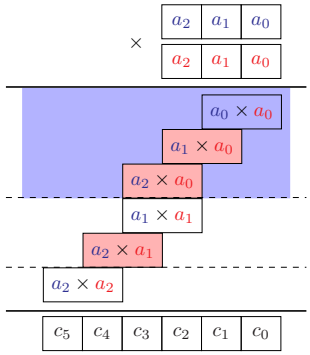


Figure 3.3: Schoolbook 3-word integer squaring (operand scanning strategy).

Listing 3.3 shown the assembly code used to compute the result of the row highlighted in blue in Figure 3.3, taking into account that the results c_0-c_3 are stored in the registers `r8-r11`, respectively. As can be observed the cost of the computation of the row is three `MULX`, one `MOV`, one `XOR` and eight `ADCX/ADOX` instructions. Given the fact that squaring algorithm needs more additions than the multiplication one, the Algorithm 4 could be useful for integers with smaller word-sizes than the supported by the multiplication algorithm. This directly depends on the cost associated with the processor's addition instruction.

3.3.3.1. Karatsuba method for squaring

The Karatsuba multiplication method [105] can be also used in order to significantly reduce the number of word multiplications required in a squaring, at cost of increase the needed additions. The binary splitting of an n-word squaring is performed writing the input operand as $a = a_L + a_H x$ where $x = r^{n/2}$, and a_L and a_H are $\frac{n}{2}$-word integers. Then, the values c_L, c_M and c_H are computed as

$$c_L = a_L^2, \qquad c_M = (a_L + a_H)^2, \qquad c_H = a_H^2.$$

And finally the squaring $c = a^2$ is calculated as

$$c = c_L + (c_M - c_L - c_H)x + c_H x^2, \tag{3.3}$$

at cost of three $\frac{n}{2}$-word squarings, two $\frac{n}{2}$-word additions, one n-word addition and two n-word subtractions. However, computing the value c_M could require an $(\frac{n}{2}+1)$-word squaring which can be performed as

$$\begin{aligned} c_M &= (a_L + a_H)^2 \\ &= (d_L + d_H x)^2 \\ &= d_L^2 + 2(d_L \cdot d_H)x + d_H^2 x^2, \end{aligned}$$

where d_H stores the carry generated by the addition $a_L + a_H$. In this way, the squaring is carry out computing the product d_L^2 and adding $2d_L x$ and x^2 to it conditionally, depending on the value of d_H.

A problem that arises when this method is used, is that the conditional addition must be done in constant time in order to reduce the risk of side channel attacks, which increases the cost of the operation. A way to perform the integer squaring, which does not needs a

conditional addition and reduces the number of subtractions, is computing the Karatsuba formula 3.3 as follow

$$c = c_L + 2(a_L \cdot a_H)x + c_H x^2, \tag{3.4}$$

this can be done because

$$\begin{aligned}(c_M - c_L - c_H) &= (a_L + a_H)^2 - c_L - c_H \\ &= a_L^2 + 2(a_L \cdot a_H) + a_H^2 - c_L - c_H \\ &= 2(a_L \cdot a_H).\end{aligned}$$

Using this formula, Karatsuba has a cost of two $\frac{n}{2}$-word squarings, one $\frac{n}{2}$-word multiplication, and two n-word additions.

In the same manner that for multiplication, we can use the technique that combines the schoolbook and Karatsuba squaring methods to improve the performance of large integers squaring. We will present an analysis of this technique for squaring in §4.

3.3.4. Modular reduction

Modular reduction is a fundamental part in the construction of finite field arithmetic, because all operations in a finite field must be reduced modulo a prime number p. Therefore, the cost of the chosen modular reduction algorithm significantly influences the associated cost of operations such as modular multiplication, modular squaring and modular exponentiation.

Intuitively, this reduction could be performed using the Euclidean division algorithm and then computing the corresponding remainder, but this method implies an integer division which is a very costly operation. In this section we will see two approaches to perform the modular reduction, which are based on the pre-computation of a particular value μ that is closely related to the used modulo p. This pre-computed value is utilized to replace the costly division by roughly the cost of a multiplication by μ.

3.3.4.1. Montgomery modular reduction

In 1985 Peter L. Montgomery proposed a novel method to compute the modular reduction without using a division [131], the main idea behind of his method consists in changing the numbers representation to the so called *Montgomery domain*. In order to do this change of representation, it is necessary first to select a Montgomery radix R which is a power of two such that $r^{n-1} < p < r^n$. Generally, $R = r^n$ is chosen where as before r represents the size of a processor register and n is the word-size of the used prime p. Finally, we can map an element $a \in \mathbb{Z}_p$ to its Montgomery's representation $\tilde{a}$ by computing

$$\tilde{a} = a \cdot R \mod p.$$

Consequently, at the start and the end of the computations a transformation to and from this representation is needed.

Given two elements $a, b \in \mathbb{Z}_p$ in its Montgomery's representation $\tilde{a}$ and $\tilde{b}$, respectively. The integer multiplication $\tilde{a} \cdot \tilde{b}$ produces the value $c' = a \cdot b \cdot R^2$ with $0 \leq c' < p^2$, which does not correspond to the Montgomery's representation of $c = a \cdot b$. Then, with the aim of obtain the Montgomery's representation of c, which is computed as $\tilde{c} = c' \cdot R^{-1} \mod p$, we use the following equation

$$\tilde{c} = \frac{c' + (\mu \cdot c' \mod R) \cdot p}{R} \equiv c' \cdot R^{-1} \mod p. \tag{3.5}$$

Where the pre-computed value μ is calculated as $\mu = -p^{-1} \mod R$, and the modulo and division operations by R can be efficiently performed using fast right/left n-word shift operations. It can be shown that when $0 \leq c' < p^2$, the result $\tilde{c}$ in Equation (3.5) is in the interval $[0, 2p)$ and at most a single conditional subtraction is needed to obtain $0 \leq \tilde{c} < p$.

The reduction algorithm for multi-precision arithmetic based on Equation (3.5), named by its author as REDC, is presented in Algorithm 5. This algorithm can be implemented at a cost of n^2+n word multiplications if the pre-computed Montgomery constant μ is adjusted to $\mu = -p^{-1} \mod r$.

Algorithm 5 REDC algorithm.

Input: A $2n$-word integer c', the n-word modulo p and $\mu = -p^{-1} \mod r$.
Output: An n-word integer $\tilde{c}$ such that $\tilde{c} = c' \cdot r^{-n} \mod p$.

1: **for** $i = 1$ **to** n **do**
2: $\quad t \leftarrow c' \mod r$
3: $\quad q \leftarrow t \cdot \mu \mod r$
4: $\quad c' \leftarrow (c' + q \cdot p)/r$
5: **end for**
6: $\tilde{c} \leftarrow c'$
7: **if** $\tilde{c} \geq p$ **then**
8: $\quad \tilde{c} \leftarrow \tilde{c} - p$
9: **end if**
10: **return** $\tilde{c}$

In a constant time implementation the conditional subtraction described above and showed in the Step 8 of Algorithm 5, can be omitted by using the technique introduced by Walter in [163]. This technique consist of using a redundant representation of the elements in Montgomery's representation and select a Montgomery radix such that $4p < R$. For instance, if we allow that $\tilde{a}, \tilde{b}$ belong to $\mathbb{Z}_{2p}$ instead of $\mathbb{Z}_p$, then the value $\tilde{c} = \tilde{a} \cdot \tilde{b} \cdot R^{-1}$ in the Step 6 of the REDC algorithm is also upper bounded by $2p$ and can be reused in subsequent computations without the need of a conditional subtraction. At the end of the whole computations the result is reduced at the cost of a single subtraction.

It is worth mentioning that the Montgomery reduction algorithm is also useful to convert elements between normal and Montgomery representations. For instance, given an element $a \in \mathbb{Z}_p$ its Montgomery's representation can be computed as $\tilde{a} = \text{REDC}(a \cdot R^2)$, and in order to convert the element $\tilde{a}$ to its normal representation we can compute $a = \text{REDC}(\tilde{a})$.

Montgomery-friendly primes

In several works, different authors have exploited a special class of prime moduli, which permit to reduce the number of word multiplications needed in the REDC algorithm [2, 80, 111, 119]. The modulus contained in this class are sometimes named as as Montgomery-friendly, because they have the very useful property that every modulus p fulfill that $p \equiv \pm 1 \mod r^m$ for a positive integer $m < n$. This property implies that $\mu = -p^{-1} \mod r^m \equiv \mp 1 \mod r^m$. In this way, if $m = 1$ we can save the multiplication by μ in the Step 3 of Algorithm 5 and the REDC algorithm can be computed using only n^2 word multiplications. In Chapter 5 and Chapter 9 we present how this kind of primes could improve the performance of a cryptographic scheme.

3.3.4.2. Barrett modular reduction

Time after the proposal of Montgomery for modular reduction, Paul Barrett presented in [15] an algorithm which does not require a change of representation. His proposal is based on the following observation, let p be an n-word prime and let be $0 \leq c' < p^2$ a $2n$-word integer. The modular reduction $c = c' \mod p$ can be computed as $c = c' - q \cdot p$, where q represents the quotient $q = \left\lfloor \frac{c'}{p} \right\rfloor$.

In order to efficiently compute this reduction method, it is necessary to pre-compute the constant value $\mu = \left\lfloor \frac{r^{2n}}{p} \right\rfloor < r^{n+1}$. Which will be used to approximate the quotient q through the following equation

$$\hat{q} = \left\lfloor \frac{\lfloor c'/r^{n-1} \rfloor \cdot \mu}{r^{n+1}} \right\rfloor. \tag{3.6}$$

This approximation of q is very close and it is possible to show that $q - 2 \leq \hat{q} \leq q$. Barrett algorithm for modular reduction is presented in Algorithm 6, on it and in Equation (3.6) we can observe that the computation of the approximation of the quotient q can be efficiently performed using an integer multiplication μ and fast left/right shifts.

Algorithm 6 Barrett reduction algorithm.

Input: A $2n$-word integer c', the n-word modulo p and $\mu = \lfloor r^{2n}/p \rfloor$.
Output: The n-word integer c such that $c = c' \mod p$.

1: $\hat{q} \leftarrow \lfloor \lfloor c'/r^{n-1} \rfloor \cdot \mu / r^{n+1} \rfloor$
2: $s \leftarrow c' \mod r^{n+1}$
3: $t \leftarrow (\hat{q} \cdot p) \mod r^{n+1}$
4: $c \leftarrow s - t$
5: **if** $c < 0$ **then**
6: $\quad c \leftarrow c + r^{n+1}$
7: **end if**
8: **while** $c \geq p$ **do**
9: $\quad c \leftarrow c - p$
10: **end while**
11: **return** c

A straightforward optimization from the Algorithm 6 can be achieved, observing that the multiplication by μ in Step 1 must be divided by r^{n+1}, therefore, only the most significant half of the product is needed. In the same way, for the multiplication by p we can observe that the product must be reduced modulo r^{n+1}, which means that only the least significant half of that product is needed. Applying this optimizations, the Barrett reduction can be computed at a cost of roughly two half multiplications of n-word by $(n+1)$-word integers. In Chapter 4.2 we will show how this half multiplications could be efficiently computed.

Folding technique

Although the Barrett reduction algorithm is a little bit more expensive than Montgomery reduction, in our work we explored the use of this algorithm applying a further optimization. The idea was proposed in 2007 by Hasenplaugh *et al.* in [84] and it is known as folding technique. This optimization allows to reduce the number of multiplications needed in the Barrett reduction, at a cost of additional pre-computation.

Given an n-word modulo p and a $2n$-word integer c', the folding technique consist in pre-computing the value $\mu' = r^{3n/2} \mod p$ that is used to compute the value $\bar{c} \equiv c' \mod p$ through the following equation

$$\bar{c} = (c' \mod r^{3n/2}) + \left\lfloor \frac{c'}{r^{3n/2}} \right\rfloor \cdot \mu'.$$

In this way, we obtain a resultant value $\bar{c} < r^{(3n/2)+1}$, at the cost of multiplying an $n/2$-word by an n-word integer. After that, in order to reduce $\bar{c}$ completely, the classical Barrett algorithm is used. This technique can be used multiple times, however, according to the author

with only two folding steps we already achieve the best results. According to implementation, that will be presented in Chapter 5, we confirm that the best performance of Barrett reduction algorithm can be obtained with two folding steps, for the cases studied in this thesis.

3.4. RNS arithmetic

Due to its parallel-friendly nature, during the last few decades many researchers have adopted the Residue Number System (RNS), which is particularly useful for performing fast arithmetic over large integers. Because it distributes the overall arithmetic computation over several small moduli, whose size in bits is frequently chosen to match the size of the registers in the target platform.

The Residue Number System relies on the ancient Chinese Remainder Theorem presented in Theorem 3.1. So, in order to use the Residue Number System arithmetic it is necessary to define an RNS-basis $\mathcal{B} = \{m_1, m_2, \ldots, m_l\}$, which is a set of l pairwise relative prime moduli. The number l of moduli to be used is such that the product $M = \prod_{i=1}^{i=l} m_i$ must comply that $p < M$, where p is the biggest number to operate. In the case of this thesis p corresponds to a large prime number. In this way, an n-word number $a \in \mathbb{Z}_M$ can be uniquely represented by the l-tuple $A = (a_1, a_2, \ldots, a_l)$, where each a_i is computed as the residue of a modulo m_i and $l = n$. In the remainder of this section, for simplicity, the operation $a \mod m$ will be written as $a = |a|_m$.

Theorem 3.1 (Chinese Remainder Theorem (CRT)). *For an integer $l \geq 2$, let $m_1, m_2, \ldots, m_l$ be non-zero integers that are pairwise relative prime, i.e. $\gcd(m_i, m_j) = 1$ for $i \neq j$. Then, for any integers $a_1, a_2, \ldots, a_l$ the system of congruences*

$$x \equiv a_1 \mod m_1, \qquad x \equiv a_2 \mod m_2, \qquad \ldots, \qquad x \equiv a_l \mod m_l,$$

has a solution, and this solution is uniquely determined modulo $m_1, m_2, \ldots, m_l$.

In the same way that for Montgomery modular arithmetic, when the RNS representation is used, a transformation to and from the RNS representation is needed at the start and the end of the computations. Let $A = (a_1, a_2, \ldots, a_l)$ be the RNS representation of $a \in \mathbb{Z}_M$, we can obtain its integer representation using the following recovery formula based on Theorem 3.1,

$$a = \left| \sum_{i=1}^{l} \left| a_i \cdot M_i^{-1} \right|_{m_i} \cdot M_i \right|_M, \quad \text{where } M_i \triangleq M/m_i. \tag{3.7}$$

A method to avoid the reduction modulo M in the evaluation of the right hand side of Equation (3.7), is rewriting the value a as follow

$$a = \sum_{i=1}^{l} \gamma_i \cdot M_i - \alpha \cdot M, \quad \text{with } \gamma_i \triangleq \left| a_i \cdot M_i^{-1} \right|_{m_i}. \tag{3.8}$$

Where α is a positive integer, and by construction $0 \leq a/M < 1$. From Equation (3.8), we can compute α as

$$\alpha = \left\lfloor \sum_{i=1}^{l} \frac{\gamma_i}{m_i} \right\rfloor \tag{3.9}$$

and since $\gamma_i < m_i$, we have that $0 \leq \alpha < l$.

3.4.1. Addition, subtraction and multiplication

Let a and b be two n-word integers with $a, b < M$, represented as the RNS tuples $A = (a_1, a_2, \ldots, a_l)$ and $B = (b_1, b_2, \ldots, b_l)$. The RNS addition denoted by $\oplus$, subtraction $\ominus$ and the RNS multiplication $\otimes$ can be performed component wise as,

$$\begin{aligned} C = A \oplus B &= (|a_1 + b_1|_{m_1}, \ldots, |a_l + b_l|_{m_l}), \\ C = A \ominus B &= (|a_1 - b_1|_{m_1}, \ldots, |a_l - b_l|_{m_l}), \\ D = A \otimes B &= (|a_1 \cdot b_1|_{m_1}, \ldots, |a_l \cdot b_l|_{m_l}). \end{aligned} \tag{3.10}$$

We can observe that, the addition, subtraction and multiplication of elements in $\mathbb{Z}_M$ can be performed using smaller computations modulo m_i, which are independent and can be carried out in parallel. Therefore, if the target platform is equipped with l processing units, then the computational cost of computing any RNS arithmetic operation in Equation (3.10) is approximately the same that the cost of a single operation modulo m_i.

3.4.1.1. Selection of moduli m_i

For the sake of efficiency, the moduli m_i are usually selected as

$$m_i = 2^w - \mu_i,$$

where the μ_i values are chosen as small as possible and as before w represent the size in bits of the processor register. If $\mu_i < 2^{\lfloor w/2 \rfloor}$, then the reduction modulo m_i found in the computation of $D = (d_1, d_2, \ldots, d_l)$ in Equation (3.10) can be efficiently performed by repeating at most twice the operation

$$d_i = t_i \bmod 2^w + \mu_i \cdot \lfloor t_i / 2^w \rfloor,$$

where $t_i = a_i \cdot b_i$. Thereafter, it is guaranteed that $d_i \in [0, 2^w[$. Since $2^w > m_i$, one may need to compute a final reduction, at a cost of at most one subtraction operation. In order to assure a constant-time implementation, this reduction is carried out by executing two unconditional reductions, followed by one conditionally subtraction.

3.4.2. Modular reduction

Modular multiplication generally is performed computing an integer multiplication followed by a modular reduction modulo p, instead of modulo M as shown Equation (3.7). This last step, can be performed using the modular reduction approach proposed in [18, 117] and adapted to GPU platforms by Jeljeli in [97] (see also [96]).

Let $D = A \otimes B$ be the RNS representation of the integer multiplication of $d = a \cdot b$ with $0 \leq d < M$ and $p < M$. The strategy proposed in [97] consist in to perform the modular reduction $d \bmod p$ directly applying the RNS recovery formula of Equation (3.8) as follows

$$z = \sum_{i=1}^{\ell} \gamma_i \cdot |M_i|_p - |\alpha \cdot M|_p, \text{ where } \gamma_i \triangleq \left| d_i \cdot M_i^{-1}, \right|_{m_i}. \tag{3.11}$$

Remark 3.1. *Let a, b be n-word elements of $\mathbb{Z}_M$, the result of the integer multiplication $d = a \cdot b$ can be uniquely recovered from its RNS representation if and only if $d < M$. In general, given that the integer product $0 \leq d < M^2$, it follows that the RNS representation of a, b and d requires an RNS-basis composed of l w-bit moduli, with $l = 2n$.*

We can get a good approximation of α, that at the same time can be efficiently computed, by using the fact that $m_i \approx 2^w$. Hence, the ratio γ_i / m_i could be approximated considering

only the σ most significant bits of the quotient $\gamma_i/2^w$ as follows,

$$\hat{\alpha} \triangleq \left\lfloor \sum_{i=1}^{l} \frac{\left\lfloor \frac{\gamma_i}{2^{w-\sigma}} \right\rfloor}{2^{\sigma}} + \Delta \right\rfloor, \tag{3.12}$$

where σ is an integer in the range $[1, w]$ and $0 < \Delta < 1$ is an error correcting parameter [97, 108].

Remark 3.2. *The integer part of the sum in Equation* (3.12) *can be efficiently computed by considering the output carries produced by the addition of the σ most significant bits of the γ_i values. Notice that the integer sum produces an integer in the range $[0, l[$.*

This approach for modular reduction is shown in Algorithm 7. From Equation (3.11) we can notice that the algorithm does not computes $d \bmod p$, but it produces an multiple of it which is bounded by $2^w \cdot l \cdot p$. In practice this implies that the RNS vector Z must be represented using at least two more moduli in the RNS-basis. Consequently, we increased the size of the RNS-basis $\mathcal{B}$ from l to $l+3$ moduli. By taking this caution measure, one guarantees that accumulating thousands of modular multiplications (required in the computation of a typical RSA modular exponentiation), will not exceed the RNS upper bound M.

Algorithm 7 RNS Modular Reduction [97].

Input: An RNS vector D, an l-moduli RNS-basis $\mathcal{B}$, a prime p, and the parameters w, σ, and Δ.

Output: An RNS vector Z, such that it corresponds to the integer representation of $z \equiv d \bmod p$.

Precomputation:

1: RNS vector $\left|M_i^{-1}\right|_{m_i}$ for $i \in \{1, \ldots, l\}$
2: Table of RNS vectors $|M_i|_p$ for $i \in \{1, \ldots, l\}$
3: Table of RNS vectors $|\alpha \cdot M|_p$ for $\alpha \in \{1, \ldots, l-1\}$

Computation:

4: **for each** processing unit i **do**
5: $\quad \gamma_i \longleftarrow \left| d_i \cdot \left|M_i^{-1}\right|_{m_i} \right|_{m_i}$ ▷ 1 RNS product
6: **end for**
7: $\alpha \longleftarrow \left\lfloor \sum_{j=1}^{l} \frac{\left\lfloor \frac{\gamma_j}{2^{w-\sigma}} \right\rfloor}{2^{\sigma}} + \Delta \right\rfloor$ ▷ Addition of l σ-bit terms
8: **for each** processing unit i **do**
9: $\quad z_i \longleftarrow \left| \sum_{j=1}^{l} \gamma_j \cdot \left| |M_j|_p \right|_{m_i} \right|_{m_i}$ ▷ l RNS products and $(l-1)$ RNS additions
10: $\quad z_i \longleftarrow \left| z_i - \left| |\alpha \cdot M|_p \right|_{m_i} \right|_{m_i}$ ▷ 1 RNS subtraction
11: **end for**
12: **return** $Z = (z_1, \ldots, z_l)$

3.4.2.1. Montgomery modular reduction

Another strategy to compute a modular reduction by a n-word prime number p, consist in use the adaptation of Montgomery reduction presented in Equation (3.5) to RNS arithmetic proposed by Posch [142] and analyzed in [108]. This adaptation requires to handle two distinct RNS-basis $\mathcal{B} = \{m_1, m_2, \ldots m_l\}$ and $\mathcal{B}' = \{m'_1, m'_2, \ldots m'_l\}$ such that $\gcd(M, M') =$

$\gcd(M, p) = 1$, where $l = n$, $M = \prod_{i=1}^{l} m_i$ and $M' = \prod_{i=1}^{l} m'_i$. In addition, the constants used in the Montgomery reduction Equation (3.5) must be redefined in order to use the RNS arithmetic, thus $R = M$ is used instead of $R = r^n$ and μ now is a vector that corresponds to the RNS representation of $-p^{-1} \mod R$ in base $\mathcal{B}$.

In the same manner that for the Montgomery reduction presented in Equation (3.5), in the RNS version of Montgomery is also possible to use the method introduced by Walter in [163] to avoid the required conditional subtraction. In this case, a redundant representation of the elements in Montgomery's representation is achieved by choosing a Montgomery radix such that $4p < R$ and a RNS-basis $\mathcal{B}'$ such that $2p < M'$ as is noted in [107, 108]. And also, at the end of the whole computations the result can be normalized at the cost of a single subtraction.

The procedure to compute the Montgomery modular reduction in RNS is presented in Algorithm 8. We can observe from it, that the multiplication $D_{\mathcal{B}}$ by μ is carried out in base $\mathcal{B}$ in Step 5, therefore, the modulo R is automatically applied to the computation and the result is equivalent to compute $\mu \cdot d \mod R$ of Equation (3.5). After that, in Step 11 the value equivalent to $d + (\mu \cdot d \mod R) \cdot p$ from Equation (3.5) is computed. This operation is performed in base $\mathcal{B}'$ because its result is always a multiple of R and thus is always 0 in base $\mathcal{B}$. Finally, in Step 12 the division by R is computed, which corresponds to the RNS representation of $d \cdot R^{-1} \mod p$ in base $\mathcal{B}'$. This computation is performed in base $\mathcal{B}'$, since the value M^{-1} does not exist in base $\mathcal{B}$. Through the algorithm it is necessary to perform two *base extensions*, which consist in transforming a number in either base $\mathcal{B}$ or base $\mathcal{B}'$ into a number in base $\mathcal{B} \cup \mathcal{B}'$. The first base extension (Steps 6 to 10) is made to derive an approximation δ from the value of γ in Step 5, that allows to compute the value $(d + (\mu \cdot d \mod R) \cdot p)/R$ in base $\mathcal{B}'$. The second base extension (Steps 13 to 17) is performed at the end of the algorithm, in order to obtain the RNS representation of the result computed in Step 12 in base $\mathcal{B}$. As can be seen, modular arithmetic based on RNS Montgomery modular reduction requires that all the operations should be performed in both RNS-basis $\mathcal{B}$ and $\mathcal{B}'$ to maintain compatibility with the reduction algorithm.

3.5. Finite field arithmetic

From the fact that the ring $\mathbb{Z}/p\mathbb{Z}$ of residue classes of integers modulo a prime number p is a field (Theorem §2.10), which is denoted as $\mathbb{F}_p$. We can represent the residue class $[a]_p \in \mathbb{Z}/p\mathbb{Z}$ by the unique integer a in the interval $[0, p-1]$, that is in the residue class of $[a]_p$. From now on, we will say that such integer a belongs to $\mathbb{F}_p$. Although depending on our purposes, some times we will also use an incompletely reduced integer to represent the residue class $[a]_p$, which is not uniquely determined since it belongs to an interval of length greater than p.

In the following, we detail how the operations over elements in a finite field can be fast and securely implemented.

3.5.1. Addition and subtraction

The operations in finite fields of addition and subtraction can be computed combining Algorithms 1 and 2 for integer addition and subtraction. For example, given two n-word elements $a, b \in \mathbb{F}_p$, the addition in the finite field is computed using Algorithm 1 to obtain the value $c = a + b < 2p$, at the end if the resultant value is greater than p we use the Algorithm 2 to compute $c - p$ in order to obtain a result in the interval $[0, p-1]$. On the other hand, the finite field subtraction of a and b is computed using Algorithm 2 to obtain $c = a - b$, which can be less than 0. In this case we use Algorithm 1 to compute $c + p$ in order to obtain the subtraction in the interval $[0, p-1]$.

Algorithm 8 RNS Montgomery Modular Reduction [108].

Input: The RNS vectors $D_{\mathcal{B}}$ and $D_{\mathcal{B}'}$, the l-moduli RNS-basis $\mathcal{B}$ and $\mathcal{B}'$, a prime p.
Output: The RNS vectors $Z_{\mathcal{B}}$ and $Z_{\mathcal{B}'}$ that corresponds to the integer representation of $z \equiv d \mod p$.

Precomputation:

1: RNS vectors $\left|M_i^{-1}\right|_{m_i}$, $\left|M_i^{'-1}\right|_{m_i'}$, $\left|M^{-1}\right|_{m_i'}$ and $|p|_{m_i'}$ for $i \in \{1, \ldots, l\}$
2: Matrices of vectors $|M_i|_{m_j'}$ and $|M_i'|_{m_j}$ for $i, j \in \{1, \ldots, l\}$
3: Tables of RNS vectors $|\alpha \cdot (-M)|_{m_i'}$ and $|\alpha \cdot (-M')|_{m_i}$ for $\alpha, i \in \{1, \ldots, l\}$

Computation:

4: **for each** processing unit i **do**
5: $\gamma_i \longleftarrow \left|D_{\mathcal{B}i} \cdot |\mu_i|_{m_i}\right|_{m_i}$ ▷ 1 RNS product
6: $\theta_i \longleftarrow \left|\gamma_i \cdot \left|M_i^{-1}\right|_{m_i}\right|_{m_i}$ ▷ 1 RNS product
7: **end for**
8: $\alpha \longleftarrow \left\lfloor \sum_{j=1}^{l} \frac{\left\lfloor \frac{\theta_j}{2^{w-\sigma}} \right\rfloor}{2^{\sigma}} \right\rfloor$ ▷ Addition of l σ-bit terms
9: **for each** processing unit i **do**
10: $\delta_i \longleftarrow \left|\sum_{j=1}^{l} \left||M_i|_{m_j'} \cdot \theta_j\right|_{m_i'} + |\alpha(-M)|_{m_i'}\right|_{m_i'}$ ▷ l RNS products and l RNS additions
11: $\gamma_i \longleftarrow \left|D_{\mathcal{B}'i} + \left(\delta_i \cdot |p|_{m_i'}\right)\right|_{m_i'}$ ▷ 1 RNS product and 1 RNS addition
12: $Z_{\mathcal{B}'i} \longleftarrow \left|\gamma_i \cdot \left|M^{-1}\right|_{m_i'}\right|_{m_i'}$ ▷ 1 RNS product and 1 RNS addition
13: $\theta_i \longleftarrow \left|Z_{\mathcal{B}'i} \cdot \left|M_i^{'-1}\right|_{m_i'}\right|_{m_i'}$ ▷ 1 RNS product
14: **end for**
15: $\alpha \longleftarrow \left\lfloor \sum_{j=1}^{l} \frac{\left\lfloor \frac{\theta_j}{2^{w-\sigma}} \right\rfloor}{2^{\sigma}} + 0.5 \right\rfloor$ ▷ Addition of l σ-bit terms
16: **for each** processing unit i **do**
17: $Z_{\mathcal{B}i} \longleftarrow \left|\sum_{j=1}^{l} \left||M_i'|_{m_j} \cdot \theta_j\right|_{m_i} + |\alpha(-M')|_{m_i}\right|_{m_i}$ ▷ l RNS products and l RNS additions
18: **end for**
19: **return** $Z_{\mathcal{B}}$ and $Z_{\mathcal{B}'}$

As can be seen in the above described procedures, it results necessary to compute a conditional addition or subtraction in order to obtain the field addition or field subtraction, respectively. However, given that these conditional operations could allow some side channel attacks, it is desirable to implement them in constant time in order to prevent such attacks. In Algorithm 9 and 10 we shown how these field operations can be securely performed.

Algorithm 9 Finite field addition

Input: The elements $a, b \in \mathbb{F}_p$ and the n-word prime number p.
Output: The element $c \in \mathbb{F}_p$ such that $c = a + b$.

1: $c' \leftarrow \text{IntegerAddition}(a, b)$ ▷ Algorithm 1
2: $c' \leftarrow \text{IntegerSubtraction}(c', p)$ ▷ Algorithm 2
3: **for** $i = 0$ **to** $n - 1$ **do**
4: $\quad d \leftarrow p_i \ \& \ c'_n$
5: **end for**
6: $c \leftarrow \text{IntegerAddition}(c', d)$ ▷ Algorithm 1
7: **return** c

Algorithm 10 Finite field subtraction

Input: The elements $a, b \in \mathbb{F}_p$ and the n-word prime number p.
Output: The element $c \in \mathbb{F}_p$ such that $c = a - b$.

1: $c' \leftarrow \text{IntegerSubtraction}(a, b)$ ▷ Algorithm 2
2: **for** $i = 0$ **to** $n - 1$ **do**
3: $\quad d \leftarrow p_i \ \& \ c'_n$
4: **end for**
5: $c \leftarrow \text{IntegerAddition}(c', d)$ ▷ Algorithm 1
6: **return** c

The field addition presented in Algorithm 9 can be implemented at a cost of $2(n + 1)$ word additions, $(n + 1)$ word subtractions and n logical `AND`s instructions. While, the implementation of the field subtraction in Algorithm 10 can have a cost of $(n + 1)$ word additions, $(n + 1)$ word subtractions and n logical `AND`s instructions.On the other hand, the non-constant time version of these operations can be implemented at a cost of $(n + 1)$ word additions and $(n + 1)$ word subtractions each one.

It is worth mentioning that, Montgomery based arithmetic is also compatible with these algorithms for finite field addition and subtraction. However, given that the elements $\tilde{a}$ and $\tilde{b}$, that correspond to the Montgomery representation of $a, b \in \mathbb{F}_p$, are upper-bounded by $2p$ when it is desirable to avoid the conditional subtraction in the REDC algorithm. Then, it is necessary to subtract or add the value $2p$ instead of p, in order to obtain the results of the field addition or subtraction in the interval $[0, 2p - 1]$.

3.5.2. Multiplication and squaring

There exist two different methods to compute the field multiplication, by performing the multiplication and reduction in a interleaved way [42, Algorithm, 11.1 and 11.3], or performing the integer multiplication and the modular reduction in two separated steps. In this thesis, we used the last method because it allows us to employ the best algorithms for both integer multiplication/squaring and modular reduction (explained in previous sections) in order to get a better performance.

The field multiplication and squaring can be straightforward implemented. This because, we can use the strategies described in §3.3.2 and §3.3.3 to compute the integer multiplication

and squaring, respectively. Followed by the Barrett modular reduction or the Montgomery modular reduction, as can be seen in Algorithm 11 and Algorithm 12.

Algorithm 11 Finite field multiplication

Input: The elements $a, b \in \mathbb{F}_p$ and the n-word prime number p.
Output: The element $c \in \mathbb{F}_p$ such that $c = a \cdot b$.

1: $c' \leftarrow \text{IntegerMultiplication}(a, b)$
2: $c \leftarrow \text{ModularReduction}(c', p)$ ▷ Algorithm 6 or Algorithm 5
3: **return** c

Algorithm 12 Finite field multiplication

Input: The element $a \in \mathbb{F}_p$ and the n-word prime number p.
Output: The element $c \in \mathbb{F}_p$ such that $c = a^2$.

1: $c' \leftarrow \text{IntegerSquaring}(a)$
2: $c \leftarrow \text{ModularReduction}(c', p)$ ▷ Algorithm 6 or Algorithm 5
3: **return** c

3.5.3. Exponentiation

There are differents methods to compute the modular exponentiation $c = a^e \mod p$, which depend on the nature of the base $a \in \mathbb{F}_p$ and the integer exponent e. When these two values vary from one computation to another, an intuitive method to perform the exponentiation is to scan the bits of the exponent e either from left to right or from right to left. Where, at each step we perform a squaring and, depending on the scanned bit value, we perform a subsequent multiplication. This binary method requires $k-1$ squarings and $\frac{1}{2}(k-1)$ multiplications in average, for a k-bit exponent e with its most significant bit equal to one. However, there are methods that tend to decrease the number of multiplications, leading to a considerable speedup in the exponentiation computation at a cost of some pre-computations. These methods consist of slicing the binary representation of e into pieces using a window of length ω and to process the windows one by one. For instance, in the 2^ω-method [42] it is necessary to pre-compute the values a^i for i in the interval $[0, 2^\omega - 1]$, thus the cost of this method is $k-\omega$ squarings, $\left(\frac{k}{\omega} - 1\right)\left(1 - \frac{1}{2^\omega}\right)$ multiplications[1] and $2^\omega - 2$ multiplications from pre-computations. A further improvement to the 2^ω-method is the strategy known as the sliding windows method [32], that allows nonzero windows of variable length with the aim of increase the occurrence of zero windows, which implies to decrease even more the number of multiplications. Besides, given that the nonzero windows are formed in such way that correspond to odd integers, the pre-computed values needed for exponentiation are reduced a half.

Nevertheless, for the methods described above it can be possible to retrieve each bit (or some useful information) of the exponent from the exponentiation computation. This has serious consequences when the exponent is some secret key. A well known method for efficiently performing this exponentiation is by considering a variant of the fixed-window exponentiation method, which first produces a regular recoding of the exponent. This recoding is performed using the unsigned recoding proposed by Joye and Tunstall in [101] and shown in Algorithm 13. Given a k-bit exponent the recoding provides an encoding of length $n = \left\lceil \frac{k}{\omega} \right\rceil + 1$ whose digits belong to the set $\{1, 2, \ldots, 2^\omega\}$, where ω is the window size used for the recoding.

[1] The factor $\left(1 - \frac{1}{2^\omega}\right)$ corresponds to the probability that the processed windows be different than zero.

Algorithm 13 Unsigned exponent regular recoding [101]

Input: A k-bit exponent e, window size ω.
Output: $f = (f_{n-1}, \ldots, f_0)$ with $f_i \in \{1, 2, \ldots, 2^\omega\}$ for $0 \leq i < n$.

1: $i \leftarrow 0 \quad j \leftarrow 1$
2: **while** $e \geq 2^\omega + 1$ **do**
3: $\quad d \leftarrow e \mod 2^\omega$
4: $\quad d' \leftarrow d + j + 2^\omega - 2$
5: $\quad f_i \leftarrow (d' \mod 2^\omega) + 1$
6: $\quad j \leftarrow \lfloor d'/2^\omega \rfloor$
7: $\quad e \leftarrow \lfloor e/2^\omega \rfloor$
8: $\quad i \leftarrow i + 1$
9: **end while**
10: $f_i \leftarrow e + j - 1$
11: **return** f

Algorithm 13 allows us to perform a regular modular exponentiation, which can prevent timing side-channel attacks. However, given that the windowed exponentiation algorithms require to access to a pre-computation table, it is necessary to implement a mechanism that protects this access. Since, a naive implementation of the access to the pre-computation table could produce a RSA signature vulnerable to cache side-channel attacks. In order to prevent this kind of attacks, whenever the pre-computed table is accessed we perform a *linear pass* memory access as a protective counter-measure [139]. This technique consists of traversing the entire pre-computing table every time that a certain position is accessed. In this way, we prevent that accesses to memory have a different running time, due to the cache miss or cache hits. The protected modular exponentiation that computes $y = x^e \mod p$ is shown in Algorithm 14. This algorithm in average has a cost of $\lfloor \frac{k}{\omega} \rfloor$ modular multiplications and $k-1$ modular squarings.

Algorithm 14 Protected fixed-window modular exponentiation

Input: The k-bit integers x, e and p, and the window size ω.
Output: A k-bit integer y such that $y = x^e \mod p$.

Precomputation:
1: Recode e using Algorithm 13 to obtain the encode f of length $n = \lceil \frac{k}{\omega} \rceil + 1$.
2: Compute $\Gamma[i] \leftarrow x^i \mod p$ for $i \in \{0, \ldots, 2^\omega\}$
Computation:
3: $y \leftarrow$ Perform a *linear pass* to recover $\Gamma[f_{n-1}]$
4: **for** $i = n - 2$ **down to** 0 **do**
5: $\quad y \leftarrow y^{2^\omega}$
6: $\quad z \leftarrow$ Perform a *linear pass* to recover $\Gamma[f_i]$
7: $\quad y \leftarrow y \cdot z$
8: **end for**
9: **return** y

Chapter 4

Protected implementation of RSA signature algorithm

Proposed by Ron Rivest, Adi Shamir, and Len Adleman in 1978 [143], RSA has become the most deployed public key cryptosystem in practical applications. An intensively used security application of RSA is the signing and verification of digital certificates. However, RSA is a relatively slow algorithm and therefore it must be carefully implemented to become competitive in terms of timing performance and memory usage. On the top of that, the computation of RSA main primitives, quite especially its modular exponentiation, must be run in constant-time. This feature presents a first line of defense against side-channel attacks [113].

In this section we focused on the efficient and secure implementation of the RSA signature algorithm using CPU and GPU platforms. First, we present the signature algorithm and the size of the keys used to implement it at different security levels. Finally, we compare two approaches to implement the modular arithmetic; using the Montgomery based arithmetic described in §3.3 and the RNS based arithmetic described in §3.4.

4.1. RSA signature scheme

The most used public key cryptosystem is RSA [143], which is generally used for digital signing of documents in day-to-day Internet applications. This intensive usage is mainly due to a large number of Internet certificates verified with RSA public keys, moreover, most certificate authorities only issue RSA certificates. Currently, RSA key exchange is used in most popular communication protocols like the Transport Layer Security (TLS) [56]. The RSA cryptosystem consists of three main algorithms.

Key generation algorithm: given a security parameter it produces the public and private keys by constructing a $2k$-bit modulo $N = p \cdot q$, where p, q are two k-bit prime numbers. The RSA public key is the tuple composed by the modulus N and the public exponent e, which is generally chosen as $e = 2^{16} + 1$. The RSA private exponent is defined as, $d = e^{-1} \mod \phi(N)$, where $\phi(\cdot)$ stands for the Euler's totient function.

Signature algorithm: given the RSA private key (d, N) and a message m, the Full Domain Hash (FDH) signature s of m is computed as $s = H(m)^d \bmod N$, where $H(\cdot)$ represents a hash function that maps m to $\mathbb{Z}_N$. It has been shown that the FDH RSA signature is provably secure [45]. A standard trick to compute the signature s is to use the Chinese Remainder Theorem, which allows us to compute a $2k$-bit RSA exponentiation using

two independent k-bit modular exponentiations that can be computed concurrently. This standard trick is shown in Algorithm 15.

Verification algorithm: knowing the public key (e, N) and a signature s of some message m, this algorithm consists in verifying that $H(m)$ and $s^e \mod N$ are equal.

Algorithm 15 RSA signature using CRT

Input: A private key $\{N = p \cdot q, d\}$, $q_{inv} = q^{-1} \mod p$ and $h = H(m)$.
Output: The signature s of the message m.

1: $s_1 = h^{d \mod (p-1)} \mod p$
2: $s_2 = h^{d \mod (q-1)} \mod q$
3: $t = q_{inv} \cdot (s_1 - s_2) \mod p$
4: $s = s_2 + t \cdot q$
5: **return** s

Through this section, we use two different approaches to compute the modular arithmetic required for the implementation of the RSA signature algorithm. These approaches are the Montgomery's representation based arithmetic presented in §3.3.4.1 and the arithmetic based on the Residue Number System described in §3.4. In both methods, we use a $2k$-bit RSA modulo N which is computed as $N = p \cdot q$, where p and q are distinct k-bit prime numbers. Nevertheless, given that RSA exponentiation modulo N can be computed using two independent k-bit exponentiations modulo p and q, respectively. We focus on the computation of $a = b^e \mod p$, where the numbers a, b and p we assume that have a bit-length of k bits.

4.1.1. Security

Although breaking RSA is not known to be equivalent to factoring the RSA modulus, it is common to approximate the RSA security according to the complexity of the best known algorithm for integer factorization. From the latest results of state-of-the-art integer factorization algorithms, it is believed that RSA-1024 offers (much) less than 80 bits of security [5]. Therefore, the National Institute of Standards and Technology (NIST) recommended in [11] the usage of the following RSA modulus sizes.

RSA modulus size in bits	Bits of security	Term
1024	80	until 2010
2048	112	until 2030
3072	128	from 2030

Table 4.1: Security levels for RSA cryptosystem.

Hence, in order to perform an urgently needed migration to higher levels of security, it would be required to achieve highly-optimized implementations of the RSA cryptosystem and its associated building blocks, so that the key exchange operation as well as the signing and signature verification of documents can be executed at a speed that is able to cope with Internet's high-volume data exchange.

4.2. Efficient implementation on CPU platforms

In this section we describe how the RSA signature algorithm can be efficiently implemented in CPU platforms. Particularly, we focus in the relatively newest Haswell and Skylake

Intel micro-architectures. For this purpose, we present how Montgomery based arithmetic can be implemented, taking advantage of the instructions specially developed for arithmetic over large integers. Besides, we present a comparison between the RNS arithmetic based on the reduction Algorithm 7 and the reduction Algorithm 8, and we also shown how this RNS arithmetic can be efficiently implemented benefiting from the set of instructions `AVX2`.

4.2.1. Montgomery based arithmetic

With the aim of efficiently compute the integer multiplication, squaring and the operations required by the Montgomery's reduction, we take advantage of the instruction `MULX` and the set of instructions `ADX` described in §3.2 following the techniques explained in previous sections.

4.2.1.1. Integer multiplication

Given that our target architectures permit to use up to 15 general purpose registers for the computations, it is possible to perform up to one 8-word multiplication maintaining all the partial results in registers. This is because, during the computation we need three registers for the operands and the output, one register for the implicit operand of `MULX`, two accumulators, and nine registers for storing the partial result of the multiplication of the multiplicand operand by a word of the multiplier. This fact, allows us to perform an Schoolbook integer multiplication that outperforms the Karatsuba ones when the operands have a word-size $0 < n \leq 8$, as shown Table 4.2.

n	`MULX`		**Clock cycles**			
	Karatsuba	Schoolbook	Karatsuba		Schoolbook	
			HW	SK	HW	SK
2	3	4	20	14	12	8
3	9	9	28	28	20	16
4	12	16	60	43	32	24
6	27	36	112	87	84	48
8	48	64	196	137	184	87
12	121	-	400	278	-	-
16	209	-	692	419	-	-
24	376	-	1328	960	-	-

Table 4.2: Comparison of timings for integer multiplication using Karatsuba and Schoolbook method. The timings are reported in number of word multiplications (using `MULX` instructions) and clock cycles measured on a Haswell(HW) and Skylake(SK) micro-architectures.

On the other hand, an n-word multiplication when $n > 8$ is computed using the Karatsuba multiplication method [105]. This method was recursively used to compute n-word multiplications for $n \in \{16, 24\}$. For the 16-word multiplication we applied one Karatsuba level (for going from 16- to 8-word multiplications). Analogously, for an 24-word multiplication we utilized two Karatsuba levels (for going from 24 to 12, then to 6-word multiplications). The results obtained from the Karatsuba approach are also presented in Table 4.2.

4.2.1.2. Comparison with other approach

In 2015, Michael Scott in [151] realized the study and implementation of a Karatsuba variant proposed by Weimerskirch and Paar [166], which is used for multiplication of arbi-

trary degree polynomials, that means, polynomials without a power of 2 degree. Scott used such method known as Arbitrary degree Karatsuba (ADK) in the *reduced-radix* scenario, where a number is represented using a word size lower than the one belonging to the target processor. This method has the advantage that the partial products can be accumulated without worrying about the carries.

We implemented this strategy using a word size of $r = 2^{62}$ bits, which was proposed by the author. A comparison of the results of our implementation of this strategy against the results obtained using our proposed combination of Karatsuba and Schoolbook method is reported in Table 4.3. In it we can observe that our proposed method achieves better performance than the reported by Scott for the n-word multiplications used in this work.

n-word multiplication	Clock cycles			
	Scott [151]		Our method	
	HW	SK	HW	SK
2×2	36	38	12	8
3×3	56	58	20	16
4×4	76	77	32	24
6×6	128	119	84	48
8×8	228	189	184	87
12×12	376	312	400	278
16×16	600	503	692	419
24×24	1224	1006	1328	960

Table 4.3: Comparison of timings for integer multiplication using Scott strategy [151] against Karatsuba-Schoolbook method. The timings are reported in clock cycles measured on a Haswell (HW) and Skylake (SK) micro-architectures.

4.2.1.3. Integer squaring

We used the Schoolbook method with the operand-scanning strategy described in §3.3.3 for n-word multiplications when $n < 6$. While, for the squaring computation of operands with a word-size $n \geq 6$ we used a variant of the Karatsuba method that takes advantage of the repeated products.

The implementation of an n-word squaring for $n = 24$ was conducted using three Karatsuba levels (for going from 24- to 12-, then to 6-word and finally 3-word multiplications/squarings), which requires to compute eighteen squaring and nine multiplications of 3-words operands; the squaring of operands with word-size $n = 12$ was performed using two Karatsuba levels (for going from 16- to 8- and then to 4-word multiplications/squarings), with a cost of six squarings and three multiplications of 4-word operands; and for the 8-word squaring we occupied one Karatsuba level (for going from 8- to 4-word multiplications/squarings) using two squarings and one multiplication of 4-word operands.

According to our experiments, we observed that for squaring computation of up to 4-word operands a better performance is obtained using the Schoolbook method, and for operands with a word-size $n \geq 6$ the best approach is to use the Karatsuba method. This results can be observed in Table 4.4.

4.2.1.4. Montgomery modular reduction

The computation of the modular reduction presented in Equation (3.5) requires to compute two n-word multiplications, which are divided or reduced modulo $R = r^n$. A straightforward optimization can be applied observing that for the multiplication $\mu \cdot c' \mod R$ is

n	Word muls		Clock cycles			
	Karatsuba	Schoolbook	Karatsuba		Schoolbook	
			HW	SK	HW	SK
2	3	3	8	6	8	6
3	6	6	20	14	20	14
4	10	10	29	26	28	23
6	21	21	60	55	60	51
8	36	-	123	109	-	-
12	57	-	274	194	-	-
16	100	-	511	331	-	-
24	235	-	1024	713	-	-

Table 4.4: Timings of integer squaring using the Schoolbook and the Karatsuba methods. The timings are reported in number of word multiplications (using `MUX` instructions) and clock cycles measured on a Haswell (HW) and Skylake (SK) micro-architectures.

only necessary to compute the least significant half of the result; and for $(\mu \cdot c' \mod R) \cdot p$ only the most significant half of the product is needed because it is divided by R.

In the same manner that for integer multiplication, these operations were performed using the Schoolbook multiplication method for the cases when $n \leq 8$. Thus, an n-word multiplication divided by R is computed using $n(n+1)/2 + n$ word multiplications; and an n-word multiplication modulo R is computed using $n(n+1)/2$ word multiplications. On the other hand, for the cases when $n > 8$ we used up to two levels of the Karatsuba method, however, in each level it is necessary to compute one $n/2$-word multiplication and two half $n/2$-word multiplications as shown in Figure 4.1.

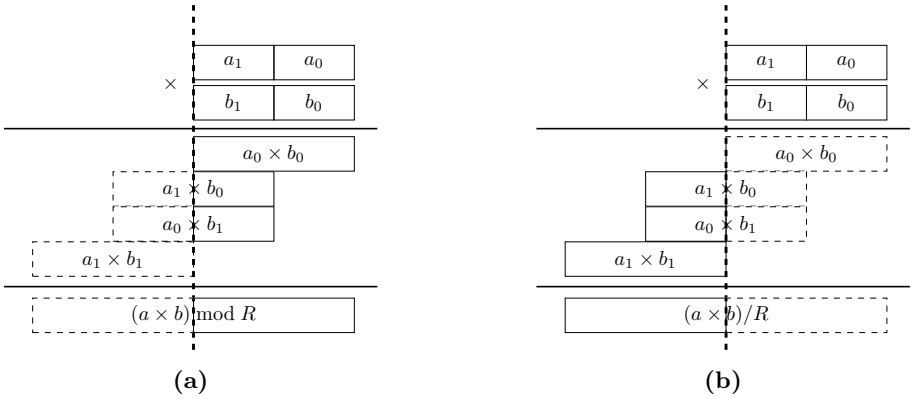


Figure 4.1: Given two n-word integers a and b written as $a = a_0 + a_1 \cdot r^{n/2}$ and $b = b_0 + b_1 \cdot r^{n/2}$, respectively. The figure (a) shown a Karatsuba n-word multiplication modulo R, and figure (b) shown a Karatsuba n-word multiplication divided by R. The dashed rectangles shown the operations that are not computed.

Following the strategies described above, we implemented the Montgomery modular reduction, thus as the modular multiplication and squaring. In Table 4.5 we present the associated cost to perform these operations.

Algorithm	Clock cycles					
	8-word		16-word		24-word	
	HW	SK	HW	SK	HW	SK
Montgomery reduction	232	224	900	728	1864	1582
Modular multiplication	424	349	1628	1233	3132	2688
Modular squaring	420	338	1500	1131	2820	2395

Table 4.5: Timings for modular reduction, modular multiplication and modular squaring. The timings are reported in clock cycles measured on Haswell (HW) and Skylake (SK) micro-architectures.

4.2.1.5. Montgomery based RSA signature

The RSA signature show in Algorithm 15 was performed using Montgomery based arithmetic. We computed the RSA modular exponentiations using the protected exponentiation method shown in Algorithm 14 and also were computed concurrently using the OpenMP library. The 8-word modular exponentiations for RSA-1024 were performed using a window size $\omega = 4$, and for the 16- and 24-word modular exponentiations we employed a window size of $\omega = 5$ for RSA-2048 and RSA-3072, respectively.

In Table 4.6 we report the latency achieved by our library for RSA signatures using 1024-, 2048, and 3072-bit keys. In the table we can see a comparison of our results against related works previously reported in the open literature on CPU platforms. Particularly, we compare our work with the presented by Bos *et al.* in [26] where the Montgomery multiplication is computed by splitting the Montgomery algorithm into two parts, which can be executed in parallel using the SIMD instructions. The authors in the same work, present their obtained results for an RSA signature using a serial implementation. We also compare with the work of Gueron and Krasnov [81] where they reported an RSA implementation that benefits from the redundant integer representation that avoids the carry propagation in the addition computations, using operands composed by digits of 29-bit words each one. In both works, the authors do not mention that its implementation is completely protected against side channel attacks.

Work	Clock cycles (Millions)			P
	RSA-1024	**RSA-2048**	**RSA-3072**	
Bos [26][1] (SSE)	2.09	12.21	-	✗
Bos [26][1]	0.88	4.92	-	✗
Gueron [81][3]	-	2.1	9.6	✗
Gueron [81][4]	-	1.9	6.0	✗
This work[3]	**0.29**	**1.92**	**5.93**	✓
This work[4]	**0.25**	**1.65**	**5.02**	✓

[1]Sandy Bridge, [2] Ivy Bridge, [3] Haswell, [4] Skylake
P = Protected Implementation.

Table 4.6: Performance comparison of RSA signature implemented in CPU platforms using Montgomery based arithmetic.

4.2.2. RNS based arithmetic

In order to perform an efficient implementation of the RNS based arithmetic presented in §3.4, we took advantage of the `AVX2` instruction set introduced in §3.2.1. Due that, multiplication in the `AVX2` instruction set is defined for 32-bit integers we use a word size $w = 32$, and therefore, numbers are represented in radix $r = 2^w$. In this way, the operations used in RSA signature with 1024-, 2048- and 3072-bit keys must be computed using integers with a word-size $n \in \{16, 32, 48\}$. On the other hand, depending of the used RNS modular reduction algorithm (see §3.4), we need to use a different size for the RNS-basis. For instance, when the RNS reduction Algorithm 7 is applied we used an RNS-basis of size $l = 2n+3$, and if the RNS reduction Algorithm 8 is employed we used two RNS-basis of size $l = n$. Given two n-word integers a and b in their RNS representation A and B in base $\mathcal{B} = \{m_1, \ldots, m_l\}$ (or in base $\mathcal{B} = \{m_1, \ldots, m_l\}$ and $\mathcal{B}' = \{m'_1, \ldots, m'_l\}$ if the reduction Algorithm 8 is used). The implementation of the main operations in the RNS based arithmetic, which takes advantage of the `AVX2` instructions described in §3.4, is presented in the following sections.

4.2.2.1. Main operations in RNS representation

Operations of addition, subtraction, and multiplication of two integers a and b in RNS representation are performed in component-wise as shown Equation (3.10). In the remainder of this section we present how these operations can be implemented using the above described `AVX2` instructions. Considering our target micro-architectures, we can compute up to eight operations modulo m_i simultaneously, therefore, all the computations described below must be performed for each vector used to store the RNS-basis $\mathcal{B}$, *i.e.* for the $\lceil \frac{l}{8} \rceil$ vectors.

Multiplication and squaring in RNS are the most complicated operations, because the `AVX2` instruction `mm256_mul_epu32` only computes four 32×32-bit multiplications. Hence, in order to compute the component-wise integer multiplication of the RNS vectors A and B, we use the `mm256_mul_epu32` instruction to calculate the products of the of odd indexes, that are stored in a vector D_0. Then, we employ the instruction `mm256_shuffle_epi32` to reorder the 32-bit values in the A and B vector registers to compute the products of the even indexes, which are stored in vector D_1 as shown in Figure 4.2.

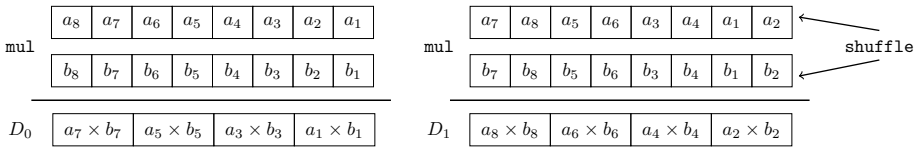


Figure 4.2: Component-wise integer multiplication of two integers a and b in RNS representation.

The modular reduction by each $m_i = 2^w - \mu_i$ in $\mathcal{B}$ is computed as it was described in §3.4.1.1. Given the two vectors D_0 and D_1 computed as in Figure 4.2 and the vector $\mathcal{M}$ composed by the μ_i small values chosen in §3.4.1.1. First, we used the `mm256_shuffle_epi32` instruction to reorder the 32-bit values in the D_0, D_1 and $\mathcal{M}$ vectors. Then, we apply the instruction `mm256_mul_epu32` to obtain the values $\mu_i \cdot \lfloor t_i/2^w \rfloor$ with $t_i = a_i \cdot b_i$, which are stored in the vectors E_0 and E_1. After, we apply the `mm256_srli_epi32` instruction to E_0 and E_1 using an offset of 32 to get the vectors F_0 and F_1, which are added using `mm256_add_epi64` to D_0 and D_1 to obtain the values $d_i = t_i \bmod 2^w + \mu_i \cdot \lfloor t_i/2^w \rfloor$. Finally, after two executions of the above procedure these d_i values stored in D_0 and D_1 correspond to $t_i \bmod m_i$. Therefore, it is necessary to mix the final vectors D_0 and D_1 to get the vector D that stores the values of $A \otimes B$. This procedure is shown in the bottom of Figure 4.3.

On the other hand, addition and subtraction can be straightforward implemented using the vector operations included in the `AVX2` instruction set. Initially, we compute the integer

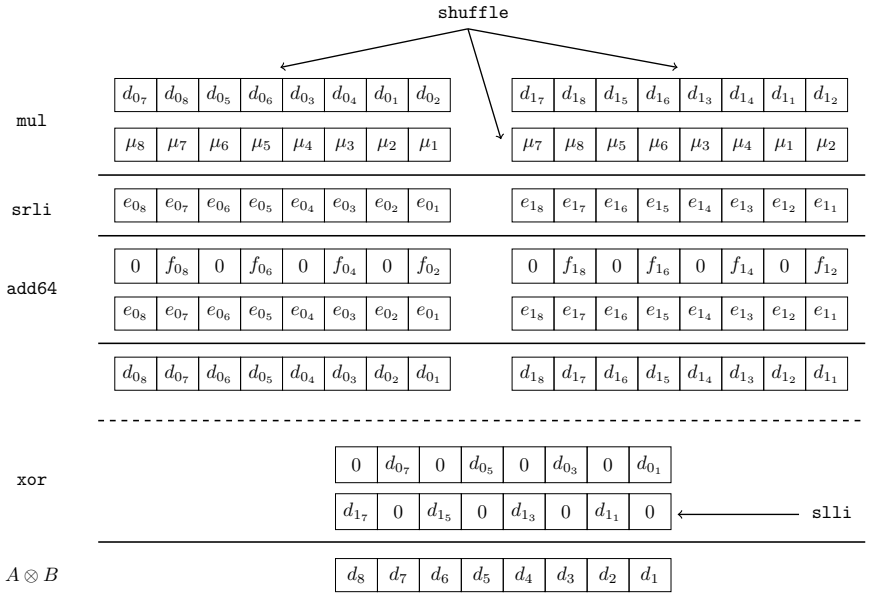


Figure 4.3: RNS multiplication/squaring using AVX2 instructions.

addition or subtraction with the `mm256_add_epi32` or `mm256_sub_epi32` instructions, which is stored in a vector C. Then, the modular reduction by each moduli m_i in the base $\mathcal{B}$ can be computed in constant time as follows: using the `mm256_cmpgt_epi32` instruction we catch the carry or borrow produced by the integer addition or subtraction, that is stored in a vector CB; then, with the instruction `mm256_and_si256` we compute the logic `AND` of CB and the vector $\mathcal{M}$ of moduli m_i, whose reult is stored in a vector D; finally, the vector D is subtracted or added to the value obtained from the above addition or subtraction, respectively. The computation of $C = A \oplus B$ and $C = A \ominus B$ is shown Figure 4.4.

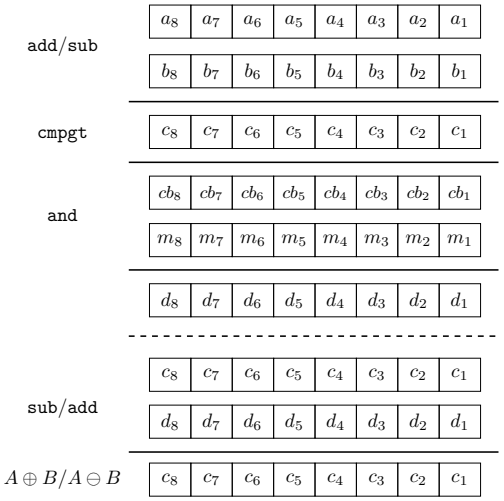


Figure 4.4: RNS addition/subtraction using AVX2 instructions.

4.2.2.2. RNS modular reduction

Modular reduction was performed using Algorithm 7 as was proposed by Jeljeli [97] and Algorithm 8 as was described by Kawamura [108]. In both algorithms it is necessary to compute the approximation $\hat{\alpha}$ presented in Equation (3.12), which was computed following the Remark 3.2. In order to compute $\hat{\alpha}$, for each vector employed to store $\Gamma = (\gamma_1, \ldots, \gamma_l)$ (see Remark 3.2) we compute an `mm256_srli_epi32` instruction with offsets of 5 for RSA-1024 and 25 for RSA-2048 and RSA-3072, when Algorithm 7 is used; and with offsets of 18 for RSA-1024 and RSA2048, and 16 for RSA-3072 when Algorithm 8 is used. After that, we apply to each vector the `mm256_slli_epi32` instruction using offsets that allow to maintain the subsequent additions in the interval $[0, 2^{32}-1]$. For instance, when Algorithm 7 is used the offsets are 19 for RSA-1024, and 17 for RSA-2048 and RSA-3072; and when Algorithm 8 is employed the offsets are 14, 10 and 8 for RSA with 1024-, 2048- and 3072-bit keys respectively. Finally, all vectors are added using `mm256_add_epi32` instructions, and the values of the resultant vector are also added in order to obtain a 32-bit value that is shifted to the right by an offset of 24.

The multiplications of matrix-vector needed in both algorithms can be performed using l RNS multiplications followed by $l-1$ RNS additions, as shown in §3.4. The matrix multiplication in Step 9 of Algorithm 7 can be done in straightforward. However, the matrix multiplications in Steps 10 and 13 in Algorithm 8 require to transpose the matrices $|M_i|_{m'_j}$ and $|M'_i|_{m_j}$. The results of the implementation of both algorithms are presented in Table 4.7. We can observe that the Montgomery reduction in RNS version is two times faster than the Jeljeli's RNS reduction. This is due mainly to the fact that for the RNS Montgomery reduction the basis used to represent the numbers are of size n, while the base used in Jeljeli's RNS reduction has a size of $2n+3$.

Algorithm	**Clock cycles**					
	8-word		16-word		24-word	
	HW	SK	HW	SK	HW	SK
Algorithm 7	3,322	2,943	11,862	10,059	26,046	22,420
Modular mult	3,522	3,039	12,066	10,332	27,450	22,627
Modular sqr	3,402	3,008	12,050	10,270	26.314	22,589
Algorithm 8	1,434	1,330	4,902	4,012	12,042	10,571
Modular mult	1,498	1,385	5,074	4,131	12,350	10,776
Modular sqr	1,494	1,382	5,066	4,123	12,206	10,750

Table 4.7: Comparison of timings for modular reduction, modular multiplication and modular squaring based on Algorithm 7 and Algorithm 8 using the AVX2 instructions. All timings are reported in clock cycles measured on Haswell (HW) and Skylake (SK) micro-architectures.

4.2.2.3. RNS based RSA signature

We perform the RSA signature using two different approaches: an arithmetic based on the reduction proposed by Jeljeli [97]; and an arithmetic based on Montgomery as was introduced by Kawuamura [108]. As before, we computed the RSA modular exponentiations using the protected exponentiation method shown in Algorithm 14 concurrently using the OpenMP library. The modular exponentiations for RSA-1024 were performed using a window size $\omega = 4$, and for modular exponentiations used in RSA-2048 and RSA-3072 we employed a window size of $\omega = 5$. In Table 4.8, we report the latency achieved by our library when

RSA signatures for 1024-, 2048, and 3072-bit keys are implemented. We can observe that RSA signature based on RNS Montgomery arithmetic is two times faster than RSA signature based on Jeljelis RNS reduction. Besides, the best results in Table 4.8 are slower by a factor of 0.2x, 3.1x and 3.9x than our best results reported in Table 4.6 for RSA-1024, RSA-2048 and RSA-3072.

RNS reduction algorithm	**Clock cycles (Millions)**						**P**
	RSA-1024		**RSA-2048**		**RSA-3072**		
	HW	SK	HW	SK	HW	SK	
Jeljeli [97]	2.3	2.0	15.1	12.9	48.7	41.9	✓
Montgomery [108]	0.99	0.90	6.3	5.1	22.5	19.8	✓

P = Protected Implementation.

Table 4.8: Timings for RSA signature algorithm using AVX2 instructions. The timings are reported in millions of clock cycles measured on Haswell (HW) and Skylake (SK) micro-architectures.

4.3. Efficient implementation on GPU platforms

Graphic Processing Units (GPU) are massively parallel processors consisting of hundreds or even thousands of cores. This contrasts with contemporary general purpose CPUs, which can only host at most tens of cores. It is then conceivable that taking advantage of the massively parallel architecture of the GPUs, one can speed up several computations where high computing power is required. A chief example of this, is the efficient implementation of cryptographic applications.

In 2006 NVIDIA introduced a parallel computing framework named `CUDA`, which was especially designed for GPU environments. `CUDA` defines three important features: a threading model, a set of conventions for calling native GPU's functions, and a hierarchical memory infrastructure. In a GPU architecture the basic computational and resource allocation units are *threads*. Threads can be grouped into *blocks*, which in turn can be grouped into a *grid*. Threads in a block are partitioned into warps. For all GPU architectures a warp is composed by 32 threads that run concurrently. A GPU architecture utilizes the Single Instruction Multiple Thread (SIMT) programming model paradigm, where all threads inside a warp can execute the same instruction at the same time. The general programming model consists of code sequences called kernels. A kernel execution can be synchronous or asynchronous. This allows programmers to manage concurrent execution through the completion of command sequences called streams.

With the aim to produce a fast and efficient implementation of the RSA signature algorithm, in this section we make extensive use of the following assembly instructions: `addc` that adds two 32/64-bits values taking into account an input carry bit and producing an output carry bit; `subc` that performs a 32/64-bits subtraction operation with input borrow and producing an output borrow bit; `mul.lo` that multiplies two 32/64-bits values and returns $x_i \times y_i \bmod 2^w$, where x_i and y_i are both non-negative integers, and w is typically selected to be the GPU word size; `mul.hi` that multiplies two 32/64-bits values and returns $x_i \times y_i/2^w$, where x_i and y_i are both non-negative integers; `mad.(hi,lo).cc` which multiply two 32/64-bits values, extracts the higher or lower half of the result, and adds a third 32/64-bit value producing an output carry.

4.3.1. RNS modular Multiplication

In the following, we describe how the RSA signature operations were carried out in the GPU platform. Initially, we consider a pre-computation step performed in the CPU server that host the GPU. In this step, the set of pair-wised relative prime moduli composing the RNS-basis $\mathcal{B}$ are chosen. Then, all the RSA signature operands and pre-computations are converted in their RNS representations. And finally, these values are sent to the GPU platform.

Modular multiplication is performed by launching l blocks with l threads, which compute redundantly an RNS integer multiplications of the form $C = A \otimes B$ where A, B and C correspond to the RNS representation of some integers. This arrangement is depicted in Figure 4.5(a), where it is shown that each thread is in charge of processing a modular product $|a_i \cdot b_i|_{m_i}$. Since each warp executes the same instruction, this arrangement avoids thread's divergence. Besides, all the threads can efficiently access each coordinate of the RNS vectors since these values are allocated on contiguous segments of memory. Finally, each thread stores the output of its modular multiplication computation on a register, thus avoiding the costly access to global memory.

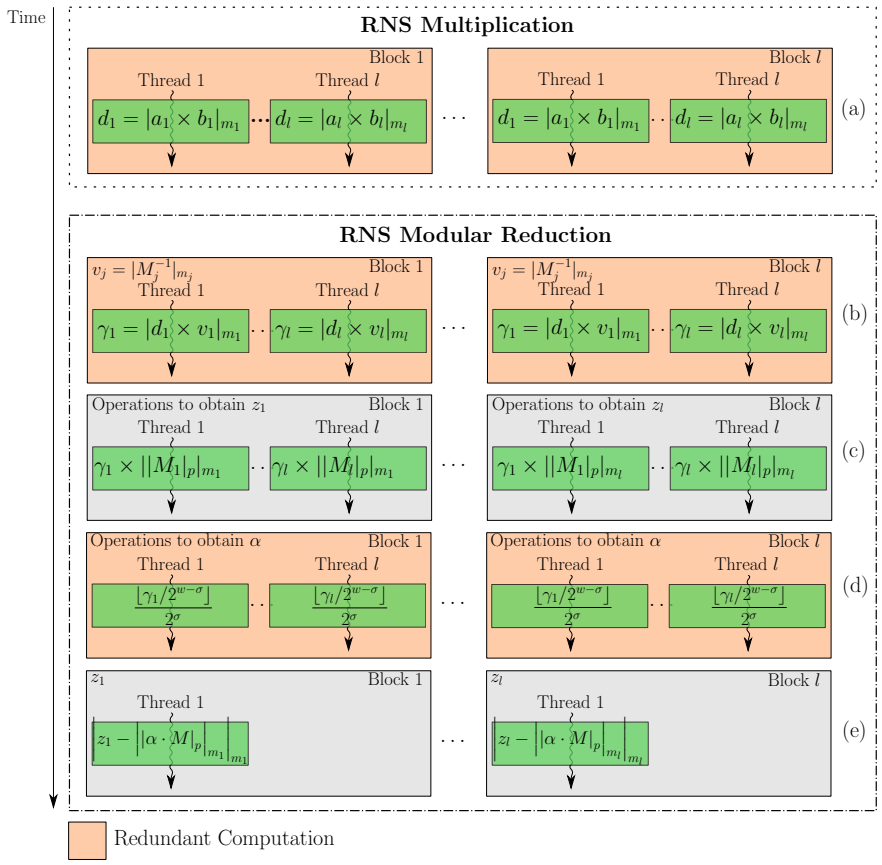


Figure 4.5: Computation of RNS modular multiplication on a GPU platform.

After all threads have completed the integer multiplication step, a modular reduction by the modulus p must be applied. This modular reduction can be performed using the Algorithm 7 proposed by Jeljeli in [97] or using the Algorithm 8 proposed by Kawuamura in [108]. Both algorithms presented in §3.4 are very similar, and we can consider that the Algorithm 8 performs roughly two times the operations used in Algorithm 7. For this reason,

in the next we just explain how the Algorithm 7 can be implemented in GPU platforms.

The modular reduction algorithm requires the pre-computation of several values (Steps 1-3 of Algorithm 7), which are processed in the hosting CPU and sent to the GPU before the main computation starts. The RNS vector $|M_i^{-1}|_{m_i}$ and the RNS table $|M_i|_p$ in Steps 1-2 are both stored in the GPU shared memory so that it can be available for all the threads. The third pre-computed value is the table containing the RNS vectors $|\alpha \cdot M|_p$, for $\alpha = 1, \ldots, l-1$. This table is mapped to the GPU texture memory because only few threads have to query it.

The multiplication operation required in Step 5 is also computed redundantly, with the aim of avoid the broadcasting of the γ_i values to all the threads for the subsequent computations (illustrated in Fig. 4.5b). Then in Step 9, the most expensive task of the reduction algorithm is performed, requiring the computation of l and $l-1$ RNS multiplications and additions, respectively. This calculation is performed in parallel by launching l blocks with l threads each (illustrated in Fig. 4.5c). If there are more than 32 active threads, then an explicit barrier must be placed in order to synchronize all threads of each block, and one must wait until all the threads have completed their execution. This allows a correct addition of partial results. Once that all the partial results have been obtained by each block, each thread stores its result in the shared memory. After all the partial results are obtained, they must be added using a binary addition tree strategy [83]. Step 7 of Algorithm 7 calculates l copies of α using l blocks as shown in Figure 4.5(d). The $l-1$ additions are computed collaboratively as previously mentioned. Finally, in Step 10 of Algorithm 7, a single thread of each one of the l blocks, performs an RNS subtraction saving the final result of the modular reduction into the global memory (see Figure 4.5e). This avoids that threads compete to each other for writing into the same memory address.

In Table 4.9 we present the latency achieved by our software library for the operations of modular reduction and modular multiplication. On it we can observe that the cost of these operations based on the Algorithm 8 is roughly twice more expensive than the cost of the operations based on Algorithm 7.

Algorithm	**Latency (μs)**					
	8-word		16-word		24-word	
	Mont	Jeljeli	Mont	Jeljeli	Mont	Jeljeli
Modular reduction	2.8	0.9	3.4	1.2	3.7	1.3
Modular mult	3.1	0.9	3.5	1.3	3.8	1.3

Table 4.9: Performance comparison of RNS operations implemented in GPU platforms.

4.3.2. RNS based RSA signature

We perform the RSA signature using two different approaches: an arithmetic based on the reduction proposed by Jeljeli [97]; and an arithmetic based on Montgomery as was introduced by Kawuamura [108]. As before, we computed the RSA modular exponentiations using the protected exponentiation method shown in Algorithm 14 concurrently. The modular exponentiations for RSA-1024, RSA-2048 and RSA-3072 were implemented employing a window size of $\omega = 5$. In Table 4.10, we report the latency achieved by our library when RSA signatures for 1024-, 2048, and 3072-bit keys are implemented. We can observe that RSA signature based on Jeljeli's modular arithmetic is two times faster than RSA signature based on Montgomery RNS reduction.

Work	Latency (ms)			P*
	RSA-1024	RSA-2048	RSA-3072	
Jang *et al.* [92]	3.8	13.8	-	✗
Fadhil *et al.* [59]	2.8	17.2	50.1	✗
Yang *et al.* [167]	2.6	6.5	-	✗
Dong *et al.* [58]	-	10.8	26.6	✗
This work Algorithm 7	**1.0**	**2.1**	**3.4**	✓
This work Algorithm 8	2.3	5.0	8.2	✓

*Protected implementation.

Table 4.10: Performance comparison of RSA private operation implemented in GPU platforms.

4.4. Conclusions

In this section, we report a parallel implementation of the RSA private operation after a careful examination of the most suited arithmetic algorithms for both, CPU and GPU high-end platforms. In spite of its massive parallelism, we observe that GPU implementations of RSA are slower than their CPU counterparts. However, the usage of the RNS arithmetic for GPU implementation enjoys a sub-quadratic complexity in the cost of the RSA exponentiation with respect to the size of its key. Thus, we believe that for those multiplication applications where extremely large operands are required, such as fully-homomorphic encryption, our RNS arithmetic library could be of interest.

Finally, we observe that the performance evaluation of our library shows that our implementation achieves a competitive latency, which is faster than previous works that implement the RSA private operation on GPU and CPU platforms using key lengths of 1024, 2048 and 3072 bits.

Part II

Pairing-based cryptography

Chapter 5

Introduction to bilinear pairings

Initially used in cryptography with destructive purposes, bilinear pairings over elliptic curves have been a very active area of research in cryptography. Because, pairings enabled the design of many novel cryptographic protocols that had not previously been feasible. For example, the practical achievement of the *identity-based cryptography* proposed by Adi Shamir in [153], the one-round three-party key agreement, and the aggregate signatures. For a survey of pairing-based protocols, and of the problems on which these protocols are based, we refer the reader to [140].

Since the introduction of cryptographic pairings as a constructive cryptographic primitive, the efficient implementation of pairing-based protocols has become an increasingly important research topic. For this reason, this chapter is dedicated to show, first, a brief mathematical background about bilinear pairings over elliptic curves. Then, to show the main algorithms that are used to securely implement the main operations found in most pairing-based protocols.

5.1. Bilinear pairings

A bilinear pairing can be defined as follow: let $\mathbb{G}_1 = (\mathbb{G}_1, +)$, $\mathbb{G}_2 = (\mathbb{G}_2, +)$ and $\mathbb{G}_T = (\mathbb{G}_T, \cdot)$ be cyclic abelian groups of prime order r. A bilinear pairing e is defined as a map:

$$e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T, \tag{5.1}$$

having the following properties:

- **Non-degenerate**. A pairing is non-degenerate, if for all $A \in \mathbb{G}_1$ exist an element $C \in \mathbb{G}_2$, such that $e(A, C) \neq 1_{\mathbb{G}_T}$ with $A \neq 0_{\mathbb{G}_1}$ and $C \neq 0_{\mathbb{G}_2}$.
- **Bilinear**. Given two elements $A, B \in \mathbb{G}_1$ and two elements $C, D \in \mathbb{G}_2$, where A, B, C and D are different than the identity element. we have that

$$\begin{aligned} e(A + B, C) &= e(A, C) \cdot e(B, C) \text{ and} \\ e(A, C + D) &= e(A, C) \cdot e(A, D), \end{aligned}$$

so that,

$$e(A + A, C) = e(A, C + C) = e(A, C) \cdot e(A, C).$$

An immediate property of the bilinearity is that for any two integers m and n, it holds that:

$$e([m]Q, [n]P) = e([m \cdot n]Q, P) = e(Q, [m \cdot n]P) = e(Q, P)^{m \cdot n}.$$

Additionally to these mathematical properties, a pairing e suitable for use in cryptography furthermore must be **easy to compute** and **hard to invert**. Inverting a pairing e means given $z \in \mathbb{G}_T$ to find $P \in \mathbb{G}_1$ and $Q \in \mathbb{G}_2$ such that $e(P,Q) = z$.

5.1.1. Types of pairings

The most efficient cryptographic pairings currently known come from elliptic curves, where the groups $\mathbb{G}_1$ and $\mathbb{G}_2$ are groups on elliptic curves and the group $\mathbb{G}_T$ is the multiplicative group of a finite field. In the work in [71] Galbraith, Patterson and Smart have defined three types of pairings according to the nature of the groups $\mathbb{G}_1$ and $\mathbb{G}_2$.

- Type 1: it is consider a pairing of type 1 if $\mathbb{G}_1 = \mathbb{G}_2$;
- Type 2: when $\mathbb{G}_1 \neq \mathbb{G}_2$ but an efficiently computable isomorphism $\phi : \mathbb{G}_2 \to \mathbb{G}_1$ is known, while none is known in the contrary direction;
- Type 3: when $\mathbb{G}_1 \neq \mathbb{G}_2$ and no efficiently computable isomorphism is known between $\mathbb{G}_1$ and $\mathbb{G}_2$, in either direction.

In most of this thesis we consider pairings of type 3 (except for §8), because they are compatible with several computational assumptions such as the Desicion Diffie-Hellman in $\mathbb{G}_1$ and $\mathbb{G}_2$.

5.1.2. Curves for fast pairing software implementation

Cryptosystems based on pairings require elliptic curves that are secure and enable efficient pairing computation. In order to guarantee an efficient arithmetic on the elliptic curve $E/\mathbb{F}_p$, it is important that the prime order r of the group $E(\mathbb{F}_p)$ be a large factor of $\#E(\mathbb{F}_p)$. A formalization of this idea consist of considering the quantity $\rho = \frac{\log p}{\log r}$. In this way, for a large p the cardinality of $E(\mathbb{F}_p)$ has roughly the same bit size that p, so ρ measures the ratio between the size of $\#E(\mathbb{F}_p)$ and the size of r. Now, if $\#E(\mathbb{F}_p)$ is prime $\rho \approx 1$ and it is consider an ideal case. In general, the value of ρ should be reasonably close to 1.

The value of the embedding degree k is entirely determined by ρ and the choice of the bit sizes for r and p^k, since $\log p^k / \log r = k\rho$. For instance, if it is desirable a security level of 128 bits, it is necessary that r has a size of 256 bits, p^k should have a size between 3000 to 5000 bits, then the embedding degree k should be in the interval 12-20 [67].

Based on the above paragraphs, Freeman, Scott and Teske [67] gave the following definition of pairing-friendly elliptic curves:

Definition 5.1 (Pairing-friendly elliptic curves)**.** *An elliptic curve $E/\mathbb{F}_p$ is pairing-friendly if the following two conditions hold:*

1. *$\#E(\mathbb{F}_p)$ has a prime factor $r \geq \sqrt{p}$,*
2. *the embedding degree of E with respect to r is less than $\log r/8$.*

This type of curves are constructed through the Complex Multiplication method (CM) [75]. In this method the embedding degree is fixed and subsequently the integers p, r and t are calculated. In order to construct pairing friendly ordinary elliptic curves we need that the previous integers satisfy the following conditions:

1. p is prime or a power of a prime,
2. r is prime,
3. t is relative prime to p,

4. r divides $q + 1 - t$,

5. $r \mid p^k - 1$ and $r \nmid p^i - 1$ for $1 \leq i < k$,

6. $4p - t^2 = Df^2$, for some sufficiently small positive integer D and some integer f.

The above conditions allow to define an ordinary elliptic curve E over a finite field $\mathbb{F}_p$ with embedding degree k and $\#E(\mathbb{F}_p) = p + 1 - t$, such that $r|\#E(\mathbb{F}_p)$. Besides, the curve equation can be determined by the value of D in the point 6. The more common cases are [17]

- $D = 1$, defining a curve with equation $E : y^2 = x^3 + ax$,
- $D = 3$, defining a curve with equation $E : y^2 = x^3 + b$.

The elliptic curve families are perametrized by the tuple of functions $(p(z),\ r(z),\ t(z))$ that satisfy the above conditions for an integer z. According to the equation in condition 6, we say that a family of elliptic curves is complete if there exist a polynomial $f(z)$ such that $4p(z) - t(z)^2 = Df(z)^2$, otherwise it is called a disperse family [67].

Some examples of complete families of pairing-friendly elliptic curves are: BN (Barreto-Naehrig) [13], BW (Brezing-Weng) [29], KSS (Kachisa-Schaefer-Scott) [103] and BLS (Barreto-Lynn-Scott) [12], which are consider for the efficient implementation of bilinear pairings. Particularly, in this thesis we use the family of curves proposed by Barreto-Naehrig.

5.1.2.1. Barreto-Naehrig elliptic curve family

The Barreto-Naehrig (BN) family of elliptic curves was consider ideal from the implementation point of view. However, after the attacks by Kim and Barbulescu [109] BN curves are not anymore ideal (see §5.1.3). These curves have an embedding degree $k = 12$ and ρ-value 1, this facts made them perfectly suited for a security level of 128 bits. Besides, this family of curves facilitate the generation and adjustment of curve parameters, in order to obtain an optimal performance.

This family define elliptic curves with prime order and the characteristic p of the finite field, the order r of the group and the Frobenius trace are parametrized by the following equations:

$$p(z) = 36z^4 + 36z^3 + 24z^2 + 6z + 1, \tag{5.2}$$
$$r(z) = 36z^4 + 36z^3 + 18z^2 + 6z + 1, \tag{5.3}$$
$$t(z) = 6z^2 + 1. \tag{5.4}$$

On this curves the equation $4p - t^2 = Df^2$ holds for $f(z) = 6z^2 + 4z + 1$ and $D = 3$, therefore, given an integer z the Equation (5.2) and Equation (5.3) produce prime numbers. The form of the curve equation is

$$E/\mathbb{F}_p : y^2 = x^3 + b, \tag{5.5}$$

moreover, this elliptic curve is isomorphic to the twist curve of degree $d = 6$ defined as

$$E'/\mathbb{F}_{p^2} : Y^2 = X^3 + b/\xi \tag{5.6}$$

where the elements $b \in \mathbb{F}_p$ and $\xi \in \mathbb{F}_{p^2}$ are not quadratic residues and are not cubic residues over $\mathbb{F}_p$ and $\mathbb{F}_{p^2}$, respectively.

Pairing groups for BN curves

Considering the BN ordinary elliptic curve $E/\mathbb{F}_p$ with embedding degree k, which defines the group $E(\mathbb{F}_p)$ of order $p+1-t=r$, where r is a prime number. The groups involved in the pairing computation are defined as:

- $\mathbb{G}_1$ is the additive group composed by the r-torsion points in $E(\mathbb{F}_p)$.
- $\mathbb{G}_2$ is the additive group generated by the point Q, that means, $\mathbb{G}_2 = \langle Q \rangle$, where given the element $Q' \in E'(\mathbb{F}_{p^{k/d}})[r]$, such that $\mathbb{G}'_2 = \langle Q' \rangle$ the point Q is defined as $Q = \Psi_6(Q')$ (see §2.4.2.1).
- $\mathbb{G}_T$ is the subgroup of $\mathbb{F}^*_{p^k}$ multiplicatively written, denoted as $\mathbb{F}^{\times}_{p^k}$, composed by the set of the r-th roots of the unity in the group $\mathbb{F}^*_{p^k}$.

The definition of the group $\mathbb{G}_2$ allows us to do part of the pairing computations in the subfield $\mathbb{F}_{p^2}$ rather than in $\mathbb{F}_{p^k}$. It is important to say that this definition is possible by the following theorem, that allows to speed up the pairing computation using twist curves.

Theorem 5.1. *Let E be an ordinary elliptic curve over $\mathbb{F}_p$ admitting a twist of degree d. Assume that r is an integer such that $r \mid \#E(\mathbb{F}_p)$ and let $k > 2$ be the embedding degree. Then there is an unique twist E' such that $r \mid \#E'(\mathbb{F}_{p^m})$, where $m = k/\gcd(k,d)$. Furthermore, if we denote by $\mathbb{G}'_2$ the unique subgroup of order r of $E'(\mathbb{F}_{p^m})$ and by $\Psi : E' \to E$ the twisting isomorphism, the subgroup $\mathbb{G}_2$ is given by $\mathbb{G}_2 = \Psi(\mathbb{G}'_2)$.*

5.1.3. Security of pairings

The DLP is believed to be intractable for certain groups carefully chosen; including the multiplicative group of a finite field (Definition 2.1.5), and the group of points on an elliptic curve defined over a finite field (Definition 2.1.5). The closely related Diffie-Hellman problem (DHP) consist in compute $[ab]P$ given P, $[a]P$ and $[b]P$. It is generally assumed that the DLP reduces in polynomial time to the DHP.

The security of many pairing-based protocols is dependent on the in tractability of the following problem:

Definition 5.2 (Bilinear Diffie-Hellman problem (BDHP))**.** *Let e be a bilinear pairing on $(\mathbb{G}, \mathbb{G}_T)$. The bilinear Diffie-Hellman problem (BDHP) is the following: Given the points P, $[a]P$, $[b]P$, $[c]P \in \mathbb{G}$, compute $e(P,P)^{abc} \in \mathbb{G}_T$.*

Hardness of the BDHP implies the hardness of the DHP in both $\mathbb{G}$ and $\mathbb{G}_T$. First, if the DHP in $\mathbb{G}$ can be efficiently solved, then one could solve an instance of the BDHP by computing $[ab]P$ and then $e([ab]P,[c]P) = e(P,P)^{abc}$. Also, if the DHP in $\mathbb{G}_\mathbb{T}$ can be efficiently solved, then the BDHP instance could be solved by computing $g = e(P,P)$, $g^{ab} = e([a]P,[b]P)$, $g^c = e(P,[c]P)$ and then g^{abc}.

The best generic algorithm known for solving the ECDLP is the Pollard's rho method [141] which has an expected running time of $O(\sqrt{r})$, where as before r is the cardinality of the group $E(\mathbb{F}_p)[r]$. Therefore, in order to offer a security level of 128 bits, we must chose a group with order r with a size of 256 bits. On the other hand, recent improvements on the computation of the DLP in a finite field of composite extension degree, reduce the asymptotic complexity of the NFS algorithm. Particularly, in the work of Kim and Barbulescu presented at CRYPTO'16 [109] introduced the extended tower-NFS technique. In this technique if the field characteristic p also has an special form, which is the case for pairings over BN curves, then the asymptotic complexity is $O(\exp(1.56 \cdot (\log p^k)^{1/3} \cdot (\log\log p^k)^{2/3}))$.

However, the works presented in §6 and §7 were done before the work of Kim and Barbulescu, when the asymptotic complexity of the NFS algorithm was $O(\exp(1.92 \cdot (\log p^k)^{1/3} \cdot$

$(\log\log p^k)^{2/3})$). The only work in this chapter that is immune to this kind of attacks is the presented in §8. In Table 5.1 we present the bit size of the field $\mathbb{F}_{p^k}$ that is necessary to resist the extended tower-NFS technique.

$\log p$	$\log p^k$	**NFS Before Kim-Barbulescu**	**Ext. Tower-NFS**
256	3072	$\approx 2^{139-\delta_1}$	$\approx 2^{110-\delta_2}$
384	4608	$\approx 2^{164-\delta_1}$	$\approx 2^{130-\delta_2}$
448	5376	$\approx 2^{175-\delta_1}$	$\approx 2^{139-\delta_2}$
512	6144	$\approx 2^{185-\delta_1}$	$\approx 2^{147-\delta_2}$

Table 5.1: Security levels for pairings over BN curves.

The numbers in Table 5.1 should be read as follows: a 256-bit finite field $\mathbb{F}_p$ that produces an extension field $\mathbb{F}_{p^k}$ of 3072 bits, will provide approximately a security level of $2^{139-\delta_1}$, where δ_1 depends on the curve and on the implementation of the NFS variant. The order of magnitude of this δ value is usually of a dozen. Therefore, if it is desirable to offer a 128-bit security level then a prime p of 448 bits should be used to avoid extended tower-NFS attack (and a 2565-bit prime p when the NFS Before Kim-Barbulescu column is consider).

5.2. Main operations in pairing-based protocols

In this section we introduce the main operations involved in most pairing-based protocols. Besides, we present the algorithms and techniques used to implement them. As was previously mentioned, in most of this thesis we are interested in pairing-based protocols over BN curves, therefore, the operations are addressed for this family of elliptic curves.

BLS short signature scheme

Proposed by Boneh, Lynn, and Shacham in 2001 [24], the BLS signature scheme remains the efficient scheme that achieves the shortest signature length to this day: about 160 bits at the 80-bit security level. Its public parameters are a bilinear pairing $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ between groups of order r, generators G_1, G_2 of $\mathbb{G}_1$ and $\mathbb{G}_2$, and a hash function $\mathcal{H}_1 : \{0,1\}^* \to \mathbb{G}_1$ modeled as a random oracle. The secret key is a randomly selected element $x \in \mathbb{Z}/r\mathbb{Z}$, the public key is the group element $P = [x]G_2$. A signature on a message $m \in \{0,1\}^*$ is obtained as $S = [x]\mathcal{H}(m)$. While, any entity possessing the public key P can verify the signature simply checking whether $e(\mathcal{H}(m), P) = e(S, G_2)$. Boneh, Lynn, and Shacham proved that this scheme is secure under the Computational Diffie–Hellman assumption when $\mathcal{H}$ is modeled as a random oracle.

As can be seen above paragraph, the BLS signature scheme public key and signature are computed through a scalar multiplication on the groups $\mathbb{G}_2$ and $\mathbb{G}_1$, respectively. For the signature is necessary to obtain the image of m under a hash function $\mathcal{H}_1(\cdot)$. Finally, the computation of a pairing is needed for the verification of a given signature. These operations can be found in most pairing-based schemes, although, also a $\mathbb{G}_2$-valued hash function $\mathcal{H}_2$ is needed in many other pairing-based cryptosystems including IBE and HIBE schemes [10, 73, 87], signature and identity-based signature schemes [21, 23, 24, 33, 169], and identity-based signcryption schemes [28, 120]. Such operations are illustrated in Figure 5.1.

5.2.1. Pairing computation

Let p be a prime number, let E be an elliptic curve defined over the finite field $\mathbb{F}_p$ with embedding degree k, and let $\bar{\mathbb{F}}_p$ be the algebraic closure of $\mathbb{F}_p$. We can say that $f(x, y)$

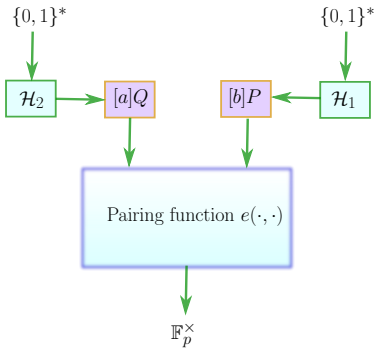


Figure 5.1: Main operations of pairing base protocols.

is a rational function on $E/\mathbb{F}_p$ if there exists a point $P = (x_P, y_P) \in E(\bar{\mathbb{F}}_p)$ such that $f(x_P, y_P) = \mathcal{O}$. The set of rational functions on $E/\mathbb{F}_p$ is denoted as $\bar{\mathbb{F}}_p(E)$ and for all $f \in \bar{\mathbb{F}}_p(E)$ it holds that $f(P) \in \{\bar{\mathbb{F}}_p \cup \mathcal{O}\}$.

Definition 5.3 (Miller function)**.** *Let R be an element in $E(\mathbb{F}_{p^k})$ and let s be a non-negative integer. A Miller function $f_{s,R}$ of length s is a rational function in $\mathbb{F}_{p^k}(E)$ with divisor $(f_{s,R}) = s(R) - ([s]R) - (s-1)(\mathcal{O})$. Let $u_\mathcal{O}$ be an $\mathbb{F}_p$-uniformizing parameter for $\mathcal{O}$. A function $f \in \mathbb{F}_{p^k}(E)$ is said to be normalized if $lc_\mathcal{O}(f) = 1$, where $lc_\mathcal{O}(f) = (u_\mathcal{O}^{-\ell} f)(\mathcal{O})$ and ℓ is the order of f at $\mathcal{O}$.*

Lemma 5.1. *Let $f_{s,R}$ a Miller function, let $\ell_{[a]R,[b]R}$ be the line trough the points $[a]R$ and $[b]R$ in $E(\mathbb{F}_{p^k})$, and let v_R be the vertical line through the point R. For all $a, b \in \mathbb{Z}$ it holds that:*

- $f_{a+b,R} = f_{a,R} \cdot f_{b,R} \cdot \ell_{[a]R,[b]R} / v_{[a+b]R}$,
- $f_{ab,R} = f_{b,R}^a \cdot f_{a,[b]R}$,
- $f_{1,R} = c$ *for some constant* c *(for example* $c = 1$*).*

Let $P, Q \in E[r]$. Victor Miller in [128] described an algorithm for evaluating a normalized Miller function $f_{r,P}$ at the point Q, which repeatedly uses the Lemma 5.1 over the binary representation of r.

Algorithm 16 Miller's Algorithm.

Input: $P, Q \in E[n] \setminus \mathcal{O}$ and $r = (r_{l-1}, \ldots, r_0)_2$.
Output: $f_{r,P}(Q)$.

1: $f \leftarrow 1$, $g \leftarrow 1$, $T \leftarrow P$
2: **for** $i \leftarrow l-2$ **downto** 0 **do**
3: $\quad T \leftarrow [2]T$
4: $\quad f \leftarrow f^2 \cdot \ell_{T,T}(Q)$
5: $\quad g \leftarrow g^2 \cdot v_{[2]T}(Q)$
6: $\quad$ **if** $r_i = 1$ **then**
7: $\quad\quad T \leftarrow T + P$
8: $\quad\quad f \leftarrow f \cdot \ell_{T,P}(Q)$
9: $\quad\quad g \leftarrow g \cdot v_{T+P}(Q)$
10: $\quad$ **end if**
11: **end for**
12: **return** f/g

When Algorithm 16 is used to compute $f_{r,P}(Q)$, one might obtain a value of 0 in the numerator or denominator. This occurs only if Q happens to be a root of one of the line functions ℓ, v encountered during the computation. Since the roots of ℓ and v must lie in $\langle P \rangle$, therefore, the Miller function computation can only fail if $Q \in \langle P \rangle$.

5.2.1.1. Weil pairing

The Weil pairing is usually not used in practice for cryptography. However, it is important to define it because the original construction of the Tate pairing uses the Weil pairing. This construction was the first pairing on elliptic curves and as its name points out was defined by André Weil [165].

Theorem 5.2. *Let E be an elliptic curve defined over a finite field $\mathbb{F}$, let r be an integer relative prime to the characteristic of $\mathbb{F}$, and let P and Q be points of r-torsion on E. The Weil pairing is defined as*

$$e_W(P,Q) = (-1)^r \frac{f_{r,P}(Q)}{f_{r,Q}(P)}. \tag{5.7}$$

e_W is well defined when $P \neq Q$ and $P, Q \neq \mathcal{O}$. Furthermore, $e_W(P,\mathcal{O}) = e_W(\mathcal{O},P) = e_W(P,P)$ for all $P \in E[r]$.

5.2.1.2. Tate pairing

The Tate pairing introduced by John Tate for number fields [158], and applied by Rück and Frey [68] to cryptography, is defined as follows:

Theorem 5.3. *Let E be an elliptic curve, let r be a prime number dividing $\#E(\mathbb{F}_p)$, let $P \in E(\mathbb{F}_{p^k})[r]$ and $Q \in E(\mathbb{F}_{p^k})/rE(\mathbb{F}_{p^k})$, and let R be any point in $E(\mathbb{F}_{p^k})$ such that $\{R, Q+R\} \cap \{P, \mathcal{O}\} = \emptyset$. Then, the Tate pairing defined as*

$$e_T(P,Q) = \left(\frac{f_{r,P}(Q+R)}{f_{r,P}(R)} \right)^{\frac{p^k-1}{r}}$$

is well defined and does not depends on R.

In practice, for the Tate pairing when Q is not a multiple of P we can take $R = \mathcal{O}$, so the Tate pairing can be computed as

$$e_T(P,Q) = f_{r,P}(Q)^{\frac{p^k-1}{r}} \tag{5.8}$$

In this way, we can use Algorithm 16 to compute $f_{r,P}(Q)$ and then we do the final exponentiation by a fast exponentiation algorithm. Besides, in the Miller's algorithm it is possible to eliminate the denominator in the Step 12 according to the work of Barreto *et al.* [14], omitting the calculation of the Steps 5 and 9 in Algorithm 16.

5.2.1.3. Ate pairing

From the Tate pairing definition Hess *et al.* in [85] showed that for all positive integer m such that $r \nmid m$, it holds that

$$e_T(Q,P)^m = f_{r,Q}(P)^{m(p^k-1)/r} \in \mathbb{F}_{p^k}^*$$

is a non-degenerate bilinear pairing. By using Lemma 5.1 and from the fact that $[r]Q = \mathcal{O}$ we have that $f_{r,Q}^m(P) = f_{mr,Q}(P)$. Thus, if λ is a positive integer such that $\lambda \equiv p \mod r$, then $\lambda^k - 1 \equiv p^k - 1 \mod r$. In this way, if we use $mr = \lambda^k - 1$ then

$$f_{mr,Q}(P) = f_{\lambda^k-1,Q}(P) = f_{\lambda^k,Q}(P).$$

Considering that for all $Q \in E'[r]$ it holds that $[\lambda^i]Q = [p^i]Q$ and using again Lemma 5.1, the rational function $f_{\lambda^k,Q}(P)$ can be expressed as follows:

$$f_{\lambda^k,Q}(P) = \prod_{i=0}^{k-1} f_{\lambda,Q}(P)^{\lambda^{k-1-i}p^i} = f_{\lambda,Q}(P)^{\sum_{i=0}^{k-1}\lambda^{k-1-i}p^i}.$$

Now, from the Hasse boundary (Theorem 2.18) we have that $t-1 \equiv p \mod r$, and then we can substitute λ by $t-1$, thus

$$e_T(Q,P)^m = f_{t-1,Q}(P)^{c(p^k-1)/r}$$

where $c = \sum_{i=0}^{k-1}\lambda^{k-1-i}p^i$ and $r \nmid c$.

Theorem 5.4. *Let E be an elliptic curve, let r be a prime number dividing $\#E(\mathbb{F}_p)$ and let $P \in E(\mathbb{F}_{p^k})[r]$ and $Q \in E'(\mathbb{F}_{p^{k/d}})$. Then, the Ate pairing is defined as*

$$e_A(Q,P) = f_{t-1,Q}(P)^{(p^k-1)/r}.$$

By Hasse's boundary the trace of Frobenious t is such that $|t| \leq 2\sqrt{p}$. If t is suitably small with respect to r, then the Ate pairing can be computed using a Millers' loop of shorter size and thus faster than the Tate pairing.

Optimal Ate pairing

The optimal Ate pairing was proposed by Vercauteren in [162], his idea consist in looking for amultiple cr of r so that we can write $cr = \sum c_i p^i$ with c_i small coefficients. Then, one can use a suitable combination of Miller functions $f_{c_i,Q}$ to construct a bilinear pairing that is a power m of the Tate pairing, where $r \nmid m$.

Theorem 5.5. *Let E be an elliptic curve, let r be a prime number dividing $\#E(\mathbb{F}_p)$, let $P \in E(\mathbb{F}_{p^k})[r]$ and $Q \in E'(\mathbb{F}_{p^{k/d}})$, and et $\lambda = \sum_{i=0}^{\varphi(k)-1} c_i p^i$ such that $\lambda = mr$, for some integer m. Then the optimal Ate pairing can be defined as*

$$\hat{e}(Q,P) = \left(\prod_{i=0}^{\varphi(k)-1} f_{c_i,Q}^{p^i}(P) \cdot \prod_{i=0}^{\varphi(k)-1} \frac{\ell_{[s_i+1]Q,[c_i p^i]Q}(P)}{v_{[s_i]Q}(P)} \right)^{(p^k-1)/r} \tag{5.9}$$

with $s_i = \sum_{j=i}^{\varphi(k)-1} c_j p^j$.

The idea for searching the coefficients c_i, such that they are as small as possible, is by computing short vectors for the lattice given in Equation (5.10), by using an available implementation of the LLL algorithm.

$$\begin{pmatrix} r & 0 & 0 & \cdots & 0 \\ -p & 1 & 0 & \cdots & 0 \\ -p^2 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ p^{\varphi(k)-1} & 0 & 0 & \cdots & 1 \end{pmatrix} \tag{5.10}$$

5.2.1.4. Optimal Ate pairing for BN curves

As it was mention before, in this thesis we are interested in the BN family of curves parametriced as shown in §5.1.2.1. For this family of curves Vercauteren in [162] showed that the short vector used to apply the Theorem 5.5 is

$$[6x+2, 1, -1, 1]$$

where 3 out of the four coefficients in the short vector are trivial. Then, the optimal Ate pairing for BN curves is given by the simple formula

$$\hat{e}(Q,P) = \left(f_{z,Q}(P) \cdot \ell_{[z]Q,\Psi_6(Q)}(P) \cdot \ell_{[z]Q+\Psi_6(Q),-\Psi_6^2(Q)}(P)\right)^{\frac{p^{12}-1}{r}} \tag{5.11}$$

with $z = 6x+2$, Ψ_6 the homomorphism on $\mathbb{G}_2$, line functions $\ell_{T,Q}$ passing through the points T, Q, and the groups $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T$ as previously defined. A specialization of Miller's algorithm for computing the optimal Ate pairing can be found in Algorithm 17, which is presented as was proposed by Aranha *et al.* in [6].

Algorithm 17 Optimal Ate pairing [6]

Input: $P \in \mathbb{G}_1$ and $Q \in \mathbb{G}_2$
Output: $f = \hat{e}(Q,P) \in \mathbb{G}_T$.

1: $s \leftarrow 6x+2$ with $s = \sum_{i=0}^{l-1} s_i 2^i$ and $s_i \in \{0,1\}$
2: $f \leftarrow 1$; $T \leftarrow Q$
3: **for** $i = l-2$ **down to** 0 **do**
4: $\quad f \leftarrow f^2 \cdot \ell_{T,T}(P)$; $\quad T \leftarrow [2]T$
5: $\quad$ **if** $s_i = 1$ **then**
6: $\quad\quad f \leftarrow f \cdot \ell_{T,Q}(P)$; $\quad T \leftarrow T+Q$
7: $\quad$ **end if**
8: **end for**
9: $R_0 \leftarrow \Psi_6(Q)$; $\quad R_1 \leftarrow \Psi_6(Q)$
10: **if** $s < 0$ **then**
11: $\quad T \leftarrow -T$; $\quad f \leftarrow f^{p^6}$
12: **end if**
13: $f \leftarrow f \cdot \ell_{T,R_0}(P)$; $\quad T \leftarrow T+R_0$
14: $f \leftarrow f \cdot \ell_{T,-R_1}(P)$; $\quad T \leftarrow T-R_1$
15: $f \leftarrow f^{(p^{12}-1)/r}$
16: **return** f

The final exponentiation shown in Step 15 of Algorithm 17 has a considerable cost, however, this cost can be significantly reduced by factorize the exponent as

$$\frac{p^{12}-1}{r} = (p^6-1)(p^2+1)\left(\frac{p^4-p^2+1}{r}\right).$$

The power $g = f^{(p^6-1)(p^2+1)} \in \mathbb{F}_{p^{12}}$ can be computed using two fast applications of the Frobenius operator, two multiplications and one inversion in $\mathbb{F}_{p^{12}}$. This happens because $p^{12}-1 = (p^6-1)(p^6+1)$, then for an element $f \in \mathbb{F}_{p^{12}}$ whose order does not divides to p^6-1, if $h = f^{p^6-1}$ we have that

$$h^{p^6+1} = f^{(p^6-1)(p^6+1)} = 1$$

and thus, $h^{p^6} = 1/h$. Notice that by choosing a suitable quadratic irreducible polynomial with a primitive root i, the field $\mathbb{F}_{p^{12}}$ can be see as a quadratic extension of $\mathbb{F}_{p^6}$. This field

towering allows us to represent the element h as $h = h_0 + h_1 \cdot i$, where $h_0, h_1 \in \mathbb{F}_{p^6}$. By taking advantage of the Frobenius operator properties, the operation h^{p^6} can be performed by a simple conjugation

$$h^{p^6} = h_0 - h_1 \cdot i = 1/h.$$

The called hard power by the exponent $d = (p^4 - p^2 + 1)/r$ is computed using the Fuentes-Castañeda *et al.* method [69]. The main idea of this method is based on the observation that: a fixed power of a pairing is a pairing. In this way, instead of raising g to the power d we can compute $g^{d'}$, where $d' = md$ and such that $r \nmid m$.

The method consist of representing polynomially the exponent as $d(x) = (p(x)^4 - p(x)^2 + 1)/r(x)$, where $p(x)$ and $r(x)$ are the polynomials parameterizing the BN curves (see §5.1.2.1). Then, the exponent can be written as

$$\begin{aligned} d(x) = & - 36x^3 - 30x^2 + 18x - 2 \\ & + p(x)(-36x^3 - 18x^2 - 12x + 1) \\ & + p(x)^2(6x^2 + 1) \\ & + p(x)^3 \end{aligned}$$

and mapped to $\mathbb{Z}^{16}$ as $d(x) \mapsto [-36, -30, -18, -2, -36, -18, -12, 1, 0, 6, 0, 1, 0, 0, 0, 1]$. After that, the authors define the matrix M shown below from the basis $\{d(x), xd(x), 6x^2d(x), 6x^3d(x)\}$ that allows us to represent all possible products $m(x)d(x)$.

$$M = \begin{bmatrix} -36 & -30 & -18 & -2 & -36 & -18 & -12 & 1 & 0 & 6 & 0 & 1 & 0 & 0 & 0 & 1 \\ 6 & 6 & 4 & 1 & 18 & 12 & 7 & 0 & 6 & 0 & 1 & -1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 & -36 & -30 & -18 & -2 & -36 & -18 & -12 & 2 & 0 & 6 & 0 & 1 \\ 0 & 0 & -1 & 0 & 6 & 6 & 4 & 1 & 18 & 12 & 8 & 0 & 6 & 0 & 1 & -1 \end{bmatrix}$$

Any non-trivial integer linear combination of the rows of M corresponds to an exponent that produces an element g of order r. Then, using the LLL algorithm a linear combination of the short vectors of M are searched [69]. One of these vectors produced by the linear combination of short vectors is

$$[12, 12, 6, 1, 12, 6, 4, 0, 12, 6, 6, 0, 12, 6, 4, -1],$$

which corresponds to the multiple $d'(x) = \lambda_0 + \lambda_1 p + \lambda_2 p^2 + \lambda_3 p^3 = 2x(6x^2 + 3x + 1)d(x)$, where

$$\begin{aligned} \lambda_0(x) &= 12x^3 + 12x^2 + 6x + 1 \\ \lambda_1(x) &= 12x^3 + 6x^2 + 4x \\ \lambda_2(x) &= 12x^3 + 6x^2 + 6x \\ \lambda_3(x) &= 12x^3 + 6x^2 + 4x - 1 \end{aligned}$$

The exponentiation $g^{d'(x)}$ can be computed applying the following strategy. First, the following exponentiations are computed

$$g^x \mapsto g^{2x} \mapsto g^{4x} \mapsto g^{6x} \mapsto g^{6x^2} \mapsto g^{12x^2} \mapsto g^{12x^3},$$

at a cost of three exponentiations by x, three squarings, and one multiplication. Then, the values $a = g^{12x^3} \cdot g^{6x^2} \cdot g^{6x}$ and $b = a \cdot (g^{2x})^{-1}$ are computed using three multiplications. Finally, the result $g^{d'}$ is obtained by computing,

$$(a \cdot g^{6x^2} \cdot g)(b)^p(a)^{p^2}(b \cdot g^{-1})^{p^3},$$

using six extra multiplications.The total cost of compute $g^{d'}$ is of three exponentiations by x, three squarings, 10 multiplications and three Frobenius applications.

5.2.2. Scalar multiplication in $\mathbb{G}_1$ and $\mathbb{G}_2$

In this section we discus the scalar multiplication $[k]P$, for a integer scalar k and a point P in the groups $\mathbb{G}_1$ and $\mathbb{G}_2$ defined over BN elliptic curves. This family of curves allow us to use the Gallant-Lambert-Vanstone (GLV) [72] and the Galbraith-Lin-Scott (GLS)[70] decomposition techniques of dimensions 2 and 4, which permit to speed up scalar multiplication in those pairing groups.

5.2.2.1. ω-NAF scalar multiplication

In the same way as for exponentiation (see §3.5.3), scalar multiplication can be speed up by using a recoded representation of the scalar. Because, it tends to decrease the number of point additions at a cost of some pre-computations. Moreover, given that subtraction of points on an elliptic curve is just as efficient as addition, it is possible to use a signed digit representation of k. A particularly useful signed digit representation is the windowed non-adjacent form ω-NAF, which is defined as follow:

Definition 5.4 (ω-NAF representation [82])**.** *Let ω be a positive integer such that $\omega \geq 2$. The ω-NAF representation of an integer k is an expression $k = \sum_{i=0}^{l-1} k_i 2^i$ where each each non-zero coefficient k_i is odd, $0 \leq k_i < 2^{\omega-1}$, the most significant coefficient in the representation $k_l \neq 0$, and at most one of any ω consecutive digits is non-zero.*

The ω-NAF representation of an integer k can be efficiently computed using Algorithm 18.

Algorithm 18 ω-NAF representation of an integer k.

Input: A positive integer k and the size of window ω.
Output: The representation ω-NAF of k of length l.

1: $i \leftarrow 0$
2: **while** $k \geq 1$ **do**
3: **if** k is odd **then**
4: $k_i \leftarrow k \bmod 2^{\omega}$; $k \leftarrow k - k_i$
5: **else**
6: $k_i \leftarrow 0$
7: **end if**
8: $k \leftarrow k/2$; $i \leftarrow i + 1$
9: **end while**
10: **return** $(k_{l-1}, k_{l-2}, \ldots, k_1, k_0)$

Theorem 5.6 (Properties of the ω-NAF representation)**.** *Let k be a positive integer,*

1. *k has an unique representation ω-NAF,*
2. *The length of the ω-NAF representation of k is at most one more than the length of the binary representation of k.*
3. *The average density of nonzero digits among all ω-NAFs of length l is approximately $1/(\omega + 1)$.*

The ω-NAF scalar multiplication is shown in Algorithm 19, which is a modification of the intuitive binary method.

It follows from the properties (2) and (3) of Theorem 5.6 That the expected running time of Algorithm 19 is approximately

$$\left[1D + (2^{\omega-2} - 1)A\right] + \left[\frac{\ell}{\omega + 1}A + \ell D\right],$$

Algorithm 19 ω-NAF scalar multiplication method.

Input: A positive integer k, a point $P \in E(\mathbb{F}_q)$ with $q = p^n$ for $n \in \mathbb{Z}^+$ and the size of window ω.
Output: $Q = [k]P$.

Precomputation:
1: Compute the ω-NAF representation s of k with length l.
2: Compute $P_i = [i]P$ para $i \in [\pm 1, \pm 3, \pm 5, \ldots, \pm 2^{\omega-1} - 1]$
Computation:
3: $Q \leftarrow \mathcal{O}$
4: **for** $i = l - 1 \rightarrow 0$ **do**
5: $\quad Q \leftarrow [2]Q$
6: $\quad$ **if** $s_i \neq 0$ **then**
7: $\quad\quad Q \leftarrow Q + P_{s_i}$
8: $\quad$ **end if**
9: **end for**
10: **return** Q

where $\ell = \lceil \log_2 k \rceil$, D represents the cost of a point doubling and A denotes the cost of a point addition. The first term of the expression of the running time corresponds to pre-computation cost , while the second term represents the cost of the computations in the Algorithm 19.

5.2.2.2. GLV scalar multiplication

The GLV method introduced by Gallant, Lambert and Vanstone [72] relies on endomorphisms that are specific to the special shape of the curve E. This method, allow us to speed up the scalar multiplication $[k]P$ in the group $\mathbb{G}_1 = E(\mathbb{F}_p)[r]$.

For our case of interest, given the BN elliptic curve $E/\mathbb{F}_p : y^2 = x^3 + b$ parametrized as shown §5.1.2.1 with $p \equiv 1 \mod 3$. We can use the GLV endomorphism $\phi : (x, y) \mapsto (\beta x, y)$ in $\mathbb{G}_1$, where $\beta^3 = 1$ for an element $\beta \in \mathbb{F}_p$ different than 1. In this case the endomorphism ϕ satisfies $\phi^2 + \phi + 1 = 0$ in the endomorphism ring $End(E)$ of E, therefore, ϕ corresponds to the scalar multiplication by $\lambda = -36x^4 + 1$. Thus, $\lambda^2 + \lambda + 1 \equiv 0 \mod r$ means that we have a 2-dimentional decomposition in $\mathbb{G}_1$.

In this way, the GLV method is used to split the ℓ-bit scalar k in two sub-scalars k_1 and k_2, such that $k \equiv (k_1 + \lambda k_2) \mod r$ where the length of sub-scalars is $\ell/2$-bits approximately. So, the scalar multiplication $[k]P$ can be efficiently computed as

$$[k]P = [k_0]P + [k_1]\phi(P)$$

using simultaneous scalar multiplication techniques.

This method requires some pre-computations. However, its efficiency lies in that reduces the number of point doublings in a half, in comparison with the binary method. Besides, the GLV can be combined with the ω-NAF strategy, in order to reduce the number of point additions as can be see it in Algorithm 20.
The cost of Algorithm 20 is

$$\left[2D + (2^{\omega-1} - 2)A\right] + \left[\frac{\ell}{\omega + 1}A + \frac{\ell}{2}D\right],$$

where $\ell = \lceil \log_2 k \rceil$. The first term of the expression corresponds to the pre-computation and the second term denotes the cost of the computation.

Algorithm 20 GLV scalar multiplication method.

Input: A positive integer k, a point $P \in E(\mathbb{F}_q)$ with $q = p^n$ for $n \in \mathbb{Z}^+$, the size of window ω and the endomorphism ϕ over $E(\mathbb{F}_p)[r]$.
Output: $R = [k]P$.

Precomputation:
1: Decompose the scalar k in sub-scalars k_1, k_2 using the GLV method.
2: Compute the ω-NAF representation s, s' of k_1, k_2 with length l.
3: Compute $P_i = [i]P, Q_i = \phi(P)$ para $i \in [\pm 1, \pm 3, \pm 5, \ldots, \pm 2^{\omega-1} - 1]$
Computation:
4: $Q \leftarrow \mathcal{O}$
5: **for** $i = l - 1 \rightarrow 0$ **do**
6: $\quad R \leftarrow [2]R$
7: $\quad$ **if** $s_i \neq 0$ **then**
8: $\quad\quad R \leftarrow R + P_{s_i}$
9: $\quad$ **end if**
10: $\quad$ **if** $s'_i \neq 0$ **then**
11: $\quad\quad R \leftarrow R + Q_{s'_i}$
12: $\quad$ **end if**
13: **end for**
14: **return** R

5.2.2.3. GLS scalar multiplication

The GLS method introduced by Galbraith, Lin and Scott [70] works over extension fields where the p-power Frobenius becomes non-trivial, so it does not rely on a determined shape of the curve E. However, if E has a special shape as in the case in the GLV method, these two strategies can be combined to give higher-dimensional decompositions. This method, allow us to speed up the scalar multiplication $[k]P$ in the group $\mathbb{G}_2 = E'(\mathbb{F}_{p^2})[r]$.

In the BN curves the group $\mathbb{G}'_2$ is always defined over an extension field, then we can combine the GLV endomorphism and the Frobenius map to get the GLS decomposition. The endomorphism in $\mathbb{G}_2$ is defined as $\psi : \Psi \circ \pi_p^i \circ \Psi^{-1}$, where π_p^i represents the i-th application of Frobenius endomorphism and Ψ represents an isomorphism $\Psi : E'(\mathbb{F}_{p^2}) \rightarrow E(\mathbb{F}_{p^{12}})$. The endomorphisms satisfies the degree-four characteristic polynomial $\psi^4 - \psi^2 + 1 = 0$ in the endomorphism ring $End(E)$ of E, therefore, ψ corresponds to the scalar multiplication by $\lambda = p$ and given that $p \equiv t - 1 \mod r$ then $\lambda = t - 1 = 6x^2$.

This method, allows us to split an ℓ-bit scalar k into sub-scalars k_1, k_2, k_3 and k_4, such that $k \equiv (k_1 + \lambda k_2 + \lambda^2 k_3 + \lambda^3 k_4) \mod r$ where the length of sub-scalars is $\ell/4$-bits approximately. So, the scalar multiplication $[k]P$ can be efficiently computed as

$$[k]P = [k_1]P + [k_2]\psi(P) + [k_3]\psi^2(P) + [k_4]\psi^3(P).$$

The efficiency of this method lies in that reduces the number of point doublings in a quarter, in comparison with the binary method. Besides, the GLS can be also combined with the ω-NAF strategy, in order to reduce the number of point additions. This strategy is shown in Algorithm 21.
The cost of Algorithm 21 is

$$\left[4D + 4(2^{\omega-2} - 1)A\right] + \left[\frac{\ell}{\omega + 1}A + \frac{\ell}{4}D\right],$$

where $\ell = \lceil \log_2 k \rceil$. The first term of the expression corresponds to the pre-computation and the second term denotes the cost of the computation.

Algorithm 21 GLS scalar multiplication method.

Input: A positive integer k, a point $P \in E(\mathbb{F}_q)$ with $q = p^n$ for $n \in \mathbb{Z}^+$, the size of window ω and the endomorphism ψ over $\mathbb{G}_2$.
Output: $R = [k]P$.

Precomputation:
1: Decompose the scalar k in sub-scalars k_1, k_2, k_3, k_4 using the GLS method.
2: Compute the ω-NAF representation s_i of k_i with length l for $0 < i \leq 4$.
3: Compute $P_i^j = [i]P_j$ para $i \in [\pm 1, \pm 3, \pm 5, \ldots, \pm 2^{\omega-1} - 1]$ and $0 < j \leq 4$
Computation:
4: $Q \leftarrow \mathcal{O}$
5: **for** $i = l - 1 \rightarrow 0$ **do**
6: $\quad R \leftarrow [2]R$
7: $\quad$ **for** $j = 0 \rightarrow 4$ **do**
8: $\quad\quad$ **if** $s_{j,i} \neq 0$ **then**
9: $\quad\quad\quad R \leftarrow R + P^j_{s_{j,i}}$
10: $\quad\quad$ **end if**
11: $\quad$ **end for**
12: **end for**
13: **return** R

5.2.3. Hashing into elliptic curve groups

This Section discusses the general problem of hashing into elliptic curves groups, particularly in the context of pairing-based cryptography. The goal of this section is to present some algorithms for hashing values to elliptic curve subgroups $\mathbb{G}_1$ and $\mathbb{G}_2$ that can be used in pairing-based cryptography protocols.

The general approach taken in this section to construct secure hash functions to the subgroups $\mathbb{G}_1$ and $\mathbb{G}_2$ of a pairing-friendly elliptic curve E is illustrated in Figure 5.2, and consists of three main steps. The first step takes an arbitrary message m to some element $\mathfrak{h}(m)$ of a set S that is *easy to hash to*, in the sense that the function $\mathfrak{h} : \{0,1\}^* \rightarrow S$ can be easily obtained from a traditional cryptographic hash function like SHA-2 or SHA-3. In our case of interest, the set S is the base field $\mathbb{F}_q$ with q a power of a prime p. The second step maps the resulting value $\mathfrak{h}(m)$ to a point $\tilde{Q} = \mathcal{H}(m) = f(\mathfrak{h}(m))$ in the elliptic curve group $E(\mathbb{F}_q)$ using a map $f : S \rightarrow E(\mathbb{F}_q)$ called an encoding function. Finally, the last step takes the point $\tilde{Q}$ (which can lie anywhere on the curve) and maps it to a point Q' in the group $\mathbb{G}_1$ or $\mathbb{G}_2$.

5.2.3.1. A naive construction

We would like to construct a hash function $\mathcal{H} : \{0,1\}^* \rightarrow \mathbb{G}$ to an elliptic curve group $\mathbb{G}$, which we can assume is cyclic of order r and generated by a given point G. The simplest, most naive way to do so is probably to start from an integer-valued hash function $\mathfrak{h} : \{0,1\}^* \rightarrow \mathbb{Z}/r\mathbb{Z}$ (for which reasonable instantiations are easy to come by) and to define $\mathcal{H}$ as:

$$\mathcal{H}(m) = [\mathfrak{h}(m)]G. \tag{5.12}$$

This is, however, a bad idea on multiple levels.

On the one hand, it is easy to see why this will typically break security proofs in the random oracle model. Indeed, at some point in a random oracle model security reduction, the simulator will typically want to *program* the random oracle by setting some of its outputs to specific values. In this case, it will want to set the value $\mathcal{H}(m)$ for some input m to a certain elliptic curve point P. However, if $\mathcal{H}$ is defined as in Equation (5.12), the simulator

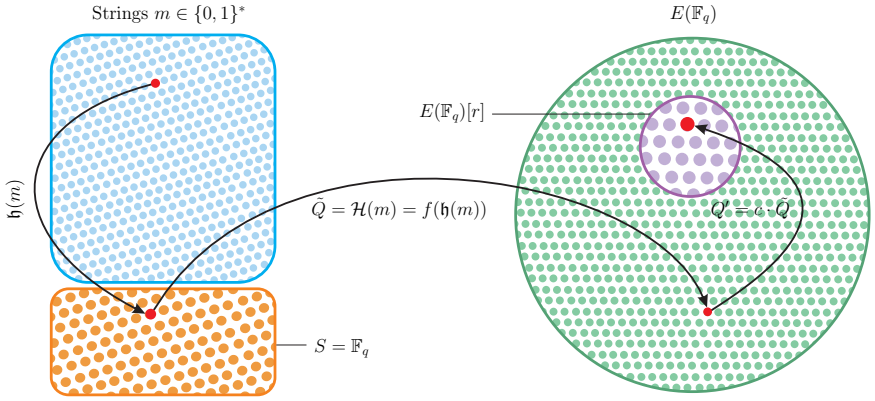


Figure 5.2: Hashing into pairing-friendly elliptic curve subgroups.

should actually program the integer-valued random oracle $\mathfrak{h}$ to satisfy $[\mathfrak{h}(m)]G = P$. In other words, it should set $\mathfrak{h}(m)$ to the discrete logarithm of P with respect to G. But this discrete logarithm is not usually known to the simulator, and it cannot be computed efficiently, therefore, the security reduction breaks down.

On the other hand, it is often not clear how this problem translates into an actual security weakness for a protocol using the hash function $\mathcal{H}$, one could think that it is mostly an artifact of the security proof. Nevertheless, a construction like Equation (5.12) leaks the discrete logarithm of $\mathcal{H}(m)$ whenever m is known, which certainly feels uncomfortable from a security standpoint. We demonstrate below that this discomfort is entirely warranted, by showing that the Boneh-Lynn-Shacham signature scheme [24] presented in §5.2, becomes completely insecure if the hash function involved is instantiated as in Equation (5.12).

Now consider the case when $\mathcal{H}$ is instantiated as in Equation (5.12). Then, the signature on a message m can be written as:

$$S = [x]\mathcal{H}(m) = [x\mathfrak{h}(m)]G = [\mathfrak{h}(m)]P$$

and hence, one can forge a signature on any message using only publicly available data. There is no security left at all when using the trivial hash function construction.

A slightly less naive variant of the trivial construction consists in defining $\mathcal{H}$ as:

$$\mathcal{H}(m) = [\mathfrak{h}(m)]Q$$

where $Q \in \mathbb{G}_2$ is an auxiliary public point distinct from the generator G_2 and whose discrete logarithm α with respect to G_2 is not published. Using this alternate construction for $\mathcal{H}$ thwarts the key-only attack described above against BLS signatures. However, the scheme remains far from secure. Indeed, the signature on a message m can be written as:

$$S = [x\mathfrak{h}(m)]Q = [\alpha x\mathfrak{h}(m)]G = [\mathfrak{h}(m)][\alpha]P.$$

Now suppose an attacker knows a valid signature S_0 on some message m_0. Then the signature S on an arbitrary m is simply

$$S = \left[\frac{\mathfrak{h}(m)}{\mathfrak{h}(m_0)}\right][\mathfrak{h}(m_0)][\alpha]P = \left[\frac{\mathfrak{h}(m)}{\mathfrak{h}(m_0)}\right]S_0,$$

where the division is computed in $\mathbb{Z}/r\mathbb{Z}$. Thus, even with this slightly less naive construction, knowing a single valid signature is enough to produce forgeries on arbitrary messages: again, a complete security breakdown.

5.2.3.2. Hashing by random trial

A classical construction of a hash function to elliptic curves that does work (and one variant of which is suggested by Boneh, Lynn, and Shacham in the original short signatures paper [24]) is the so-called *try-and-increment* algorithm.

Consider an elliptic curve E over a finite field $\mathbb{F}_q$ of odd characteristic, defined by the Weierstrass equation $E : y^2 = x^3 + ax^2 + bx + c$ for some $a, b, c \in \mathbb{F}_q$. A probabilistic way to find a point on $E(\mathbb{F}_q)$ is to pick a random $x \in \mathbb{F}_q$, check whether $t = x^3 + ax^2 + bx + c$ is a square in $\mathbb{F}_q$, and if so, set $y = \pm\sqrt{t}$ and return (x, y). If t is not a square, then x is not the abscissa of a point on the curve: then one can pick another x and try again.

It is an easy consequence of the Hasse bound presented in Theorem 2.18 that the success probability of a single trial is very close to $1/2$. Indeed, if we denote by χ_q the non-trivial quadratic character of $\mathbb{F}_q^*$, extended by 0 to $\mathbb{F}_q$, we have:

$$\#E(\mathbb{F}_q) = 1 + \sum_{x \in \mathbb{F}_q} (1 + \chi_q(x^3 + ax^2 + bx + c)) = q + 1 + \sum_{x \in \mathbb{F}_q} \chi_q(x^3 + ax^2 + bx + c).$$

On the other hand, the success probability w of a single iteration of this point construction algorithm is the proportion of $x \in \mathbb{F}_q$ such that $\chi_q(x^3 + ax^2 + bx + c) = 1$ or 0, namely:

$$w = \frac{\alpha}{2q} + \frac{1}{q} \sum_{x \in \mathbb{F}_q} \frac{1 + \chi_q(x^3 + ax^2 + bx + c)}{2}$$

where $\alpha \in \{0, 1, 2, 3\}$ is the number of roots of the polynomial $x^3 + ax^2 + bx + c$ in $\mathbb{F}_q$. This gives:

$$w = \frac{1}{2} + \frac{\#E(\mathbb{F}_q - q - 1 + \alpha)}{2q} = \frac{1}{2} + O\left(\frac{1}{\sqrt{q}}\right).$$

Now this point construction algorithm can be turned into a hash function based on an $\mathbb{F}_q$-valued random oracle $\mathfrak{h} : \{0,1\}^* \to \mathbb{F}_q$. To hash a message m, the idea is to pick the x-coordinate as, essentially, $\mathfrak{h}(m)$ (which amounts to picking it at random once) and carry out the point construction above. However, since one should also be able to retry in case the first x-coordinate that is tried out is not the abscissa of an actual curve point, we rather let $x \leftarrow \mathfrak{h}(c||m)$, where c is a fixed-length counter initially set to 0 and incremented in case of a failure. Since there is a choice of sign to make when taking the square root of $t = x^3 + ax^2 + bx + c$, we also modify $\mathfrak{h}$ to output an extra bit for that purpose: $\mathfrak{h} : \{0,1\}^* \to \mathbb{F}_q \times \{0,1\}$. This is the try-and-increment algorithm, described more precisely in Algorithm 22 (and called MapToGroup in [24]). The failure probability after up to ℓ iterations is about $2^{-\ell}$ by the previous computations, so choosing the length of the counter c to be large enough for up to $\ell \approx 128$ iterations, say, is enough to ensure that the algorithm succeeds except with negligible probability.

Boneh, Lynn, and Shacham proved that this construction can replace the random oracle $\mathcal{H} : \{0,1\}^* \to E(\mathbb{F}_q)$ in BLS signatures without compromising security. In fact, it is not hard to see that it is indifferentiable from such a random oracle, in the sense of Maurer, Renner, and Holenstein [123]: This ensures that this construction can be plugged into almost all protocols requiring a random oracle $\mathcal{H} : \{0,1\}^* \to E(\mathbb{F}_q)$ while preserving random oracle security proofs.

Nevertheless, there are various reasons why Algorithm 22 is not a completely satisfactory construction for hash functions to elliptic curves. There is arguably a certain lack of mathematical elegance in the underlying idea of picking x-coordinates at random until a correct one is found, especially as the length of the counter, and hence the maximum number of trials, has to be fixed (to prevent collisions). More importantly, this may have adverse consequences for the security of physical devices implementing a protocol using this construction: for example, since the number of iterations in the algorithm depends on the input m, an adversary

Algorithm 22 The try-and-increment algorithm

Input: the message $m \in \{0,1\}^*$ to be hashed.
Output: the resulting point (x, y) on the curve $E/\mathbb{F}_q : y^2 = x^3 + ax^2 + bx + c$.

1: $c \leftarrow 0$ ▷ c is represented as a $\lceil \log_2 \ell \rceil$-bit bit string
2: $(x, b) \leftarrow \mathfrak{h}(c||m)$ ▷ $\mathfrak{h}$ is a random oracle to $\mathbb{F}_q \times \{0,1\}$
3: $t \leftarrow x^3 + ax^2 + bx + c$
4: **if** t is a square in $\mathbb{F}_q$ **then**
5: $\quad y \leftarrow (-1)^b \cdot \sqrt{t}$ ▷ define $\sqrt{\cdot}$ as the smaller square root wrt some ordering
6: $\quad$ **return** (x, y)
7: **else**
8: $\quad c \leftarrow c + 1$
9: $\quad$ **if** $c < \ell$ **then**
10: $\quad\quad$ **Goto** step 2
11: $\quad$ **end if**
12: **end if**
13: **return** $\perp$

can obtain information on m by measuring the running time or the power consumption of a physical implementation.

5.2.3.3. The issue of timing attacks

A concrete situation in which this varying running time can be a serious issue is the case of embedded devices (especially e-passports) implementing an elliptic curve-based Password-Authenticated Key Exchange (PAKE) protocol.

PAKE is a method for two parties sharing a common low-entropy secret (such as a four-digit PIN, or a self-picked alphabetic password) to derive a high-entropy session key for secure communication in an authenticated way. One of the main security requirements is, informally, that an attacker should not be able to gain any information about the password, except through a brute force online dictionary attack (i.e., impersonating one of the parties in the protocol and attempting to authenticate with each password, one password at a time), which can be prevented in practice by latency, smart card blocking, and other operational measures. In particular, a PAKE protocol should be considered broken if a passive adversary can learn any information about the password.

Now consider the PAKE protocol described in Figure 5.3, which is essentially Jablon's Simple Password-base Exponential Key Exchange (SPEKE) [90] implemented over an elliptic curve, except with a random salt as suggested in [91]. The public parameters are an elliptic curve group $\mathbb{G}$ of prime order p and a hash function $\mathcal{H} : \{0,1\}^* \to \mathbb{G}$. The two parties share a common password π, and derive a high-entropy $K \in \mathbb{G}$ using Diffie-Hellman key agreement in $\mathbb{G}$ but with a variable generator $G \in \mathbb{G}$ computed by hashing the password.

But if the hash function $\mathcal{H}$ is instantiated by the try-and-increment construction and an eavesdropper is able to measure the running time of one of the parties, she will find different running times or different power traces depending on how many trials it takes to find a suitable x-coordinate in the computation of $\mathcal{H}(s||\pi)$. Since it takes a single iteration with probability close to $1/2$, an execution of the protocol provides at least one bit of information about π to the adversary (and about $\sum_{k\geq 1} 2^{-k} \log_2(2^{-k}) = 2$ bits on average).

This leads to a so-called *partition attack*, conceptually similar to those described by Boyd *et al.* in [27]: The adversary can count the number of iterations needed to compute $\mathcal{H}(s||\pi_0)$ for each password π_0 in the password dictionary, keeping only the π_0's for which this number of iterations matches the side-channel measurement. This reduces the search space by a factor of at least 2 (and more typically 4) for each execution of the protocol, as the running

Alice (passport)		**Bob (reader)**
	$\xleftarrow{\quad s \quad}$	$s \xleftarrow{\$} \{0,1\}^k$
$G \leftarrow \mathcal{H}(s\|\|\pi)$		$G \leftarrow \mathcal{H}(s\|\|\pi)$
$r_A \xleftarrow{\$} \mathbb{Z}_p$		$r_B \xleftarrow{\$} \mathbb{Z}_p$
$A \leftarrow [r_A]G$	$\xrightarrow{\quad A \quad}$	
	$\xleftarrow{\quad B \quad}$	$B \leftarrow [r_B]G$
$K \leftarrow [r_A]B$		$K \leftarrow [r_B]A$

Figure 5.3: A randomized variant of the SPEKE protocol.

times for different values of s are independent. As a result, the eavesdropper can typically reduce his search space to a single password after at most a few dozen executions of the protocol!

A rather inefficient countermeasure that can be considered is to run all ℓ iterations of the try-and-increment algorithm every time. However, even that is probably insufficient to thwart the attack: indeed, the usual algorithm (using quadratic reciprocity) for testing whether an element of $\mathbb{F}_q$ is a square, as is done in Step 4 of Algorithm 22, also has different running times depending on its input. This can provide information to the adversary as well, unless this part is somehow tweaked to run in constant time[1], which seems difficult to do short of computing the quadratic character with an exponentiation and making the algorithm prohibitively slow with ℓ exponentiations every time. In principle, padding the quadratic reciprocity-based algorithm with dummy operations might provide a less computationally expensive solution, but implementing such a countermeasure securely seems quite daunting. A construction that naturally runs in constant time would certainly be preferable and this is the objective of the §6.1.

[1] By constant time, we mean "whose running time does not depend on the input" (once the choice of parameters like E and $\mathbb{F}_q$ is fixed), and not $O(1)$ time in the sense of complexity theory.

Chapter 6

Constant-time hashing into elliptic curves

In all pairing based protocols, the hash functions are modeled as random oracles [16] in security proofs. However, it is not immediately clear how such a hash function can be instantiated in practice. Indeed, random oracles to groups like $(\mathbb{Z}/p\mathbb{Z})^*$ can be easily constructed from random oracles to fixed-length bit strings, for which conventional cryptographic hash functions usually provide acceptable substitutes. On the other hand, constructing random oracles to a elliptic curves, even from random oracles to bit strings, appears difficult in general, and some of the more obvious instantiations actually break security completely. Therefore in this section is discuss how it can be done correctly, both from a theoretical and a very concrete standpoint.

6.1. Encoding functions to elliptic curves

A natural way to construct a constant-time hash function to an elliptic curve E would be to use, as a building block, a suitable function $f : \mathbb{F}_q \to E(\mathbb{F}_q)$ that can be efficiently computed in constant time. Then, combining f with a hash function $\mathfrak{h} : \{0,1\}^* \to \mathbb{F}_q$, we can hope to obtain a well-behaved hash function to $E(\mathbb{F}_q)$.

Of course, not all such functions f are appropriate: for example, when $q = p$ is prime, the trivial encoding described in §5.2.3.1 is essentially of that form, with $f : u \mapsto [\hat{u}]G$ (and $u \mapsto \hat{u}$ any lifting of $\mathbb{F}_p$ to $\mathbb{Z}$).

On the other hand, if f is a bijection between $\mathbb{F}_q$ and $E(\mathbb{F}_q)$ whose inverse is also efficiently computable, then the following construction:

$$\mathcal{H}(m) = f(\mathfrak{h}(m)) \tag{6.1}$$

is well-behaved, in the sense that if $\mathfrak{h}$ is modeled as a random oracle to $\mathbb{F}_q$, then $\mathcal{H}$ can replace a random oracle to $E(\mathbb{F}_q)$ in any protocol while preserving proofs of security in the random oracle model. Indeed, contrary to what happens in the case of the trivial encoding (where programming the random oracle would require computing discrete logarithm), a simulator can easily choose a value $\mathcal{H}(m_0) = P_0$ by setting $\mathfrak{h}(m_0) = f^{-1}(P_0)$. More generally, such a construction is, again, indifferentiable from a random oracle to $E(\mathbb{F}_q)$.

The same holds if f induces a bijection from $\mathbb{F}_q \setminus T$ to $E(\mathbb{F}_q) \setminus W$ for some finite or negligibly small sets of points T and W.

More generally, we will be considering cases where f is not necessarily an efficiently invertible bijection but only a so-called samplable mapping, in the sense that for each $P \in E(\mathbb{F}_q)$, one can compute a random element of $f^{-1}(P)$ in probabilistic polynomial time.

6.1.1. The Boneh-Franklin encoding

It was actually one of the first papers requiring hashing to elliptic curves, namely Boneh and Franklin's construction [22] of identity-based encryption from the Weil pairing, that introduced the first practical example of a hash function of the form presented in Equation (6.1). Boneh and Franklin used elliptic curves of a very special form:

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

over a field $\mathbb{F}_q$ such that $q \equiv 2 \mod 3$. In $\mathbb{F}_q$, $u \mapsto u^3$ is clearly a bijection, and thus each element has a unique cube root. This makes it possible, following Boneh and Franklin, to define a function f as:

$$\begin{array}{rcl} f : \mathbb{F}_q & \to & E(\mathbb{F}_q) \\ u & \mapsto & \left((u^2 - b)^{1/3}, u\right). \end{array}$$

In other words, instead of picking the x-coordinate and trying to deduce the y-coordinate by taking a square root (which may not exist) as before, we first choose the y-coordinate and deduce the x-coordinate by taking a cube root (which always exists).

Obviously, the function f is a bijection from $\mathbb{F}_q$ to all the finite points of $E(\mathbb{F}_q)$. In particular, this implies that $\#E(\mathbb{F}_q) = 1 + \#\mathbb{F}_q = q + 1$; thus, E is supersingular (and hence comes with an efficient symmetric pairing). This also means that f satisfies the conditions mentioned in the previous section; therefore, construction in Equation (6.1) can replace the random oracle $\mathcal{H}$ required by the Boneh-Franklin IBE scheme, or any other protocol proved secure in the random oracle model. And it can also easily be computed in constant time: It suffices to compute the cube root as an exponentiation to a fixed power α such that $3\alpha \equiv 1 \mod (q-1)$.

Note that in fact, the group $\mathbb{G}$ considered by Boneh and Franklin isn't $E(\mathbb{F}_q)$ itself, but a subgroup $\mathbb{G} \subset E(\mathbb{F}_q)$ of prime order. More precisely, the cardinality q of the base field is chosen of the form $6r - 1$ for some prime $r \neq 2, 3$. Then $E(\mathbb{F}_q)$ has a unique subgroup $\mathbb{G}$ of order r (the curve has cofactor 6), which is the group actually used in the scheme. Hashing to $\mathbb{G}$ rather than $E(\mathbb{F}_q)$ is then easy:

$$\mathcal{H}(m) = f'(\mathfrak{h}(m)) \text{ where } f'(u) = [6]f(u). \tag{6.2}$$

The encoding f' defined in that way isn't injective but it is samplable: indeed, to compute a random preimage of some point $P \in \mathbb{G}$, we can simply compute the six points Q_i such that $[6]Q_i = P$, and return $f^{-1}(Q_i)$ for a random index i. Using that observation, Boneh and Franklin prove that construction in Equation (6.2) can replace the random oracle to $\mathbb{G}$ in their IBE scheme. More generally, it is easy to see that it is indifferentiable from a random oracle in the sense of Maurer *et al.* [123].

6.1.2. Beyond supersingular curves

The previous example suggests that a sensible first step towards constructing well-behaved constant-time hash functions to elliptic curves is to first obtain mappings $f : \mathbb{F}_q \to E(\mathbb{F}_q)$ that are computable in deterministic polynomial time and samplable, and admit constant-time implementations. We will refer to such mappings as *encoding functions* or simply *encodings*.

Note that despite what the name might suggest, there is no assumption of injectivity for those mappings.

It turns out that constructing encodings to elliptic curves beyond special cases such as Equation (6.2) is far from an easy task. In fact, Schoof mentioned the presumably easier problem of constructing a *single* non-identity point on a general elliptic curve over a finite field as open in his 1985 paper on point counting [146], and little progress was made on this problem before the 2000s. Nevertheless, we now know how to construct encodings to essentially all elliptic curves thanks to the work of numerous researchers.

We now present the two most important constructions, due to Shallue and van de Woestijne on the one hand, and Icart on the other.

6.1.3. The Shallue-van de Woestijne approach

In a paper presented at ANTS in 2006, Shallue and van de Woestijne [152] proposed a general construction of an encoding function that applies to all elliptic curves over finite fields of odd characterstic.

Consider the general Weierstrass equation for an elliptic curve in odd characteristic (possibly including 3):

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

Let further $g(x) = x^3 + ax^2 + bx + c \in \mathbb{F}_q[x]$. It is possible to construct an encoding function to $E(\mathbb{F}_q)$ from a rational curve on the three-dimensional variety:

$$V : y^2 = g(x_1)g(x_2)g(x_3)$$

(which, geometrically, is the quotient of $E \times E \times E$ by $(\mathbb{Z}/2\mathbb{Z})^2$, where each non-trivial element acts by $[-1]$ on two components and by the identity on the third one). Indeed, if $\phi : \mathbb{A}^1 \to V$, $t \mapsto (x_1(t), x_2(t), x_3(t), y(t))$ is such a rational curve (i.e., a rational map of the affine line $\mathbb{A}^1$, parametrized by t, to the variety V), then for any $u \in \mathbb{F}_q$ that is not a pole of ϕ, at least one of $g(x_i(u))$ for $i = 1, 2, 3$ is a quadratic residue (because the product of three quadratic non-residues is not a square, and hence cannot be equal to $y(u)^2$). This yields a well-defined point $\left(x_j(u), \sqrt{g(x_j(u))}\right) \in E(\mathbb{F}_q)$ where j is the first index such that $g(x_j(u))$ is a square, and thus we obtain the required encoding function.

Then, Shallue and van de Woestijne show how to construct such a rational curve ϕ (and in fact a large number of them). They first obtain an explicit rational map $\psi : S \to V$, where S is the surface of equation:

$$S : y^2 \cdot \left(u^2 + uv + v^2 + a(u + v) + b\right) = -g(u),$$

which can also be written, by completing the square with respect to v, as:

$$\left[y(v + \frac{1}{2}u + \frac{1}{2}a)\right]^2 + \left[\frac{3}{4}u^2 + \frac{1}{2}au + b - \frac{1}{4}a^2\right] y^2 = -g(u).$$

Now observe that for any fixed $u \in \mathbb{F}_q$, the previous equation defines a curve of genus 0 in the (v, y)-plane. More precisely, it can be written as:

$$z^2 + \alpha y^2 = -g(u)$$

with $z = y(v + \frac{1}{2}u + \frac{1}{2}a)$ and $\alpha = \frac{3}{4}u^2 + \frac{1}{2}au + b - \frac{1}{4}a^2$. This is a non-degenerate conic as soon as α and $g(u)$ are both non-zero (which happens for all $u \in \mathbb{F}_q$ except at most 5), and then admits a rational parametrization, yielding a rational curve $\mathbb{A}^1 \to S$. Composing with ψ, we get the required rational curve on V, and hence an encoding, provided that $q > 5$.

6.1.4. Icart's approach

In [89], Icart introduced an encoding function based on a very different idea, namely, trying to adapt the Boneh-Franklin encoding discussed in §6.1.1 to the case of an ordinary elliptic curve. More precisely, consider again an elliptic curve E given by a short Weierstrass equation:

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

over a field $\mathbb{F}_q$ of odd characteristic with $q \equiv 2 \mod 3$. The idea is again to reduce the equation to a binomial cubic, which can be solved directly in $\mathbb{F}_q$ (where $u \mapsto u^3$ is a bijection).

Unlike the simple case considered by Boneh and Franklin, this cannot be done by picking y as a constant: doing so results in a trinomial cubic which does not always have a root in $\mathbb{F}_q$. Icart's idea is to set $y = ux + v$ for two parameters u, v to be chosen later. This gives:

$$x^3 - u^2x^2 + (a - 2uv)x + b - v^2 = 0$$

and after completing the cube:

$$\left(x - \frac{u^2}{3}\right)^3 + \left(a - 2uv - \frac{u^4}{3}\right)x = v^2 - b - \frac{u^6}{27},$$

Thus, by setting $v = (a - 3u^4)/(6u)$, it is possible to cancel the term of degree 1 and obtain a binomial cubic equation:

$$\left(x - \frac{u^2}{3}\right)^3 = v^2 - b - \frac{u^6}{27},$$

which is easy to solve for x in $\mathbb{F}_q$. This gives Icart's encoding, described as follow:

$$\begin{aligned} f : \mathbb{F}_q &\to E(\mathbb{F}_q) \\ u &\mapsto \left(\left(v^2 - b - \frac{u^6}{27}\right)^{1/3} + \frac{u^2}{3}; ux + v\right) \end{aligned}$$

where $v = (a - 3u^4)/(6u)$. By convention, $f(0) = O$, the identity element.

This encoding applies to a more restricted setting than the Shallue-van de Woestijne encodings, due to the requirement that $q \equiv 2 \mod 3$, but it has the advantage of being very easy to describe and implement in constant time.

6.2. Hashing to pairing-friendly curves

In the previous section, we have described several constructions of encoding functions to elliptic curves. It is not clear, however, that they solve our initial problem of hashing to elliptic curve groups. There are two issues at play: the first is the lack of indifferentiability, and the second is the fact that we want to map to the subgroup $\mathbb{G}_1$ or $\mathbb{G}_2$ rather than the whole curve.

6.2.1. The issue of indifferentiability

The basic construction of a hash function $\mathcal{H} : \{0,1\}^* \to E(\mathbb{F}_q)$ from an $\mathbb{F}_q$-valued random oracle $\mathfrak{h} : \{0,1\}^* \to \mathbb{F}_q$ and an encoding $f : \mathbb{F}_q \to E(\mathbb{F}_q)$, as suggested in previous paragraphs, is simply:

$$\mathcal{H}(m) = f(\mathfrak{h}(m)). \tag{6.3}$$

However, unlike what happens for the Boneh-Franklin encoding, the resulting hash function $\mathcal{H}$ does not necessarily have strong security properties.

Consider the case when f is Icart's encoding, for example (most other encodings are similar). One can then prove some limited security properties on $\mathcal{H}$, such as that $\mathcal{H}$ is one-way if $\mathfrak{h}$ is [89, Lemma 5]. However, unlike the Boneh-Franklin encoding, f is not a surjective or "almost" surjective function to the target group $E(\mathbb{F}_q)$. Indeed, in his original paper [89], Icart could only show that the image $f(\mathbb{F}_q)$ satisfies $\#f(\mathbb{F}_q) \gtrsim (1/4)\cdot\#E(\mathbb{F}_q)$, and conjectured that, in fact, $\#f(\mathbb{F}_q) \approx (5/8)\cdot\#E(\mathbb{F}_q)$ (a conjecture which was later proved in [61, 64]). As a result, the hash function $\mathcal{H}$ constructed from f using formula in Equation (6.3) is easily distinguished from a random oracle!

To see this, note that since f is an algebraic function, we can efficiently compute $f^{-1}(P)$ for any $P \in E(\mathbb{F}_q)$ by solving a polynomial equation over $\mathbb{F}_q$. In particular, it is possible to decide efficiently whether P is in the image of f or not. Therefore, we can construct a distinguisher D between $\mathcal{H}_0 = \mathcal{H}$ and a random oracle $\mathcal{H}_1$ to $E(\mathbb{F}_q)$ as follows. D is given as input $P = \mathcal{H}_b(m) \in E(\mathbb{F}_q)$ for some message m and a random bit $b \in \{0,1\}$. It answers with a guess of the bit b, as $b = 0$ if P is in $f(\mathbb{F}_q)$ and $b = 1$ otherwise. Then D has a constant positive advantage. Indeed, it answers correctly with probability 1 if $P \notin f(\mathbb{F}_q)$, and with probability $1/2$ otherwise, hence it has a non-negligible advantage in the distinguishing game. Thus, clearly, construction in Equation (6.3) does not behave like a random oracle when f is Icart's encoding (or most other encodings), and cannot replace a random oracle in a generic way.

In many protocols requiring a hash function to an elliptic curve group, this is actually not much of a problem, and an encoding with an image size that is a constant fraction of $\#E(\mathbb{F}_q)$ is often good enough. The reason is that, in a random oracle proof of security, the simulator programs the random oracle by setting the hash of some message m to a value P, but that point P itself can usually be anything depending on some randomness. So the simulator might typically want to set $\mathcal{H}(m)$ to $P = [r]G$ for some random r, say. Now if $\mathcal{H}$ is defined in the protocol using a construction like Equation (6.3), the simulator would pick a random r and set $\mathfrak{h}(m)$ to one of the preimages $u \in f^{-1}(P)$ if $P \in f(\mathbb{F}_q)$. If, however, P is not in the image of f, the simulator would pick another random r and try again.

Nevertheless, it seems difficult to give formal sufficient conditions on a protocol for it to remain secure when the elliptic curve-valued random oracle is replaced by a construction like Equation (6.3). One can actually find protocols that are secure in the random oracle model, but in which using that construction instead breaks security completely [30].

Therefore, it would be desirable to obtain from the encodings discussed thus far a construction that does satisfy the *indifferentiability* property mentioned in §5.2.3.2, and can thus be used as a plug-in replacement for elliptic curve-valued random oracles in a very large class of protocols. The problem was solved by Brier *et al.* [30] in the case of Icart's function, and by Farashahi *et al.* [60] in general. They prove that the following construction achieves indifferentiability from a random oracle:

$$\mathcal{H}(m) = f(\mathfrak{h}_1(m)) + f(\mathfrak{h}_2(m)) \tag{6.4}$$

where $\mathfrak{h}_1$ and $\mathfrak{h}_2$ are modeled as random oracles $\{0,1\}^* \to \mathbb{F}_q$ (and the addition is the usual group operation in $E(\mathbb{F}_q)$). As a result, to obtain an efficient indifferentiable hash function construction, it suffices to know how to compute a function of the form of Equation (6.3) efficiently: do it twice, add the results together, and you get indifferentiability. Therefore, and since in many cases it is sufficient by itself, the form in Equation (6.3) is what the rest of this chapter will focus on.

6.2.2. Hashing to subgroups

Most of the discussion so far has focused on the problem of hashing to the whole group $E(\mathbb{F}_q)$ of points on the elliptic curve E. But this is not in fact what we need for pairing-based

cryptography: The groups we would like to hash to are pairing groups $\mathbb{G}_1$ and $\mathbb{G}_2$, which are *subgroups* of an elliptic curve group. Let us review how hashing to those subgroups works.

6.2.2.1. Hashing to $\mathbb{G}_1$

This case is simpler. Consider a pairing-friendly curve $E/\mathbb{F}_p$ over a prime field. The group $\mathbb{G}_1$ is just the group $E(\mathbb{F}_p)[r]$ of r-torsion points in $E(\mathbb{F}_p)$, for some large prime divisor r of $\#E(\mathbb{F}_p)$. If we denote by c the cofactor of E, i.e., the integer such that $\#E(\mathbb{F}_p) = c \cdot r$, then c is always coprime to r, and $\mathbb{G}_1$ can thus be obtained as the image of the homomorphism $[c]$ of multiplication by c in $E(\mathbb{F}_p)$.

Now suppose we are given some well-behaved hash function $\mathcal{H} : \{0,1\}^* \to E(\mathbb{F}_p)$. Then we can construct a map $\mathcal{H}_1 : \{0,1\}^* \to \mathbb{G}_1$ by defining $\mathcal{H}_1(m) = [c]\mathcal{H}(m)$, and it turns out that $\mathcal{H}_1$ is still a well-behaved hash function. For example, Brier *et al.* [30] show that if $\mathcal{H}$ is indifferentiable from a random oracle, then so is $\mathcal{H}_1$. This results from the fact that the muliplication-by-c homomorphism is efficiently computable, regular (all elements of $\mathbb{G}_1$ have the same number of preimages, namely c), and efficiently samplable (we can sample a uniformly distributed preimage of an element in $\mathbb{G}_1$ by adding to it a random element of order c in $E(\mathbb{F}_p)$, which we can, for example, generate as $[r]P$ for P, a uniformly sampled random point).

As a result, to hash to $\mathbb{G}_1$ efficiently, we simply need an efficient way of computing some curve-valued hash function $\mathcal{H}$, and an efficient way of evaluating the multiplication-by-c map. The latter is typically quite cheap.

6.2.2.2. Hashing to $\mathbb{G}_2$

This case is more complicated in general. Indeed, generally speaking, $\mathbb{G}_2$ is one specific subgroup of order r in the group $E(\mathbb{F}_{p^k})[r]$ of r-torsion points of E over the embedding field $\mathbb{F}_{p^k}$. But since $E(\mathbb{F}_{p^k})[r]$ is isomorphic to $(\mathbb{Z}/r\mathbb{Z})^2$, there are many subgroups of order r, and one cannot just multiply by some cofactor to map into $\mathbb{G}_2$. The approach used to hash to $\mathbb{G}_2$ will differ according to which of the three *pairing types*, in the sense of Galbraith, Paterson, and Smart [71], we are working with.

For Type I pairings, the distortion map provides an efficiently computable isomorphism from $\mathbb{G}_1$ to $\mathbb{G}_2$. Therefore, we can simply hash to $\mathbb{G}_1$ as above and compose with the distortion map to obtain a hash function to $\mathbb{G}_2$.

For Type II pairings, $\mathbb{G}_2$ is not the image of any efficiently computable homomorphism, and as a result, there is in fact no way of efficiently hashing to that group. *One cannot instantiate protocols that require hashing to $\mathbb{G}_2$ in the Type II setting.* In rare cases where one really needs both the ability to hash to $\mathbb{G}_2$ and the existence of a one-way isomorphism $\mathbb{G}_2 \to \mathbb{G}_1$, a possible workaround is to replace $\mathbb{G}_2$ with the entire group $E(\mathbb{F}_{p^k})[r]$ of order r^2, which we can hash to as above using the multiplication by $\#E(\mathbb{F}_{p^k})/r^2$ in $E(\mathbb{F}_{p^k})$. This is usually called the Type IV pairing setting [38, 36]. The cofactor multiplication in that case is quite costly, so it may be interesting to optimize it. However, it is usually possible to convert such protocols to the significantly more efficient Type III setting [37], so we will not consider that case further in the rest of this chapter.

For Type III pairings, $\mathbb{G}_2 \subset E(\mathbb{F}_{p^k})[r]$ is the eigenspace of the Frobenius endomorphism π associated with the eigenvalue q, and the complementary subspace of $\mathbb{G}_1$ (which is the eigenspace for the eigenvalue 1). As a result, $\mathbb{G}_2$ is the image of the efficient endomorphism $\frac{\pi - Id}{q-1}$ of $E(\mathbb{F}_{p^k})[r]$. Therefore, we can hash to $\mathbb{G}_2$ by cofactor multiplication to get into $E(\mathbb{F}_{p^k})[r]$, composed with that endomorphism. In practice, however, it is much more preferable to represent $\mathbb{G}_2$ as a subgroup in the degree-d twist E' of E over a lower degree extension $\mathbb{F}_{p^{k/d}}$. Doing so, $\mathbb{G}_2$ simply appears as the subgroup $E'(\mathbb{F}_{p^{k/d}})[r]$ of r-torsion points on that curve, and hashing can be done exactly as in the case of $\mathbb{G}_1$. Contrary to the case of $\mathbb{G}_1$,

however, the cofactor in this case is usually quite large, and it is thus a major concern to make it as fast as possible. This is one of the main issues discussed in the coming sections.

6.3. Case study: the Barreto-Naehrig elliptic curves

In this section we discuss how to apply the the Shallue and van de Woestijne encoding described in §6.1.3 along with the hashing techniques discussed in §6.2, in order to construct points that belong to the groups of a popular instantiation of Type III pairings, namely, bilinear pairings implemented over the BN elliptic curves defined in §5.1.2.1.

6.3.1. Constant-time hashing to $\mathbb{G}_1$

In Latincrypt 2012, Fouque and Tibouchi [65] presented a specialization of the procedure proposed by Shallue and van de Woestijne [152] applied to Barreto-Naehrig curves, which are defined over the finite field $\mathbb{F}_p$, with $p \equiv 7 \mod 12$, or $p \equiv 1 \mod 12$. The mapping covers a 9/16 fraction of the prime group size $r = \#E(\mathbb{F}_p)$. In a nutshell, the procedure proposed in [65] consists in the following.

Let t be an arbitrary non-zero element in the base field $\mathbb{F}_p^*$ such that $x_1, x_2, x_3 \in \mathbb{F}_p*$ are defined as

$$\begin{aligned} x_1 &= \frac{-1+\sqrt{(-3)}}{2} - \frac{\sqrt{-3}\cdot t^2}{1+b+t^2}, \\ x_2 &= \frac{-1-\sqrt{(-3)}}{2} + \frac{\sqrt{-3}\cdot t^2}{1+b+t^2}, \\ x_3 &= 1 - \frac{(1+b+t^2)^2}{3t^2}. \end{aligned}$$

The Shallue-van de Woestijne encoding applied to the Barreto-Naehrig curves of the form $E : y^2 = g(x) = x^3 + b$ is given by the following projection:

$$\begin{aligned} f : \mathbb{F}_p^* &\rightarrow E(\mathbb{F}_p) \\ t &\mapsto \left(x_i, \chi_p(t)\cdot\sqrt{g(x_i)}\right), \end{aligned}$$

where the index $i \in \{1,2,3\}$ is the smallest integer such that $g(x_i)$ is a square in $\mathbb{F}_p$ and the function $\chi_p : \{-1,0,1\}$, computes the non-trivial quadratic character over the field $\mathbb{F}_p^*$, also known as quadratic residuosity test. The procedure just outlined is presented in Algorithm 23.

Remark 6.1 (The Barreto-Naehrig curve $\mathbb{G}_1$ subgroup)**.** *Notice that the Barreto-Naehrig curves are exceptional in the sense that the subgroup $\mathbb{G}_1$ is exactly the same as $E(\mathbb{F}_p)$. In other words, for this case, the cofactor c is equal to one, and therefore, the procedure presented in Algorithm 23 effectively completes the hashing to $\mathbb{G}_1$.*

Remark 6.2 (Implementation aspects)**.** *All the computations of Algorithm 23 are performed over the base field $\mathbb{F}_p$ at a cost of two field inversions, three quadratic character tests, one square root, and few field multiplications. Notice that the values $\sqrt{-3}$ and $\frac{-1+\sqrt{-3}}{2}$ are precomputed offline in Steps 1–2. Moreover, when p is chosen such that $p \equiv 3 \mod 4$, the square root $\sqrt{x^3+b}$ (line 10) can be computed by the power $(x_i^3+b)^{\frac{p+1}{4}}$.*

In order to ensure a constant-time behavior, the quadratic residuosity test of a field element a can be computed by performing the exponentiation $a^{\frac{p-1}{2}}$. Alternatively, one can performs the quadratic residuosity test by recursively applying Gauss' law of quadratic

Algorithm 23 Constant-time hash function to $\mathbb{G}_1$ on BN curves [65]

Input: $t \in \mathbb{F}_p^*$, parameter $b \in \mathbb{F}_p$ of $E/\mathbb{F}_p : y^2 = x^3 + b$.
Output: A point Point $P = (x, y) \in \mathbb{G}_1$

Precomputation:
1: $sqrt_3 \leftarrow \sqrt{-3}$
2: $j \leftarrow (-1 + sqrt_3)/2$
Computation:
3: $w \leftarrow sqrt_3 \cdot \frac{t}{1+b+t^2}$
4: $x_1 \leftarrow j - t \cdot w$
5: $x_2 \leftarrow -1 - x_1$
6: $x_3 \leftarrow 1 + 1/w^2$
7: $\alpha \leftarrow \chi_q(x_1^3 + b)$ ▷ Using Euler's Criterion: $(x_1^3 + b)^{\frac{p-1}{2}}$
8: $\beta \leftarrow \chi_q(x_2^3 + b)$ ▷ Using Euler's Criterion: $(x_2^3 + b)^{\frac{p-1}{2}}$
9: $i \leftarrow [(\alpha - 1) \cdot \beta \mod 3] + 1$
10: **return** $P \leftarrow \left(x_i, \chi_p(t) \cdot \sqrt{x_i^3 + b}\right)$ ▷ Using Euler's Criterion: $t^{\frac{p-1}{2}}$

reciprocity at a computational cost similar to computing the greatest common divisor of a and p. Unfortunately, it is difficult to implement it in constant-time. That is why the authors of [65] suggested using blinding techniques in order to thwart potential timing attacks. This variant was adopted in several papers such as [41, 168] and implemented in Algorithm 24.

Algorithm 24 Blind factor version of the Hash function to $\mathbb{G}_1$ on Barreto-Naehrig curves [65]

Input: $t \in \mathbb{F}_p^*$, parameter $b \in \mathbb{F}_p$ of $E/\mathbb{F}_p : y^2 = x^3 + b$.
Output: A point Point $P = (x, y) \in \mathbb{G}_1$

Precomputation:
1: $sqrt_3 \leftarrow \sqrt{-3}$
2: $j \leftarrow (-1 + sqrt_3)/2$
Computation:
3: $w \leftarrow sqrt_3 \cdot \frac{t}{1+b+t^2}$
4: $x_1 \leftarrow j - t \cdot w$
5: $x_2 \leftarrow -1 - x_1$
6: $x_3 \leftarrow 1 + 1/w^2$
7: $r_1, r_2, r_3 \xleftarrow{\$} \mathbb{F}_p^*$
8: $\alpha \leftarrow \chi_q(r_1^2 \cdot (x_1^3 + b))$ ▷ Using blind factor approach
9: $\beta \leftarrow \chi_q(r_2^2 \cdot (x_2^3 + b))$ ▷ Using blind factor approach
10: $i \leftarrow [(\alpha - 1) \cdot \beta \mod 3] + 1$
11: **return** $P \leftarrow \left(x_i, \chi_p(r_3^2 \cdot t) \cdot \sqrt{x_i^3 + b}\right)$ ▷ Using blind factor approach

Remark 6.3 (On the security of Algorithm 24)**.** *Strictly speaking, the blinding factor protection of Algorithm 24 is not provably secure against timing attacks. This is because even if the blinding factors are uniformly distributed in the base field, and kept unknown to the adversary, the input of the algorithm computing the quadratic characther χ_p, is not uniformly distributed in all of $\mathbb{F}_p^*$, but only among its quadratic residues or quadratic non-residues. Practically speaking, this is not very significant: Very little secret information can leak in that way. Moreover, if the quadratic residuosity test is performed in constant-time, then no information will be leaked at all. Nevertheless, it is always possible to achieve provable protection through additional blinding. For example, one can randomly multiply by a blind*

factor that is a known square/non-square with probability 1/2, and then adjust the output accordingly.

6.3.2. Deterministic construction of points in $E'(\mathbb{F}_{p^2})$ for BN curves

The Barreto-Naehrig family of elliptic curves has an embedding degree of $k = 12$, and an associated twist curve E' with degree $d = 6$. This defines the group $\mathbb{G}_2$ as

$$\mathbb{G}_2 = E'(\mathbb{F}_{p^{k/d}})[r] = E'(\mathbb{F}_{p^2})[r].$$

As already mentioned, the encoding described by Fouque and Tibouchi in [65] applies over finite fields $\mathbb{F}_p$, where $p \equiv 7 \mod 12$ or $p \equiv 1 \mod 12$. In the case of the Barreto-Naehrig curves, one observes that since $p \equiv 7 \mod 12$, then $p^2 \equiv 1 \mod 12$. As a result, the encoding presented in [65] can be applied as it is in order to find random points over $E'/\mathbb{F}_{p^2}$, except that several computations must be performed over the quadratic field extension $\mathbb{F}_{p^2}$. The corresponding procedure is shown in Algorithm 25.

Algorithm 25 Deterministic construction of points in $E'(\mathbb{F}_{p^2})$ for Barreto-Naehrig curves.

Input: $t \in \mathbb{F}_p^*$, parameter $B = b_0 + b_1 u \in \mathbb{F}_{p^2}$ of $E'/\mathbb{F}_{p^2} : Y^2 = X^3 + B$.
Output: A point Point $Q = (x, y) \in E'(\mathbb{F}_{p^2})$

Precomputation:
1: $sqrt_3 \leftarrow \sqrt{-3}$
2: $j \leftarrow (-1 + sqrt_3)/2$
Computation:
3: $a_0 \leftarrow 1 + b_0 + t^2$
4: $a_1 \leftarrow b_1$
5: $A \leftarrow 1/A$ ▷ with $A = a_0 + a_1 u \in \mathbb{F}_{p^2}$
6: $c \leftarrow sqrt_3 \cdot t$
7: $W \leftarrow (c \cdot a_0) + (c \cdot a_1)u$ ▷ with $W = w_0 + w_1 u \in \mathbb{F}_{p^2}$
8: $a_0 \leftarrow w_0 \cdot t$
9: $a_1 \leftarrow w_1 \cdot t$
10: $X_1 \leftarrow (j - a_0) - a_1 u$ ▷ with $X_1 = x_{1,0} + x_{1,1} u \in \mathbb{F}_{p^2}$
11: $X_2 \leftarrow (-x_{i,0} - 1) - x_{1,1} u$ ▷ with $X_2 = x_{2,0} + x_{2,1} u \in \mathbb{F}_{p^2}$
12: $X_3 \leftarrow 1/W^2$ ▷ with $X_3 = x_{3,0} + x_{3,1} u \in \mathbb{F}_{p^2}$
13: $X_3 \leftarrow (1 + x_{3,0}) + x_{3,1} u$
14: $\alpha \leftarrow \chi_{p^2}(X_1^3 + B)$
15: $\beta \leftarrow \chi_{p^2}(X_2^3 + B)$
16: $i \leftarrow [(\alpha - 1) \cdot \beta \mod 3] + 1$
17: **return** $Q \leftarrow \left(X_i, \chi_{p^2}(t) \cdot \sqrt{X_i^3 + B}\right)$

Notice that all the operations of Algorithm 25 are performed over the base field $\mathbb{F}_p$ and its quadratic extension $\mathbb{F}_{p^2} = \mathbb{F}_p[u]/u^2 - \beta$, where $\beta = -1$ is not a square over $\mathbb{F}_p$. In particular, the steps 14 and 15 of Algorithm 25 must compute in constant-time the quadratic character $\chi_{p^2}(\cdot)$ over the extension field $\mathbb{F}_{p^2}$. To this end, one can use the procedure described in [4], which is an improvement over the work made by Bach and Huber [9]. The authors of [4] proposed to compute the quadratic character over the quadratic field extension $\mathbb{F}_{p^2}$ by descending the computation to the base field $\mathbb{F}_p$, using the following identity:

$$\begin{aligned} \chi_{p^2}(a) = a^{\frac{p^2-1}{2}} &= (a \cdot a^p)^{\frac{p-1}{2}} \\ &= (a \cdot \bar{a})^{\frac{p-1}{2}} = \chi_p(a \cdot \bar{a}) = \chi_p(|a|) \end{aligned} \tag{6.5}$$

where $\bar{a} = a_0 - a_1 u$ and $|a|$ is the conjugate and the norm of a, respectively. The above computation can be carried out at a cost of two squarings, one addition, and the computation of the quadractic character $\chi_p(|a|)$. As before, $\chi_p(|a|)$ can be computed either by performing one exponentiation over the base field $\mathbb{F}_p$, or alternatively, by applying blinding techniques.

Furthermore, in line 17 of Algorithm 25 one needs to extract a square root over the quadratic field $\mathbb{F}_{p^2}$. This operation can be efficiently computed in constant-time using the complex method proposed by Scott in [147].

Remark 6.4. *Notice that in line 17 of Algorithm 25, the procedure proposed in [65] guarantees that the term $X_i^3 + B$ is always a quadratic residue over the field $\mathbb{F}_{p^2}$. Hence, one can safely omit the steps that verifies if the square root exists in the method proposed by Scott [147].*

6.3.3. Efficient hashing to $\mathbb{G}_2$

In this section we are interested in the efficient computation of the third mapping shown in Figure 5.2, namely, the computation of the scalar multiplication, $Q' = [c]\tilde{Q}$.

Let $E'(\mathbb{F}_{p^{k/d}})$ be an abelian group of order $\#E'(\mathbb{F}_{p^{k/d}}) = c \cdot r$, where c is a composite integer known as the cofactor of the twist elliptic curve E'. As we have seen, hashing to $\mathbb{G}_2$ can be done by deterministically selecting a random point $\tilde{Q}$ in $E'(\mathbb{F}_{p^{k/d}})$ we have that

$$\{[c]\tilde{Q} \mid \tilde{Q} \in E'(\mathbb{F}_{p^{k/d}})\} = \{Q' \in E'(\mathbb{F}_{p^{k/d}}) \mid [r]Q' = \mathcal{O}\}.$$

However, since in most pairing-friendly elliptic curves the cofactor c in the group $\mathbb{G}_2$ has a considerably large size, which is certainly much larger than the prime order r, a direct computation of such scalar multiplication will be quite costly.

In the rest of this section, we describe a method which for several families of pairing-friendly elliptic curves, allows us to compute the scalar multiplication $Q' = [c]\tilde{Q}$ on a time-computational complexity of $O(1/\varphi(k)\log c)$. The following material closely follows the discussion presented in [150, 69, 110].

6.3.3.1. The Fuentes *et al.* method

Observe that a multiple c' of the cofactor c such that $c' \not\equiv 0 \pmod{r}$ will also hash correctly to the group $\mathbb{G}_2$, since the point $[c']\tilde{Q}$ is also in $E(\mathbb{F}_{p^{k/d}})[r]$. The method presented in [69, 110] is based in the following theorem.

Theorem 6.1. *Since $p \equiv 1 \mod d$ and $E'(\mathbb{F}_{p^{k/d}})$ is a cyclic group, then there exists a polynomial $h(z) = h_0 + h_1 z + \cdots + h_{\varphi(k)-1} z^{\varphi(k)-1} \in \mathbb{Z}[z]$ such that $[h(\Psi)]P$ is a multiple of $[c]P$ for all $P \in E'(\mathbb{F}_{p^{k/d}})$ and $|h_i|^{\varphi(k)} \leq \#E'(\mathbb{F}_{p^{k/d}})/r$ for all i.*

This Theorem proved in [69] by means of the following two auxiliary lemmas.

Lemma 6.1. *Let d be the degree of the twist curve E', if $p \equiv 1 \mod d$, then $\Psi(\tilde{Q}) \in E'(\mathbb{F}_{p^{k/d}})$, for all $\tilde{Q} \in E'(\mathbb{F}_{p^{k/d}})$.*

The above lemma proves that the endomorphism $\Psi : E' \to E$, defined over $\mathbb{F}_{q^d}$, fixes $\tilde{E}(\mathbb{F}_q)$ as a set. The next lemma shows the effect of ψ on elements in $\tilde{E}(\mathbb{F}_q)$.

Lemma 6.2. *Let $t^2 - 4p = Df^2$ and $\tilde{t} - 4q = D\tilde{f}^2$, for some value of f and $\tilde{f}$, where $q = p^{k/d}$ and D is the discriminant of the curve E. Also, let $\tilde{n} = \#E'(\mathbb{F}_{p^{k/d}})$. If the following conditions are satisfied,*

- $p \equiv 1 \mod d$,

- $\gcd(\tilde{f}, \tilde{n}) = 1$,
- $E'(\mathbb{F}_{p^{k/d}})$ *is cyclic,*

then $\psi(\tilde{Q}) = [a]\tilde{Q}$ *for all* $\tilde{Q} \in E'(\mathbb{F}_{p^{k/d}})$*, where,* $a = (t \pm f(\tilde{t}-2)/\tilde{f})/2$.

Once the value of a such that $[a]\tilde{Q} = \psi(\tilde{Q})$ has been computed, it is necessary to find the polynomial $h \in \mathbb{Z}[w]$, with the smallest coefficients, such that $h(a) \equiv 0 \mod c$. To this end, one needs to consider a matrix M, with rows representing the polynomials $h_i(w) = w^i - a^i$, such that $h_i(a) \equiv 0 \mod c$. Hence, any linear combination of the rows of the matrix M will correspond with a polynomial $h'(w)$ that satisfies the above condition.

Since the Frobenius endomorphism π acting over $E(\mathbb{F}_{p^k})$ has order k, and since ψ is an endomorphism that acts on the cyclic group $E'(\mathbb{F}_{p^{k/d}})$, then ψ operating over $E'(\mathbb{F}_{p^{k/d}})$ also has order k. Furthermore, since the integer number a satisfies the congruence $\Phi_k(a) \equiv 0 \mod \tilde{n}$, where Φ_k is the k-th cyclotomic polynomial, then the polynomials $h(w) = w^i - a^i$ with $i \geq \varphi(k)$ can be written as linear combinations of $w-a, \ldots, w^{\varphi(k)-1} - a^{\varphi(k)-1} \mod c$, where $\varphi(\cdot)$ is the Euler's totient function. Because of the aforementioned argument, only the polynomials with degree less than $\varphi(k)$ are considered, as follow.

$$M = \begin{array}{c} \begin{matrix} a^0 & a^1 & a^2 & \cdots & a^{\varphi(k)-1} \end{matrix} \\ \begin{pmatrix} c & 0 & 0 & \cdots & 0 \\ -a & 1 & 0 & \cdots & 0 \\ -a^2 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & & \ddots & \\ -a^{\varphi(k)-1} & 0 & 0 & \cdots & 1 \end{pmatrix} \end{array} \longrightarrow \begin{array}{rcl} c & \equiv & 0 \mod c \\ -a+a & \equiv & 0 \mod c \\ -a^2+a^2 & \equiv & 0 \mod c \\ \vdots & \vdots & \vdots \\ -a^{\varphi(k)-1}+a^{\varphi(k)-1} & \equiv & 0 \mod c \end{array}$$

In this case, the rows of the matrix M can be seen as vectors, which form a lattice basis. Now, the Lenstra-Lenstra-Lovász algorithm [118] can be applied to M in order to obtain an integer basis for M with small entries. According to the Minkowski's theorem [129], a vector v that represents a linear combination of the basis of the lattice L, will be found. This solution will correspond to the polynomial h with coefficients smaller than $|c|^{1/\varphi(k)}$.

In the rest of this Section, explicit equations for computing the hash to the group $\mathbb{G}_2$ on the BN curves [13] with embedding degree $k = 12$.

6.3.3.2. Barreto-Naehrig curves

For the Barreto-Naehrig elliptic curves, the group order $\tilde{n} = \#E'(\mathbb{F}_{p^2})$ and the trace of the twist E' over $\mathbb{F}_{p^2}$, are parametrized as follows:

$$\begin{aligned} \tilde{n} &= (36x^4 + 36x^3 + 18x^2 + 6x + 1)(36x^4 + 36x^3 + 30x^2 + 6x + 1), \\ \tilde{t} &= 36x^4 + 1 \end{aligned}$$

where $\tilde{n}(x) = r(x)c(x)$. Using Lemma 6.2, we find that

$$a(x) = -\frac{1}{5}(3456x^7 + 6696x^6 + 7488x^5 + 4932x^4 + 2112x^3 + 588x^2 + 106x + 6).$$

It is interesting to notice that $a(x) \equiv p(x) \mod r(x)$ and $\psi(Q') = [a(x)]Q' = [p(x)]Q'$ for all $Q' \in \tilde{E}(\mathbb{F}_{p^2})[r]$. Note that $a(x) \equiv p(x) (\mathrm{mod}\ r)$ and thus $\psi Q = [a(x)]Q = [p(x)]Q$ for all $Q \in \tilde{E}(\mathbb{F}_q)[r]$.

Following the strategy mentioned above, the lattice L can be built, and then reducing $-a(x)^i$ modulo $c(x)$, one obtains

$$\left[\begin{array}{c|ccc} c(x) & 0 & 0 & 0 \\ -a(x) & 1 & 0 & 0 \\ -a(x)^2 & 0 & 1 & 0 \\ -a(x)^3 & 0 & 0 & 1 \end{array}\right] \rightarrow \left[\begin{array}{c|ccc} 36x^4 + 36x^3 + 30x^2 + 6x + 1 & 0 & 0 & 0 \\ 48/5x^3 + 6x^2 + 4x - 2/5 & 1 & 0 & 0 \\ 36/5x^3 + 6x^2 + 6x + 1/5 & 0 & 1 & 0 \\ 12x^3 + 12x^2 + 8x + 1 & 0 & 0 & 1 \end{array}\right].$$

From this lattice, one finds the polynomial $h(x) = x+3xz+xz^2+z^3$. Working modulo $\tilde{n}(x)$, we have that

$$h(a) = -(18x^3 + 12x^2 + 3x + 1)c(x),$$

and since $\gcd(18x^3 + 12x^2 + 3x + 1, r(x)) = 1$, the following map is a homomorphism of $\tilde{E}(\mathbb{F}_q)$ with image $\tilde{E}(\mathbb{F}_q)[r]$:

$$Q \mapsto [x]Q + \psi([3x]Q) + \psi^2([x]Q) + \psi^3(Q).$$

We can compute $Q \mapsto [x]Q \mapsto [2x]Q \mapsto [3x]Q$ using one doubling, one addition, and one multiply-by-x. Given Q, $[x]Q$, $[3x]Q$, we can compute $[h(a)]Q$ using three ψ-maps, and three additions. In total, we require one doubling, four additions, one multiply-by-x, and three ψ-maps.

6.4. Implementation

This section presents an implementation of the hashing to the groups $\mathbb{G}_1$ and $\mathbb{G}_2$ procedures, as defined in the Barreto-Naehrig curves. We decided to use the Barreto-Naehrig curves, mainly because they are the preferred curves for efficient implementations of bilinear pairings in many pairing libraries, such as [6, 130, 168].

6.4.1. Intel processor

Table 6.1 reports the timings (in 10^3 clock cycles) achieved by our software implementation of all the required building blocks for computing the hash functions to the Barreto-Naehrig curve subgroups $\mathbb{G}_1$ and $\mathbb{G}_2$. Our library was written in the C and C++ languages and compiled with `gcc 4.9.2`. It was run on a Haswell Intel core i7-4700MQ processor running at 2.4 GHz, with both the Turbo-Boost and Hyper-threading technologies disabled.

We used two values of the Barreto-Naehrig parameter x, namely, $x_0 = -(2^{62} + 2^{55} + 1)$, which is a standard choice recommended in [74] and used in many pairing libraries, such as [6, 130, 168]. We also report the timings for the parameter choice: $x_1 = -(2^{62} + 2^{47} + 2^{38} + 2^{37} + 2^{14} + 2^7 + 1)$, which is the value recommended in [149] to avoid subgroup attacks in the group $\mathbb{G}_T$.

Group	**Operation**	**Parameter x for BN curves**	
		x_0	x_1
$\mathbb{F}_p$	SHA256	1.81	1.81
$\mathbb{G}_1$	Algorithm 23	122.81	156.48
	Algorithm 24	95.83	104.83
	Hash $\mathbb{G}_1$ with Alg. 23	124.62	158.29
	Hash $\mathbb{G}_1$ with Alg. 24	97.64	106.64
$\mathbb{G}_2$	$[c]\tilde{Q}$	161.63	175.02
	Alg. 25 (in CT)	186.71	236.94
	Alg. 25 (with BF)	134.03	153.12
	Hash $\mathbb{G}_2$ with Alg. 25 (in CT)	344.82	408.24
	Hash $\mathbb{G}_2$ with Alg. 25 (with BF)	293.29	322.21

Table 6.1: Cost of the main operations for hashing into $\mathbb{G}_1$ and $\mathbb{G}_2$ using BN curves at the 128-bit security level over Intel processor.

6.4.2. ARM processor

Table 6.2 reports the timings (in 10^3 clock cycles) achieved by our software implementation of all the required building blocks for computing the hash functions to the Barreto-Naehrig curve subgroups $\mathbb{G}_1$ and $\mathbb{G}_2$. Our library was written in the C language, taking advantage of the NEON technology, and compiled using the `android-ndk-r10b` Native Development Kit, and executed on an ARM Exynos 5 Cortex-A15 platform running at 1.7GHz. Once again, we used the two values for the x parameters chosen for the Intel processor in the preceding section.

Group	**Operation**	**Parameter x for BN curves**	
		x_0	x_1
$\mathbb{F}_p$	SHA256	8.67	8.67
$\mathbb{G}_1$	Algorithm 23	1047.80	1357.71
	Algorithm 24	655.32	709.40
	Hash $\mathbb{G}_1$ with Alg. 23	1056.43	1366.23
	Hash $\mathbb{G}_1$ with Alg. 24	664.02	718.12
$\mathbb{G}_2$	$[c]\tilde{Q}$	754.10	806.06
	Alg. 25 (in CT)	1337.21	1717.81
	Alg. 25 (with BF)	809.01	863.29
	Hash $\mathbb{G}_2$ with Alg. 25 (in CT)	2099.95	2530.20
	Hash $\mathbb{G}_2$ with Alg. 25 (with BF)	1570.75	1680.37

Table 6.2: Cost of the main operations for hashing into $\mathbb{G}_1$ and $\mathbb{G}_2$ using BN curves at the 128-bit security level Over ARM processor.

Chapter 7

Protected implementation of pairing-based authentication protocols

In this section the problem of authentication is addressed, which is undoubtedly one of the most important goals of modern cryptography. Throughout this section we describe a software implementation of paring-based two-factor authentication protocols performed in a secure way, in order to thwart simple side-channel attacks. Such protocols allow authenticate two entities using a four-digit password and a software token. The implementation takes advantage of the ARM Cortex-A processors features found in recent mobiles devices, and also of the Intel Haswell processors found in contemporary laptops models.

7.1. Introduction

In recent years, it has massively increased the use of mobile devices such as smart phones and tablets, which allow to perform financial transactions, multimedia processing, and a large number of tasks, since they are equipped with powerful processors. For this reason, they have become a target for attackers because users not only use them as communication devices, but also as devices where its sensitive information is stored. Considering this, it becomes necessary to provide security services such as authentication, confidentiality, integrity, non-repudiation, and access control that protect the data used in such devices. This security services generally are performed through cryptographic protocols that require efficient, fast and protected implementations.

Specifically referring to elliptic curve or pairing based protocols, which are the main interest in this section, we found that most works in the state-of-the-art are focused on just implementing the main primitives used in this kind of protocols, such as point scalar multiplication or the pairing function, leaving aside the implementation of complete protocols.

For example, in 2012 Tolga Acar *et al.* [1] implemented the optimal Ate pairing over the family of curves Barreto-Naehrig (BN) offering a security level of 128- and 192-bits. The authors used affine and projective coordinates for its computations on an ARM Cortex-A9 processor, concluding that affine coordinates are more efficient that projective ones when it is used a security level higher than 128-bits; later, Gurleen Grewal et al. [79], in the same year, presented an implementation of optimal Ate pairing also in BN curves using different security levels over ARM processors. They observed that projective homogeneous coordinates are unequivocally the best choice for pairing computation, when a security level of 128-bits and high optimization levels are used; in 2014, Faz-Hernández *et al.* [63] proposed an algorithm for computing point scalar multiplications in a secure way in order to thwart

side-channel attacks. Also they apply a novel technique that interleaves ARM and NEON instructions to perform a fast finite field arithmetic.

One of the few works of which we have knowledge that a complete protocol was implemented on mobile devices is the performed by Sánchez *et al.* [145], where the authors implemented an unprotected cryptographic library that supports an attribute-based encryption protocol, offering a security level of 127-bits using the NEON technology of ARM platforms.

We present in this section an implementation of a software library that supports two pairing-based multi-factor authentication protocols, which were proposed by Michael Scott in [148, 149]. Our implementation is efficient and protected against timing-attacks since we used the following strategies: the library was designed and adapted specifically to compute optimal pairings over BN curves. We performed a protected point scalar multiplication and we used a constant-time implementation of hashing into the groups $\mathbb{G}_1$ and $\mathbb{G}_2$ (see §6.1). In both implementations we offered a security level of 127-bits and regular executions are guaranteed on secret data, *i.e.* constant-time execution. In addition, we demonstrate that implementation of these protocols is feasible in mobile devices with restricted computing capacity, particularly those equipped with an ARM Cortex-A15 processor, which allows the usage of the NEON instruction set, and those equipped with with an Intel Core i7 processor.

7.1.1. Authentication

Authentication is undoubtedly one of the most important goals of modern cryptography. This security service allows to certify the identity of the participants in a communication protocol, using mechanisms such as digital signatures, passwords, or biometric characteristics. Generally, the authentication process consist of two phases: first, the identification phase that corresponds to the assignment of an unique identifier for each user; and the verification phase that consist of generating information from an identifier to validate the relationship user-identifier.

In a client-server scenario, identifying a server using public key cryptographic methods is considered a solved problem. However, the process that a client should follow in order to be authenticated is more problematic. There exist systems that rely in passwords to perform such authentication. However, those systems are faced with the fact that users find it difficult to handle long passwords, so typically they choose short and easy to remember passwords that have low entropy. This fact in turn has an impact in the security of computer systems, because this systems becomes vulnerable to dictionary or exhaustive search attacks.

One way to solve the above problem is using multi-factor authentication that is a technique that has proven to be difficult to infringe. However, this kind of authentication has as disadvantage the high cost associated with its implementation in a system.

7.1.1.1. Multi-factor authentication

Multi-factor authentication consist in verifying two or more aspects about the user. An example of this is the two-factor authentication process realized at ATMs, where a bank card and a four-digit Personal Identification Number (PIN) is used. Generally, the factors that are verified during the authentication refer to one of the following user's aspects: something that the user knows, for example a password; something that the user has, could be a physical token, smart card or USB memory; or something that the user is, which refers to a biometric feature such as its fingerprint or its iris image, among many others.

Most multi-factor authentication protocols use passwords and smart cards, because they can take advantage of characteristics provided by the smart cards such as its computing capacity and difficulty of cloning. However, this cards have some disadvantages derived of the cost of purchase, issuance, and management of tokens. Besides, from the users point of view, employing more than one authentication factor requires the management of several

tokens, which can be complicated. A more accessible alternative is to use mobile devices as tokens, since they have computing capacity and allow to include useful information to deal with different systems that implement this type of authentication. This alternative is the one studied in the rest of the section.

7.2. Two-factor authentication protocols

Recently, Michael Scott proposed [148, 149] a pair of pairing-based authentication protocols that use as factors a four-digit PIN and a software token. This latter factor, allows to use a mobile device as physical token to perform multi-factor authentication in one or more systems, keeping the tokens in a single device. In the following we detail the protocols used in this work.

In 2012, Michael Scott [148] proposed the protocol shown in Figure 7.1 which is balanced, in the sense that the operations performed by the client and the server are the same. So, it is advisable to use this protocol in devices with sufficient resources such as computational power and memory. On the other hand, in 2013 the author presented the protocol shown in Figure 7.2 which is unbalanced, because it was designed to be used on a client device with restricted resources. In this way, the client performs the minimum number of operations, and the server having sufficient resources, performs the most expensive operations. Next, the three main blocks of these protocols are described in detail.

Registry

This registry phase is the same for both protocols. In this phase, the identity-based secrets for both clients and servers are provided. In order to perform this task the existence of a certifying authority (CA) is assumed, which using its secret key s computes the identity-based secrets as $S_c = [s]C \in \mathbb{G}_1$ and $S_c = [s]C \in \mathbb{G}_1$ for clients and servers, respectively. In the scalar multiplications C and S represent the evaluation of client's identity $Id_c \in \{0,1\}^*$ and server's identity $Id_s \in \{0,1\}^*$ in the functions $\mathcal{H}_1$ and $\mathcal{H}_2$ that computes the hash to the groups $\mathbb{G}_1$ and $\mathbb{G}_2$ (see §6.1), respectively.

Authentication phase

Once the registration phase is finished and before the authentication process begins, the client must compute its token using a four-digit PIN as follows $Token = S_c - [PIN]C$. After that, the client and server can authenticate each other and agree a secret key K to encrypt subsequent communications using Figure 7.1 or 7.2 protocol. In such figures, r denotes the order of the groups involved in the pairing computation and $\mathfrak{h}(\cdot)$ represent the standard hash function $\mathfrak{h} : \{0,1\}^* \to \mathbb{Z}_r$.

PIN change and PIN recovery

Note that in both protocols to change the client's PIN the server participation is not necessary. Therefore, when the client decide to change its current PIN α for a new one β it must compute a new $Token = S_c - [\beta]C$. It should be noted that the previous token cannot be used to complete the authentication process if α is unknown.

On the other hand, when a client wants to recover a forgotten PIN he needs to interacts with the server in order to obtain its PIN. Thus, the server deliberately creates a situation where the she can perform exhaustive search attack over the set of PINs. For example, in the case of Figure 7.1 protocol this process is carried out as follows: the client computes $X = \mathfrak{h}(e(Token + [g]C, S))$, where g is a notion of the forgotten PIN, and sends its identity and X to the server; afterwards, the client proves its identity to the server by answering

a question, whose answer only the client knows. After that, the server goes offline and computes the value $Y = \mathfrak{h}(e(C, S_s - [i]S))$ for all possible values of i until it obtains a Y value such that $X = Y$; then, the server sends i to the client, which is the difference between the correct PIN α and g; finally, the client computes its forgotten PIN as $\alpha = i + g$.

Note that, the PIN α is never revealed to the server because he does not know the value g used.

Client		**Server**
$x \xleftarrow{\$} \mathbb{Z}_r$		$y, w \xleftarrow{\$} \mathbb{Z}_r$
Id_c	$\longleftrightarrow$	Id_s
$C \leftarrow \mathcal{H}_1(Id_c)$		$S \leftarrow \mathcal{H}_2(Id_s)$
$S \leftarrow \mathcal{H}_2(Id_s)$		$C \leftarrow \mathcal{H}_1(Id_c)$
$P_c \leftarrow [x]C$	$\longleftrightarrow$	$P_s \leftarrow [y]S,\ P_g \leftarrow [w]C$
$\pi_c \leftarrow \mathfrak{h}(P_c\|\|P_s\|\|P_g)$		$\pi_s \leftarrow \mathfrak{h}(P_s\|\|P_c\|\|P_g)$
$\pi_s \leftarrow \mathfrak{h}(P_s\|\|P_c\|\|P_g)$		$\pi_c \leftarrow \mathfrak{h}(P_c\|\|P_s\|\|P_g)$
$R \leftarrow Token + [PIN]C$		
$a \leftarrow [x + \pi_c]$		
$k \leftarrow e([a]R, [\pi_s]S + P_s)$		$k \leftarrow e([\pi_c]C + P_c, [y + \pi_s]S_s)$
$t \leftarrow \mathfrak{h}(k\|\|[x]P_g)$		$t \leftarrow \mathfrak{h}(k\|\|[w]P_c)$
	$K \leftarrow \mathfrak{h}(Id_c\|\|Id_s\|\|t)$	

Figure 7.1: Balanced two-factor authentication protocol [148].

Client		**Server**
$x, m \xleftarrow{\$} \mathbb{Z}_r$		$y, n \xleftarrow{\$} \mathbb{Z}_r$
$C \leftarrow \mathcal{H}_1(Id_c)$		
$R \leftarrow Token + [PIN]C$		
$Z \leftarrow [m]R,\ U \leftarrow [x]C$		$S \leftarrow \mathcal{H}_2(Id_s)$
$Id_c,\ U,\ Z$	$\longrightarrow$	$C \leftarrow \mathcal{H}_1(Id_c)$
		$t \leftarrow e([n]Z, S)$
	$\longleftarrow$	$y,\ t$
$V \leftarrow [-(x + y)]R$	$\longrightarrow$	$w \leftarrow e([n]V, S)$
$k \leftarrow t^{(x+y)/m}$		$k \leftarrow e([n](U + [y]C), S_s)$
		if $w \cdot k \neq 1_{\mathbb{G}_T}$ then fail!
$K \leftarrow \mathfrak{h}(k)$		otherwise $K \leftarrow \mathfrak{h}(k)$

Figure 7.2: Unbalanced two-factor authentication protocol [149].

7.3. Implementation

In this section we discuss aspects to take into a count when the protocols described in §7.2 are implemented in a secure way, such that it is possible to thwart some simple side-channel attacks.

7.3.1. Hash into the groups $\mathbb{G}_1$ and $\mathbb{G}_2$

Hash to $\mathbb{G}_1$ also known as the map-to-point primitive, was defined by Boneh and Franklin as $\mathcal{H}_1 : \{0,1\}^* \to \mathbb{G}_1$ in [22]. For this section purposes, map-to-point operation was implemented using the techniques presented in §6.1 for Barreto-Naehrig elliptic curves. Where, a standard hash function $\mathfrak{h}$ is used to project the identity $Id_{\{c,s\}}$ to an element $t \in \mathbb{F}_p^*$, and then we evaluate t in the deterministic coding proposed by Fouque and Tobouchi [65] shown in Algorithm 23 presented in §6.

The hash function into the group $\mathbb{G}_2$ consist of finding a point $Q \in \mathbb{G}_2$ starting whit a random point in $E'(\mathbb{F}_{p^2})$ as we saw in §6.1, also over Barreto-Naehrig curves. In order to implement this operation we used a standard hash function $\mathfrak{h}$ that project the identity $Id_{\{c,s\}}$ to an element $t \in \mathbb{F}_{p^2}^*$. Then we evaluate t as show in Algorithm 25 in order to obtain a random point Q' in $E'(\mathbb{F}_{p^2})$. Finally, using the Fuentes *et al.* [69] method we get the point $Q \in \mathbb{G}_2$.

7.3.2. Scalar Multiplication and modular exponentiation

In this section we present the methods used to perform the protected computation of scalar multiplication $Q = [k]P$ where $k = (k_{\ell-1}, \ldots, k_0)_2 \in \mathbb{Z}_r$ and the point P belongs to $\mathbb{G}_1$ or $\mathbb{G}_2$. Also we present the method used to implement the modular exponentiation $f^k \in \mathbb{G}_T$ with k as above.

7.3.2.1. Scalar multiplication in $\mathbb{G}_1$ and $\mathbb{G}_2$

In order to perform a scalar multiplication in the group $\mathbb{G}_1$ we used the method GLV introduced by Gallant *et al.* [72], which was detailed in §5.2.2.2. This method take advantage of the existence of an efficiently computable endomorphism $\psi(P) = [\lambda]P$ over the elliptic curve, for any order-r point P and some $\lambda \in \mathbb{Z}_r$. Then, the GLV method is used to split the scalar k into two sub-scalars k_1 and k_2 where the length of the sub-scalars is $\ell/2$ approximately. In that way, scalar multiplication $[k]P$ can be efficiently computed as $[k]P = [k_1]P + [k_2]\psi(P)$ using simultaneous scalar multiplication techniques.

On the other hand, the scalar multiplication in the group $\mathbb{G}_2$ was performed using the method proposed by Galbraith *et al.* in [70], which generalizes the GLV approach described above (see §5.2.2.3). This method takes advantage of the endomorphism in $\mathbb{G}_2$ defined as $\psi : \Psi \circ \pi^i \circ \Psi^{-1}$, where π^i represents the i-th application of Frobenius endomorphism and Ψ represents the isomorphism $\phi : E'(\mathbb{F}_{p^2}) \to E(\mathbb{F}_{p^{12}})$. The GLS technique allows to split the scalar into four sub-scalars k_1, k_2, k_3 and k_4 where the length of sub-scalars k_i is approximately $\ell/4$. In this way, the scalar multiplication $[k]P$ can be performed as $[k]P = [k_1]P + [k_2]\psi(P) + [k_3]\psi^2(P) + [k_4]\psi^3(P)$.

7.3.2.2. Modular exponentiation in $\mathbb{G}_T$

Exponentiation in the group $\mathbb{G}_T$ was performed using the method proposed by Galbraith *et al.* [70]. The endomorphism used in the group $\mathbb{G}_T$ is simply the Frobenius endomorphism, which allows computing the operation f^k as $f^k = f^{k_1} + f^{k_2 p} + f^{k_3 p^2} + f^{k_4 p^3}$. Were the length of sub-scalars k_i is approximately $\ell/4$.

7.3.2.3. Protection against side-channel attacks

Despite the computational improvement offered by the methods described above, those procedures do not offer protection against side-channel attacks. A countermeasure to this situation consists of applying an scalar recoding technique. Particularly, in this section we used the scalar recoding technique proposed by Joye and Tunstall in [101], where they

observed that any odd integer i in the interval $[0, 2^w)$ can be written as $i = 2^{w-1} + (-(2^{w-1} - 1))$. Then, dividing repeatedly an ℓ'-bit integer $n = n - ((n \mod 2^w) - 2^{w-1})$ by 2^{w-1} its parity remains and the obtained residues are in the set $\{\pm 1, \pm 3, \ldots, \pm 2^{w-1} - 1\}$ producing in this way a regular representation of length $\ell = 1 + \ell'/(w-1)$ with a non-zero digit density of $1/(w-1)$.

In addition, in order to achieve a regular execution of the scalar multiplication algorithms which is desirable to obtain a protected implementation against simple side-channel attacks, conditional sentences were avoided. The result of this procedure is shown in Algorithm 26 for the case of $\mathbb{G}_1$. Such algorithm can be easily generalized to the case of $\mathbb{G}_2$.

Algorithm 26 Scalar multiplication in $\mathbb{G}_1$ protected against side-channel attacks.

Input: $P \in \mathbb{G}_1$, the scalar $k \in \mathbb{Z}_r$ with $n = \lceil \log_2 k \rceil$, $\psi : (x, y) \mapsto (\beta \cdot x, y)$ and the window size w.

Output: $Q = [k]P \in \mathbb{G}_1$.

Precomputation:

1: Compute $R[j] = [i]P$ with $i \in \{1, \ldots, 2^{w-1} - 1\}$ and $0 \leq j < 2^{w-2}$.

Scalar regular recoding:

2: Decompose k as $k = (k_0 + \lambda k_1) \mod r$ using the GLV method.

3: $par_i \leftarrow (k_i[0] \& 0\text{x}01) \oplus 0\text{x}01$; $k_i \leftarrow k_i - par_i$ for $i = 0, 1$.

4: Convert k_i to the Joye and Tunstall [101] regular recoding of length ℓ for $i = 0, 1$.

Computation:

5: $Q \leftarrow R\left[n_0[\ell - 1 >> 1]\right] + \psi\left(-R\left[n_1[\ell - 1 >> 1]\right]\right)$

6: **for** $i \leftarrow \ell - 2$ **down to** 0 **do**

7: **for** $j \leftarrow 0$ **to** $w - 1$ **do**

8: $Q \leftarrow [2]Q$

9: **end for**

10: $s \leftarrow (n_0[i] >> 1)$; $t \leftarrow (n_0[i] \oplus s) - s$ ▷ s = sign of $n_0[i]$ and $t = abs(n_0[i])$.

11: $tmp = R[t >> 1]$

12: $tmp_{y_0} \leftarrow tmp[1]$; $tmp_{y_1} = -tmp[1]$ ▷ $tmp[1]$ is the y-coordinate of point tmp

13: $tmp[1] = tmp_{y_{s\&0\text{x}1}} + tmp_{y_{s\&0\text{x}1}} + tmp_{y_{s\&0\text{x}1}}$

14: $Q \leftarrow Q + tmp$

15: $s \leftarrow (n_1[i] >> 1)$; $t \leftarrow (n_1[i] \oplus s) - s$ ▷ s = sign of $n_1[i]$ and $t = abs(n_1[i])$.

16: $tmp \leftarrow \psi(R[t >> 1])$

17: $tmp_{y_0} \leftarrow tmp[1]$; $tmp_{y_1} = -tmp[1]$ ▷ $tmp[1]$ is the y-coordinate of point tmp

18: $tmp[1] = tmp_{y_{s\&0\text{x}1}} + tmp_{y_{s\&0\text{x}1}} + tmp_{y_{s\&0\text{x}1}}$

19: $Q \leftarrow Q + tmp$

20: **end for**

21: $tmp_0 \leftarrow Q$; $tmp_1 \leftarrow Q + P$; $Q \leftarrow tmp_{par_0}$

22: $tmp_0 \leftarrow Q$; $tmp_1 \leftarrow Q + \psi(-P)$; $Q \leftarrow tmp_{par_1}$

23: **return** Q

For our implementation, we used a value of w equal to 5, because in this way we obtained the best balance between speed and memory. Such choice of w produces a computational cost of $1D + 7A$ for pre-computation and $124D + 65A + 33M_\beta$ for scalar multiplication in $\mathbb{G}_1$, where M_β represents the multiplication by λ. On the other hand, for scalar multiplication in $\mathbb{G}_2$ and modular exponentiation in $\mathbb{G}_T$ the same techniques were implemented, producing a computational cost of $1D + 7A$ or $1S + 7M$ for pre-computation and $64D + 71A + 18\psi$ or $64D + 71A + 18\pi$ for evaluation of scalar multiplication or modular exponentiation, respectively. In all cases, we only store 8 pre-computed values.

7.3.3. Pairing computation

In this section we used the optimal Ate pairing shown in Algorithm 17, as was proposed by Aranha *et al.* in [6]. This pairing is defined over BN elliptic curves, which are perametrized as show in §5.1.2.1. For our implementation we used two different values for the x parameter that defines the prime p and the group order r (see Equations (5.5)). In the implementation of Figure 7.1 protocol we used $x_1 = -(2^{62} + 2^{55} + 1)$ that produces a Miller's loop (lines 3-8 of Algorithm 17) of length equal to $s = -(2^{64} + 2^{63} + 2^{57} + 2^{56} + 2^2)$. This parameter x_1 cannot be used to perform the protocol in Figure 7.2, because it produces a implementation vulnerable to the attack known as "small subgroup attack".

This attack takes advantage of the fact that in the field $\mathbb{F}_{p^{12}}$ there exist relatively few elements of order r. Therefore, the possible orders are the divisors of $p^{12} - 1 = (p^6 - 1)(p^2 + 1)((p^4 - p^2 + 1)/r)$. Using this information, the attack consists of impersonating the server, causing that the value k generated during the protocol has small order. In this way, the fake server can exhaustively search the correct value of k in order to successfully complete the protocol. For example, if we use x_1 the value $((p^4 - p^2 + 1)/r)$ can be factorized as $13 \cdot 3793 \cdot 29173 \cdot 716953 \cdot 569360689 \cdot C_{205}$, which allows that a fake server implement such attack using a value of k with order 13. For this reason, we decided to use $x_2 = -(2^{62} + 2^{47} + 2^{38} + 2^{37} + 2^{14} + 2^7 + 1)$ that produces a Miller's loop of length equal to $s = -(2^{64}+2^{63}+2^{49}+2^{48}+2^{41}+2^{38}+2^{16}+2^{15}+2^9+2^8+2^2)$ and assures that $((p^4-p^2+1)/r)$ is a prime number. This choice avoids the attack just described.

The exponentiation $f^{(p^{12}-1)/r}$ computed in step 12 of Algorithm 17, called final exponentiation, was implemented using the lattice basis reduction approach proposed by Fuentes *et al.* [69] with a computational cost of three Frobenius endomorphism applications , three exponentiations by x, twelve multiplications, 3 squarings and one inversion in $\mathbb{F}_{p^{12}}$. Such arithmetic operations were performed using towering technique where the finite field $\mathbb{F}_{p^{12}}$ is represented as:

$$\begin{aligned}
\mathbb{F}_{p^2} &= \mathbb{F}_p[u]/(u^2 - \beta), \text{ with } \beta = -1, \\
\mathbb{F}_{p^4} &= \mathbb{F}_{p^2}[V]/(V^2 - \xi), \\
\mathbb{F}_{p^6} &= \mathbb{F}_{p^2}[V]/(V^3 - \xi), \text{ with } \xi = u + 1, \\
\mathbb{F}_{p^{12}} &= \mathbb{F}_{p^6}[W]/(W^2 - \gamma), \text{ with } \gamma = V,
\end{aligned}$$

7.4. Results and conclusions

We implemented the protocols in Figures 7.1 and 7.2 following the techniques described above, using as client a Arandale development card equipped with an ARM Cortex-A15 processor at 1.7 GHz, and using as server a laptop with an Intel Core i7 processor at 2.4 GHz. All presented results were measured disabling the Turbo-Boost and Hyper-Threading technologies.

In Table 7.1 it is shown the associated cost to the operations of: point addition, point doubling, scalar multiplication in $\mathbb{G}_1$ and $\mathbb{G}_2$; hash functions $\mathcal{H}_1$ and $\mathcal{H}_2$; modular exponentiation in $\mathbb{G}_T$; and the operations that compose the optimal Ate pairing computation. All costs are given in thousand of clock cycles.

Finally, Table 7.2 shows the cost of the authentication process for both protocols. It can be seen that for the protocol in Figure 7.2, our implementation is superior in comparison with the author's implementation, although the comparison with the results of the author is a bit unfair since he reports his results using java script. On the other hand, for the protocol in Figure 7.1 we consider that our results are competitive taking into account that the author reports his results using an Intel core i5 processor at 2.4 GHz for both client and server.

In this work we show how to carry out efficient implementations of two pairing-based

Group	Operation	Method	Client		Server	
			x_1	x_2	x_1	x_2
$\mathbb{G}_1$	ADD	Mix. Coord.	4.25		1.08	
	DBL	Jac. Coord.	3.21		0.84	
	Scalar Mult.	GLV-NAF	775		207.910	
		GLV-Reg	832		208.254	
	$\mathcal{H}_1$	Alg. 23	1056.4	1366.2	124.62	158.29
$\mathbb{G}_2$	ADD	Mix. Coord.	12.1		3.05	
	DBL	Jac. Coord.	7.79		2.02	
	Scalar Mult.	GLS-NAF	1,511		419.597	
		GLS-Reg	1,712		425.707	
	$\mathcal{H}_2$	Alg. 25	2,099.9	2,530.2	344.82	408.24
$\mathbb{G}_T$	Exponentiation	GS	3,485		790.660	
Pairing	Miller loop		3,214	3,423	720.43	769.13
	Final exp.	Fuentes [69]	2,238	2,245	458.39	561.73
	Optimal Ate	Alg. 17	5,549	5,746	1,164	1,317

Table 7.1: Cost of main building blocks of the pairing-based protocols shown in §7.2.

Work	Protocol	Client	Server
Author	Figure 7.1	4.1 ms	4.4 ms
	Figure 7.2	3 seg.	"few" ms
Our	Figure 7.1	7.2 ms	7.3 ms
	Figure 7.2	4.2 ms	4.4 ms

Table 7.2: Cost of authentication protocols taking into acount the communication overhead.

two-factor authentication protocols, which are protected against time analysis attacks, over restricted platforms in terms of computational resources. In such way that it is possible to maintain a secure authentication scheme without sacrificing efficiency using a relatively low computational cost.

Chapter 8

Implementation of BLS signature protocol over curves with embedding degree one

In this section, we present how to efficiently implement the main building blocks found in the BLS digital signature algorithm, which was introduced in §5.2. Particularly, we focused in its implementation, using bilinear pairings defined over elliptic curves with embedding degree one, which does not have any of the speedups that come from working in a subfield during the pairing computation.

8.1. Introduction

Since the discovery of its constructive cryptographic properties, pairings have been used to design several cryptographic protocols. Because, they allow to solve in an elegant way the problem of the Identity Based Encryption, which was proposed by Shamir in 1984 [153]. Bilinear pairings are frequently constructed using elliptic curves with small embedding degrees, *i.e.* Using an elliptic curve E defined over a finite field $\mathbb{F}_p$, whose embedding degree k is the smaller positive integer such that $n \mid p^k - 1$ for a prime number r that divides $\#E(\mathbb{F}_p)$. A necessary condition for the security of the pairings, is that the Discrete Logarithm Problem must be intractable in the finite field $\mathbb{F}_p^k$. Until 2015, the assumption was that the fastest algorithm for computing logarithms in small-characteristic finite fields $\mathbb{F}_p$ was Coppersmith's algorithm [44] with a running time $O(\exp(1.52 \cdot (\log p^k)^{1/3} \cdot (\log \log p^k)^{2/3}))$, while the fastest algorithm for computing logarithms in large-characteristic finite fields $\mathbb{F}_p$ was the Number Field Sieve (NFS) [77] with running time $O(\exp(1.92 \cdot (\log p^k)^{1/3} \cdot (\log \log p^k)^{2/3}))$. However, in these year Kim-Barbulescu [109] presented in CRYPTO'16 the extended tower-NFS technique. In this technique if the field characteristic p has a medium-size and has an special form, which is the case for the popular pairings over BN curves, then the asymptotic complexity is $O(\exp(1.56 \cdot (\log p^k)^{1/3} \cdot (\log \log p^k)^{2/3}))$. From their experiments Kim and Barbulescu show that DLP in $\mathbb{F}_{p^2}$ is significantly easier than the DLP in $\mathbb{F}_p$. This analysis cast some suspicions to the intractability of the DLP in field extension $\mathbb{F}_{p^k}$ when p has an special form.

According with the work performed by Chatterjee, Menezes and Rodríquez-Henríquez [35] the aforementioned improvements in algorithms for computing discrete logarithms do not apply to the DLP in prime-order fields $\mathbb{F}_p$ provided that the prime p does not have a special

form. In this way, the fastest general purpose algorithm for the DLP in $\mathbb{F}_p$ has a complexity $O(\exp(1.92 \cdot (\log p^k)^{1/3} \cdot (\log\log p^k)^{2/3}))$. In consequence, elliptic curves with embedding degree one can be used to implement pairing based protocols. Because, the group $\mathbb{G}_T$ in these pairings is an order-r subgroup of $\mathbb{F}_p^*$. Hence, its security is not directly affected by the improvements using the algorithms for computing logarithms in extension fields.

In this section we present the implementation of the digital signature algorithm BLS, constructed over elliptic curves with embedding degree one. With the aim of offering a security level of 128-bits we use a prime number of 3072 bits.

8.2. Elliptic curves with embedding degree one

Elliptic curves with embedding degree one, which are the core of this section, can be constructed as follows:

Definition 8.1 (Elliptic curves with embedding degree one)**.** *Let p be a prime number computed as $p = A^2 + 1$ with $A = h \cdot r$, where r is a prime number and such that $r \equiv 3 \mod 4$. The elliptic curve*

$$E/\mathbb{F}_p : y^2 = x^3 + ax \tag{8.1}$$

wehre a can take the values -1 and -4 depending on the nature of A, i.e. if $A \equiv 0 \mod 4$ then $a = -1$ and if $A \equiv 2 \mod 4$ then $a = -4$. The ordinary elliptic curve E has trace 2 and the cardinality of the group $E(\mathbb{F}_p)$ is $p - 1$.

Theorem 8.1. *The elliptic curve group $E(\mathbb{F}_p)$ is isomorphic to $\mathbb{Z}/A\mathbb{Z} \oplus \mathbb{Z}/A\mathbb{Z}$. In addition, the map $\psi : (x, y) \mapsto (-x, Ay)$ is a distortion map on this group. In other words, if a $P \in E[r]/\mathcal{O}$ then $\psi(P) \notin \langle P \rangle$. Thus, for any $P \in E[r]/\mathcal{O}$, the pair of points $(P, \psi(P))$ generate $E[r]$.*

Pairing in these curves

Let $\mathbb{G}_1$ an arbitrary order-r subgroup of $E(\mathbb{F}_p)$ and let $\mathbb{G}_2 = E[r]$. Let $P \in \mathbb{G}_1$ and $Q \in \mathbb{G}_2$, then the Weil and Tate pairing are defined as follows:

$$e_W(P, Q) = (-1)^n \frac{f_{r,P}(Q)}{f_{r,Q}(P)},$$

that is degenerate if $Q \in \langle P \rangle$; and

$$e_T(P, Q) = (f_{r,P}(Q))^{(p-1)/r},$$

which is degenerate if $Q \in \langle \psi(P) \rangle$, respectively.

8.2.1. BLS signature algorithm for this pairings

Given as public parameters a bilinear pairing $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$, the generators G_1, G_2 of $\mathbb{G}_1$ and $\mathbb{G}_2$, and a hash function $\mathcal{H}_1 : \{0,1\}^* \to \mathbb{G}_2$. The secret key is a randomly selected element $x \in \mathbb{Z}/r\mathbb{Z}$, the public key is the group element $P = [x]G_1$. A signature on a message $m \in \{0,1\}^*$ is obtained as $S = [x]\mathcal{H}(m)$. While, any entity possessing the public key P can verify the signature simply checking whether $e(G_1, S) = e(P, \mathcal{H}(m))$.

8.2.2. Used constructions

We used three different methods to construct the characteristic p of the finite field $\mathbb{F}_p$, these methods are described below:

- **Construction A:** we use the prime number $r = 2^{256} + 2^{96} - 1$ such that $r \equiv 3 \mod 4$, as cofactor h we use a randomly generated 1280-bit number. The prime p generated satisfies $p \equiv 1 \mod 16$. This construction of p allows us to speedup the membership subgroup test; the scalar multiplication since the scalars has 256 bits of length; the Miller's algorithm because the loop length is r. However, this prime increase the cost of the hash function to the elliptic curve group $E(\mathbb{F}_p)[r]$ and the final exponentiation when the Tate pairing is used.
- **Construction B:** for this method we choose a 1536-bit prime number r and a cofactor $h = 2$. The prime p generated is such that $p \equiv 5 \mod 8$. This construction just allows us to speedup the computation of the hash function to the elliptic curve group $E(\mathbb{F}_p)[r]$.
- **Construction C:** we use the prime number $r = 2^{256} + 2^{96} - 1$, the cofactor h was randomly chosen such that the least significant half of the binary representation of h is equal to zero. In this way, the prime p generated using this method is a Montgomery-friendly prime. This, allows us to use the specialized reduction Algorithm 39, which improves the finite field arithmetic and therefore the overall cost of the protocol.

8.3. Finite field and elliptic curve arithmetic

This section is dedicated to show the main algorithms used to construct the finite field and elliptic curve arithmetic. Besides we show the implementation results of each of the operations in both arithmetics.

8.3.1. Finite field arithmetic

A fundamental part of the implementation of the BLS signature protocol is the efficient computation of the finite field arithmetic. For this purpose, we used the techniques presented in §3.3 to compute the multiplication and squaring over the integers. It is important to mention that, the size of the integers to be operated is 48 64-bit words. Then for multiplication and squaring an extra Karatsuba level is necessary. The modular reduction was performed using the Barrett algorithm for the constructions A and B, while the Montgomery reduction algorithm was used for the three constructions. We implemented both reduction algorithms in order to compare the costs and thus choose the one that gives the best performance. All the algorithms were implemented taking advantage of the assembly instructions described in §3.2.

8.3.1.1. Barrett modular reduction

Although the Barrett modular reduction algorithm is slightly more expensive than the Montgomery algorithm, in our work we explored to use this algorithm applying a further optimization known as folding technique. This technique was introduced in §3.3.4.2 and is presented in Algorithm 27.

This algorithm can be improved using the techniques described in §3.3.4.2 for the classical Barrett reduction algorithm. Thus, the multiplications in Steps 11 and 13 have a cost of $\frac{3n^2+\lambda n}{2\lambda^2}$ and $\frac{2n^2+n}{2\lambda} - \frac{n^2}{2\lambda^2} + 1$ word multiplications for $\lambda = 2^F$, respectively. In this way, the cost of the Algorithm 27 is $\sum_{i=1}^{F} n/2^i + \frac{n^2}{\lambda^2} + \frac{n^2+n}{\lambda} + 1$ word multiplications, taking into account the multiplications in the Step 9. For our purposes, we used $F = 2$, *i.e.* two folding steps, because experimentally this value give us a better performance.

Algorithm 27 Barrett reduction using the folding technique.

Input: A $2n$-word integer $c = (c_{2n-1}, \ldots, c_0)_r$, the n-word modulo $p = (p_n, \ldots, p_0)_r$ and the number of folding steps F.

Output: The n-word integer $c' = (c'_n, \ldots, c'_0)_r$ s.t. $c' = c \mod p$.

Precomputation:

$\mu \leftarrow \lfloor r^{(1+2^{-F})n}/p \rfloor$
for $i \leftarrow 1$ **to** F **do**
  $P_i \leftarrow r^{(1+2^{-i})n} \mod p$
end for

Computation:

$N_0 \leftarrow c$
for $i \leftarrow 1$ **to** F **do**
  $c'_1 \leftarrow N_{i-1} \mod r^{(1+2^{-i})n}$
  $c'_2 \leftarrow \lfloor N_{i-1}/r^{(1+2^{-i})n} \rfloor$
  $N_i \leftarrow c'_1 + c'_2 \cdot P_i$
end for
$\hat{q} \leftarrow \lfloor \lfloor N_F/r^n \rfloor \cdot \mu / r^{2^{-F}n} \rfloor$
$c'_1 \leftarrow N_F \mod r^{n+1}$
$c'_2 \leftarrow (\hat{q} \cdot p) \mod r^{n+1}$
$c' \leftarrow c'_1 - c'_2$
while $c' \geq p$ **do**
  $c' \leftarrow c' - p$
end while
return c'

8.3.1.2. Exponentiation

Exponentiation was constructed using the sliding window exponentiation method, which consist of decomposing the exponent into non-zero windows with maximum length ω bits, and variable length zero windows. This method reduces the number of multiplications at the extra cost of some pre-computations, Algorithm 28 shown the sliding window exponentiation. In our work, $\omega = 4$ achieve the best performance of this algorithm and just stores seven values as pre-computation.

8.3.1.3. Square root

The way to compute an square root depends on the nature of the prime p [4]. In the case of the construction B, the prime p generated is such that $p \equiv 5 \mod 8$. For this type of prime the best algorithm to compute the square root is the proposed by Atkin [7], presented in Algorithm 29, which has a cost of one exponentiation, two squarings and and 5 multiplications in $\mathbb{F}_p$.

On the other hand, the constructions A and C generate primes p such that $p \equiv 1 \mod 16$. For this type of primes there is not specialized method to compute the square root. Then it is necessary to use a costly generic algorithm. There are two generic algorithms to perform this operation the Tonelli-Shanks [154, 160] and the Müller [134] algorithm. The efficiency of the Tonelli-Shanks algorithm depends on the value s obtained by write $p - 1 = 2^s t$, where t is an odd integer. Therefore, this method is useful just for small values of s. For our purposes the value $s = n/2$, where n is the size in bits of p. Consequently, the Tonelli-Shanks becomes inefficient. For this reason we adopt the Müller algorithm, which computes the square root using a Lucas sequence. This method is presented in Algorithm 30 and has a cost of three quadratic residuosity tests χ_p, one evaluation of the Lucas sequence, one

Algorithm 28 Sliding window modular exponentiation.

Input: $x \in \mathbb{F}_p$, an exponent $e = (e_{n-1}, \ldots, e_0)_2$ and a widow size $\omega \geq 1$.
Output: $c = x^e \mod p$.

Precomputation:
1: Compute the values $x^3, x^5, \ldots, x^{2^\omega - 1}$.
Computation:
2: $c \leftarrow 1$; $i \leftarrow n-1$
3: **while** $i \geq 0$ **do**
4: **if** $e_i = 0$ **then**
5: $c \leftarrow c^2$; $i \leftarrow i-1$
6: **else**
7: $s \leftarrow \max\{i - \omega + 1, 0\}$
8: **while** $e_s = 0$ **do**
9: $s \leftarrow s + 1$
10: **end while**
11: $u \leftarrow 0$
12: **for** $h \leftarrow i$ **to** $i - s + 1$ **do**
13: $c \leftarrow c^2$ $u \leftarrow 2u + e_h$
14: **end for**
15: $c \leftarrow c \cdot x^u$ $i \leftarrow s - 1$ ▷ u is always odd and x^u was pre-computed.
16: **end if**
17: **end while**
18: **return** c

Algorithm 29 Atkin's square root algorithm for $p \equiv 5 \mod 8$ [4].

Input: An element $a \in \mathbb{F}_p^*$.
Output: c satisfying $c^2 = a$ if it exists and false otherwise.

Precomputation:
1: $t \leftarrow 2^{\frac{p-5}{8}}$
Computation:
2: $a_1 \leftarrow a^{\frac{p-5}{8}}$
3: $(a_1^2 a)^2$
4: **if** $a_0 = -1$ **then**
5: **return** false
6: **end if**
7: $b \leftarrow t a_1$
8: $i \leftarrow 2(ab)b$
9: $c \leftarrow ab(i-1)$
10: **return** c

inversion, two multiplications and two squarings in $\mathbb{F}_p$.

8.3.1.4. Inversion

Depending on the used modular reduction algorithm, the multiplicative inverse of an element in $\mathbb{F}_p$ is computed using the binary extended Euclidean algorithm [82, Algorithm 2.22, p. 41] when the modular arithmetic uses the Barrett reduction algorithm; or the Montgomery inversion algorithm as was defined by Kaliski in [104] if the Montgomery reduction algorithm was used.

Algorithm 30 Müller's square root algorithm for $p \equiv 1 \mod 16$ [4].

Input: $a \in \mathbb{F}_p^*$.
Output: c satisfying $c^2 = a$ if it exists and false otherwise.

1: $t \leftarrow 1$
2: $a_1 \leftarrow \chi_p(a-4)$
3: **while** $a_1 = 1$ **do**
4: Select randomly $t \in \mathbb{F}_p^* \setminus \{1\}$
5: **if** $at^2 - 4 = 0$ **then**
6: **return** $2t^{-1}$
7: **end if**
8: $a_1 \leftarrow \chi_p(at^2-4)$
9: **end while**
10: $\alpha \leftarrow at^2 - 2$
11: $c \leftarrow V_{\frac{p-1}{4}}(\alpha, 1)/t$
12: $a_0 \leftarrow c^2 - a$
13: **if** $a_0 \neq 0$ **then**
14: **return** false
15: **end if**
16: **return** c

8.3.1.5. Implementation results of finite field arithmetic

In this section we show the timings obtained from the implementation of the finite field operations described in previous sections. As was mentioned before, we used Barrett based arithmetic just for the constructions A and B, and Montgomery based arithmetic for all the three constructions. From Table 8.1 we can observe that Montgomery based arithmetic achieves a better performance against the Barrett based arithmetic using the folding technique. Therefore, in the following, only the costs corresponding to Montgomery based arithmetic will be presented.

Op.	**Haswell**			**Skylake**		
	A	B	C	A	B	C
Mul	10.3 (10.4)	10.4 (10.5)	6.8	8.7	8.7	6.1
Sqr	9.7 (9.8)	9.7 (9.8)	6.1	8.3	8.1	5.4
Exp	32,804 (33,521)	17,975 (18,332)	19,075	27,916	15,363	17,282
Inv	1,991 (2,802)	1,997 (2,053)	1,399	1,889	1,903	1,301
Sqrt	64,771 (65,332)	35,768 (36,459)	43,695	56,196	30,554	39,383

Table 8.1: Timings of the finite field operations for the different construccions of p. The timings were measured in thousand of clock cycles on micro-architectures Intel Haswell and Skylake. The parentesis costs corresponds to the aritmetic based on the Barrett reduction.

8.3.2. Elliptic curve arithmetic

In this section we present the algorithms used to compute a point addition, point doubling and the scalar multiplication. Besides, we show the associated cost to these algorithms using M, S and I to represent the operations of multiplication, squaring and inversion in $\mathbb{F}_p$, respectively.

8.3.2.1. Point addition and point doubling

With the aim of avoid costly inversions in $\mathbb{F}_p$, we represent the points in $E(\mathbb{F}_p)$ using modified Jacobian coordinates [43, 40], where a point $(X, Y, Z, W = Z^2)$ corresponds to the point (x, y) in affine coordinates with $x = X/Z^2$ y $y = Y/Z^3$. Furthermore, in order to have a better performance we used mixed coordinates for the point addition. Thus the mixed point addition in Algorithm 31 has a cost of 7M and 4S en $\mathbb{F}_p$. On the other hand, point doubling was performed using modified Jacobian coordinates as shown in Algorithm 32 at a cost of 1M and 8S.

Algorithm 31 Mixed point addition (Affine-Jacobian coordinates).

Input: The points $P = (X_P, Y_P, Z_P, W_P)$ in modified Jacobian coordinates and $Q = (x_Q, y_Q)$ in affine coordinates.
Output: $R = P + Q = (X_R, Y_R, Z_R, W_R)$ in modified Jacobian coordinates.

1: $t_1 \leftarrow x_Q \cdot W_P$
2: $t_2 \leftarrow y_Q \cdot Z_P \cdot W_P$
3: $t_3 \leftarrow t_1 - X_P$
4: $t_4 \leftarrow t_3^2$
5: $t_5 \leftarrow 4t_4$
6: $t_6 \leftarrow t_3 \cdot t_5$
7: $t_7 \leftarrow t_2 - Y_P$
8: $t_8 \leftarrow 2t_7$
9: $t_9 \leftarrow t_7^2$
10: $t_{10} \leftarrow 4t_9$
11: $t_{11} \leftarrow X_P \cdot t_5$
12: $X_R \leftarrow t_{10} - t_6 - 2t_{11}$
13: $Y_R \leftarrow t_8 \cdot (t_{11} - X_R) - 2(Y_P \cdot t_6)$
14: $Z_R \leftarrow (Z_P + t_3)^2 - W_P - t_4$
15: $W_R \leftarrow Z_R^2$
16: **return** $R = (X_R, Y_R, Z_R, W_R)$

Algorithm 32 Point doubling (Jacobian coordinates).

Input: The point $P = (X_P, Y_P, Z_P, W_P)$ in modified Jacobian coordinates and the coefficient a of the curve.
Output: $Q = 2P = (X_Q, Y_Q, Z_Q, W_Q)$ in modified Jacobian coordinates.

1: $t_1 \leftarrow X_P^2$
2: $t_2 \leftarrow Y_P^2$
3: $t_3 \leftarrow t_2^2$
4: $t_4 \leftarrow W_P^2$
5: $t_5 \leftarrow 2((X_P + t_2)^2 - t_1 - t_3)$
6: $t_6 \leftarrow 3t_1 + at_4$
7: $X_Q \leftarrow t_6^2 - 2t_5$
8: $Y_Q \leftarrow t_6 \cdot (t_5 - X_Q) - 8t_3$
9: $Z_Q \leftarrow (Y_P + Z_P)^2 - t_2 - W_P$
10: $W_Q \leftarrow Z_Q^2$
11: **return** $Q = (X_Q, Y_Q, Z_Q, W_Q)$

8.3.2.2. Scalar multiplication

This operation was performed using the ω-NAF scalar multiplication method described in §5.2.2.1. Given the cost of the Algorithm 18 presented in that section, the scalar multiplication used in this work has a cost of $310A + 1537D$ when $\log(r) = 1536$ and $54A + 257D$ when $\log(r) = 256$, using in both cases a window $\omega = 4$.

8.3.2.3. Implementation results of elliptic curve arithmetic

In Table 8.2 we show the timings for the elliptic curve arithmetic. The scalars used to compute the scalar multiplication are presented below:

- **Construction A y C:** in both constructions we used a 256-bit integer r with a Hamming weight of 3, and a 1280-bit integer h. Although, for construction C the binary representation of the integer h has the less significant half in zero.
- **Construction B:** here we used a 1536-bit integer r with a Hamming weight of approximately 768 bits, and $h = 2$.

Operations	Haswell			Skylake		
	A	B	C	A	B	C
Addition	113.1	112.2	74.0	95.8	95.3	66.3
Doubling	90.3	90.1	59.0	75.5	76.1	52.3
SM by r	23,180	171,426	15,203	19,667	145,913	13,477
SM by h	143,556	90.2	85,410	121,964	76.3	75,321
SM	39,735	172,742	26,358	34,012	145,692	23,541

Table 8.2: Timings for the elliptic curve arithmetic. The timings are measured in thousand of clock cycles.

8.4. Main building blocks of the BLS protocol

In this section we show the building blocks necessary to construct the BLS signature protocol described in §8.2.1. As we can observe, in the protocol is necessary to compute a scalar multiplication, a pairing and a hash function to the group $E[r]$. It is also necessary to verify that the point corresponding to the signature belongs to the correct subgroup. This operation is called the subgroup membership test.

8.4.1. Pairing

The pairing computation is performed using the Weil and the Tate pairing described in §5.2.1.1 and §5.2.1.2, respectively. The core of these pairings is the Miller Algorithm whose efficiency depends on the addition step (lines 7-9 of Algorithm 16) and the doubling step (lines 3-5 of Algorithm 16). These two operations were performed using the formulas proposed by Hu *et al.* [88] in Jacobian coordinates, which have a cost of 12M + 5S and 8M + 10S for the addition step presented in Algorithm 33 and the doubling step shown in Algorithm 34 respectively.

In this way, the Miller Algorithm has a cost of $(n-1)(8\text{M}+10\text{S})+(m-1)(12\text{M}+5\text{S})+2\text{M}+\text{I}$ for a n-bit integer r with a Hamming weight m. The costs of the Miller Algorithm for the different constructions of p used in this work are presented below:

Algorithm 33 Addition step in Miller's loop.

Input: The points $T = (X_T, Y_T, Z_T, W_T)$ in modified Jacobian coordinates, $P = (x_P, y_P)$ and $Q = (x_Q, y_Q)$ in affine coordinates and the Miller values $f, g \in \mathbb{F}_p$.

Output: $T = T + P = (X_T, Y_T, Z_T, W_T)$ in modified Jacobian coordinates and updates the values f, g.

1: $t_1 \leftarrow x_P \cdot W_T$
2: $t_2 \leftarrow y_P \cdot Z_T \cdot W_T$
3: $t_3 \leftarrow t_1 - X_T$
4: $t_4 \leftarrow t_3^2$
5: $t_5 \leftarrow 4t_4$
6: $t_6 \leftarrow t_3 \cdot t_5$
7: $t_7 \leftarrow t_2 - Y_T$
8: $t_8 \leftarrow 2t_7$
9: $t_9 \leftarrow t_7^2$
10: $t_{10} \leftarrow 4t_9$
11: $t_{11} \leftarrow X_T \cdot t_5$
12: $X_T \leftarrow t_{10} - t_6 - 2t_{11}$; $Y_T \leftarrow t_8 \cdot (t_{11} - X_T) - 2(Y_T \cdot t_6)$
13: $Z_T \leftarrow (Z_T + t_3)^2 - W_T - t_4$; $W_T \leftarrow Z_T^2$
14: $t_{12} \leftarrow (t_7 + Z_T)^2 - t_9 - W_T$
15: $l \leftarrow W_T \cdot (y_Q - y_P) - t_{12} \cdot (x_Q - x_P)$ $v \leftarrow W_T \cdot x_Q - X_T$
16: $f \leftarrow f \cdot l$; $g \leftarrow g \cdot v$
17: **return** $T = (X_T, Y_T, Z_T, W_T), f, g$

Algorithm 34 Doubling step in Miller's loop.

Input: The points $T = (X_T, Y_T, Z_T, W_T)$ in modified Jacobian coordinates, $Q = (x_Q, y_Q)$ in affine coordinates, the Miller values $f, g \in \mathbb{F}_p$ and the coefficient a of the curve.

Output: $T = 2T = (X_T, Y_T, Z_T, W_T)$ in modified Jacobian coordinates and updates the values f, g.

1: $t_1 \leftarrow X_T^2$
2: $t_2 \leftarrow Y_T^2$
3: $t_3 \leftarrow t_2^2$
4: $t_4 \leftarrow W_T^2$
5: $t_5 \leftarrow 2((X_T + t_2)^2 - t_1 - t_3)$
6: $t_6 \leftarrow 3t_1 + at_4$
7: $X_T \leftarrow t_6^2 - 2t_5$; $Y_T \leftarrow t_6 \cdot (t_5 - X_T) - 8t_3$
8: $Z_T \leftarrow (Y_T + Z_T)^2 - t_2 - W_T$; $W_T \leftarrow Z_T^2$
9: $t_7 \leftarrow W_T \cdot x_Q - X_T$
10: $l \leftarrow Z_T \cdot W_T \cdot y_Q + Y_T - t_6 \cdot t_7$
11: $v \leftarrow Z_T \cdot t_7$
12: $f \leftarrow f^2 \cdot l$; $g \leftarrow g^2 \cdot v$
13: **return** $T = (X_T, Y_T, Z_T, W_T), f, g$

- **Construction A y C:** we used a 256-bit integer r with Hamming weight 3, thus, the Miller algorithm has a cost of 2066M + 2560S + I.
- **Construction B:** using a 1536-bit integer r with a Hamming weight of approximately 768 bits the Miller algorithm has a cost of 21486M + 19185S + I.

Therefore, given that the Weil pairing is computed using two applications of the Miller algorithm, one multiplication and one inversion in $\mathbb{F}_p$, the total cost of the Weil pairing for the constructions A and C is 4133M + 5120S + 3I. While, for the construction B this pairing

has a cost of 42973M + 38370S + 3I. On the other hand, the cost of the Tate pairing is the cost of one application of the Miller algorithm plus one exponentiation by $(p-1)/r$.

8.4.2. Hash function to elliptic curve subgroup

This operation was performed using the algorithm proposed by Boneh, Lynn y Shacham [24], called *try-and-increment*. This algorithm was presented in §5.2.3.2 and basically consist in given the equation of the curve $E/\mathbb{F}_p : x^2 + ax$ pick randomly an element $x \in \mathbb{F}_p$, check whether $t = x^3 + ax$ is a square, then if so, set $y = \pm t$ and return the point $[h](x, y) \in E[r]$. If t is not a square, then one can pick another x and try again. The cost of this procedure is dominated by the cost of the square root and the scalar multiplication by h.

8.4.3. Subgroup membership testing

Given $P, Q \in E[r]/\mathcal{O}$. The subgroup membership testing consist in testing whether $Q \in \langle P \rangle$. According to the Weil pairing properties the point $Q \in \langle P \rangle$ if and only if $e_W(P, Q) = 1$. On the other hand for the Tate pairing is no guaranteed that $e_T(P, P) = 1$, however if E is an trace-2 elliptic curve then the point $Q \in \langle P \rangle$ if and only if $e_T(P, \psi(Q)) = 1$ [35]. Thus, the cost of the subgroup membership testing is of one pairing computation.

8.5. Results and conclusions

In this section we present the results obtained from our software library for the implementation of the BLS signature algorithm, using the algorithms and techniques shown in the previous sections. All the timings were measured in the processors Intel i7-4700 at 2.4 GHz. with Haswell micro-architecture and Intel i7-6700 at 3.0 GHz. with micro-architecture Skylake. Both with the Turbo-Boost and Hyper-Threading technologies disabled. Our library was compiled using `GCC` in its version 6.3 and using the compilation flag `-O3`.

In Table 8.3 it is reported the associated timings to the construction of the Miller's algorithm, Weil and Tate pairings, and the cost of the Hash function to elliptic curves. We can observe, as was expected, the Tate pairing achieves a better performance than the Weil pairing by a factor of ×1.2 for constructions A and C and a factor ×1.9 for construction C.

Operations	**Haswell**			**Skylake**		
	A	B	C	A	B	C
Addition Step	0.175	0.174	0.115	0.149	0.148	0.104
Doubling Step	0.181	0.181	0.119	0.153	0.153	0.107
Miller Algorithm 16	48.5	418.5	32.01	41.41	355.75	28.73
Weil pairing	98.64	840.08	65.33	84.76	713.53	58.90
Tate pairing	81.30	435.73	51.17	69.36	374.47	45.98
Hash function	210.3	37.9	130.5	180.1	32.5	116.0

Table 8.3: Timings for the main bulding blocks that compose the BLS protocol. The timings are presented in millions of clock cycles.

Table 8.4 shown the costs in millions of clock cycles of the signature and verification phases of the BLS signature protocol. The verification phase was implemented using both Weil and Tate pairings. However as it was observed before, the Tate pairing offers the best performance. The best results for the BLS protocol are obtained with the construction C, but an analysis of the security of this construction is necessary to be sure of thwart application

of special NFS algorithm. If it turns out that construction C is vulnerable, then construction A is the best option.

Operations	**Haswell**			**Skylake**		
	A	B	C	A	B	C
Signature	250.1	210.6	156.9	214.1	178.2	139.5
Verification (Weil)	512.1	2,325.2	327.6	438.6	1,980.0	293.3
Verification (Tate)	477.5	1,516.5	299.3	407.8	1,301.9	267.4

Table 8.4: Timing of the BLS signature and verification phases. The timings are presented in millions of clock cycles.

Part III

Isogeny-based cryptography

Chapter 9

Introduction to the supersingular isogeny Diffie-Hellman protocol

The hypothetical existence of a sufficiently powerful quantum computer would mean that currently used public key cryptography is completely insecure. This is so because, an implementation of Shor's algorithm in such computer would easily break the cryptosystems based on the Discrete Logarithm Problem and integer factorization. As a reaction to this situation the cryptographic community has proposed alternative problems that could resist a quantum attacker, one of such proposals is the hardness of finding an isogeny map between supersingular elliptic curves which is known as isogeny-based cryptography.

The supersingular isogeny Diffie-Hellman (SIDH) [95] key exchange protocol has positioned itself as a promising candidate for post-quantum cryptography. One salient feature of the SIDH protocol is that it requires exceptionally short key sizes. However, the latency associated to SIDH is higher than the ones reported for other post-quantum cryptosystem proposals. Aiming to accelerate the SIDH runtime performance, in the following chapters several algorithmic optimizations targeting finite field and elliptic curve arithmetic, and protocol operations are presented.

9.1. Isogenies

Isogenies are homomorphisms between elliptic curves, which play a fundamental role in the theory of elliptic curves and cryptography since they allow us to relate one elliptic curve to another.

Definition 9.1 (Isogeny)**.** *Let E_1 and E_2 be elliptic curves over a finite field $\mathbb{F}_q$. An isogeny from E_1 to E_2 is a non-constant homomorphism $\phi : E_1(\mathbb{F}_q) \to E_2(\mathbb{F}_q)$ that is given by rational functions defined over $\mathbb{F}_q$. This means that $\phi(P+Q) = \phi(P) + \phi(Q)$ for all $P, Q \in E_1(\mathbb{F}_q)$.*

Proposition 9.1. *Two elliptic curves E_1 and E_2 defined over $\mathbb{F}_q$ are said to be isogenous over $\mathbb{F}_q$ if there exists an isogeny $\phi : E_1 \to E_2$ defined over $\mathbb{F}_q$. Besides, two curves E_1 and E_2 are isogenous over $\mathbb{F}_q$ if and only if $\#E_1(\mathbb{F}_q) = \#E_2(\mathbb{F}_q)$.*

From Definition 9.1 we know that there are rational functions r_1 and r_2 such that if $\phi(x_1, y_1) = (x_2, y_2)$ then $(x_2, y_2) = (r_1(x_1), y_1 r_2(x_1))$, for $(x_1, y_1) \in E_1(\mathbb{F}_q)$ and $(x_2, y_2) \in E_2(\mathbb{F}_q)$. Moreover, if we write $r_1(x) = s(x)/t(x)$ using polynomials $s(x)$, $t(x)$ that do not have a common factor. The **degree of the isogeny** ϕ is defined as

$$\deg(\phi) = max\{\deg s(x), \deg t(x)\}.$$

If the derivative $r_1'(x)$ is not identically 0, we say that ϕ **is separable**.

Proposition 9.2. *Let $\phi : E_1 \to E_2$ be an isogeny. If ϕ is separable, then*

$$\deg \phi = \#\ker(\phi).$$

If ϕ is not separable, then

$$\deg \phi > \#\ker(\phi).$$

In particular, the kernel of an isogeny is a finite subgroup of $E_1(\mathbb{F}_q)$.

The following proposition tells us that an elliptic curve isogenous to an elliptic curve E is essentially uniquely determined by the kernel of the isogeny to it.

Proposition 9.3. *Let E_1, E_2 and E_3 be elliptic curves over a finite field $\mathbb{F}_q$ and suppose that there exist separable isogenies $\phi_2 : E_1 \to E_2$ and $\phi_3 : E_1 \to E_3$ defined over $\mathbb{F}_q$. If $\ker \phi_2 = \ker \phi_3$, then E_2 is isomorphic to E_3 over $\mathbb{F}_q$. In fact, there is an isomorphism $\beta : E_2 \to E_3$ such that $\beta \circ \phi_2 = \phi_3$.*

A very important property of isogenies is the existence of dual isogenies.

Theorem 9.1. *Let $\phi : E_1 \to E_2$ be an isogeny of elliptic curves. Then, there exists a dual isogeny $\widehat{\phi} : E_2 \to E_1$ such that $\widehat{\phi} \circ \phi$ is multiplication by $\deg \phi$ on E_1.*

Since every isogeny has a dual isogeny, the property of being isogenous over $\mathbb{F}_q$ is an equivalence relation on the finite set of $\overline{\mathbb{F}}_q$-isomorphism classes of elliptic curves defined over $\mathbb{F}_q$. Accordingly, we define an isogeny class to be an equivalence class of elliptic curves, taken up to $\overline{\mathbb{F}}_q$-isomorphism, under this equivalence relation. Curves in the same isogeny class are either all supersingular or all ordinary. In this chapter we assume that we are in the supersingular case.

Proposition 9.4. *Supersingular curves are all defined over $\mathbb{F}_{p^2}$, and for every prime $\ell \nmid p$, there exist $\ell + 1$ isogenies counting multiplicities of degree ℓ originating from any given such supersingular curve.*

Due to the work of Velú [161], given an elliptic curve E and a subgroup H of $E(\mathbb{F}_q)$ it is possible to construct an isogeny. Velú provided us with a formula that can be used to find the isogeny ϕ and the isogenous curve E/H. In the following sections we present the explicit formulas to compute an isogeny for the Montgomery and Edwards elliptic curve models.

9.2. Elliptic curve models

There exist diverse forms of elliptic curves. However, in this chapter we focus on Montgomery and twisted Edwards curves defined over the field $\mathbb{F}_q$ where $q = p^2$.

9.2.1. Montgomery curves and their arithmetic

Given a finite field $\mathbb{F}_q$, Montgomery elliptic curves are defined by the equation:

$$E_{A,B}/\mathbb{F}_q : \quad By^2 = x^3 + Ax^2 + x\,, \tag{9.1}$$

such that $A, B \in \mathbb{F}_q$, $A^2 \neq 4$ and $B \neq 0$. The j-invariant of $E_{A,B}$ is calculated as,

$$j(E) = 256\frac{(A^2-3)^3}{A^2-4}\,.$$

Moving to projective coordinates ($\mathbb{P}^2$) implies that an affine point (x, y) is represented by $(\lambda X : \lambda Y : \lambda Z)$ such that $\lambda \neq 0$, $x = X/Z$ and $y = Y/Z$. The point at infinity is a special case that is written as $\mathcal{O} = (0 : 1 : 0)$. Points can also be mapped to $\mathbb{P}^1$ using[1]

$$\begin{array}{rccl} \mathbf{x}: & \mathbb{P}^2 & \to & \mathbb{P}^1 \\ (X : Y : Z) & \mapsto & & (X : Z) \\ \mathcal{O} & \mapsto & & (1 : 0)\,. \end{array} \tag{9.2}$$

We write $\mathbf{x}(P) = (X : Z) \in \mathbb{P}^1(\mathbb{F}_p)$ when both X and Z belong to the field $\mathbb{F}_p$.

In his landmark paper [132], Montgomery introduced the concept of a *differential* addition operation, which given $\mathbf{x}(P)$, $\mathbf{x}(Q)$, and $\mathbf{x}(P - Q)$, calculates $\mathbf{x}(P + Q)$. It is noticed that this formula fails whenever $P - Q \in \{\mathcal{O}, (0, 0)\}$.

Let $P, Q \in E_{A,B}(\mathbb{F}_q)$ and $R_0, R_1, R_2 \in \mathbb{P}^1$. We denote a point doubling operation as $[2]R_0$ and can be computed by

$$\begin{aligned} 4X_{R_0}Z_{R_0} &= (X_{R_0} + Z_{R_0})^2 - (X_{R_0} - Z_{R_0})^2, \\ X_{[2]R_0} &= (X_{R_0} + Z_{R_0})^2(X_{R_0} - Z_{R_0})^2, \\ Z_{[2]R_0} &= (4X_{R_0}Z_{R_0})((X_{R_0} - Z_{R_0})^2 + ((A + 2)/4)(4X_{R_0}Z_{R_0})). \end{aligned}$$

This operation has a cost of 2M+2S [52] where M corresponds to the cost of a multiplication and S represent a squaring in the finite field.

On the other hand, a differential addition denoted as $R_0 +_{(R_2)} R_1$, such that $R_0 = \mathbf{x}(P)$, $R_1 = \mathbf{x}(Q)$, and $R_2 = \mathbf{x}(P - Q)$ can be computed as follows

$$\begin{aligned} X_{R_0+_{(R_2)}R_1} &= Z_{R_2}((X_{R_0} - Z_{R_0})(X_{R_1} + Z_{R_1}) + (X_{R_0} + Z_{R_0})(X_{R_1} - Z_{R_1}))^2, \\ Z_{R_0+_{(R_2)}R_1} &= X_{R_2}((X_{R_0} - Z_{R_0})(X_{R_1} - Z_{R_1}) + (X_{R_0} + Z_{R_0})(X_{R_1} - Z_{R_1}))^2. \end{aligned}$$

at a cost of 4M+2S. When performing a differential addition one multiplication can be saved whenever $Z_{P-Q} = 1$ [52].

Montgomery also introduced in [132] a procedure that calculates $\mathbf{x}([k]P)$ from $\mathbf{x}(P)$ and an integer k. This procedure is better known as the Montgomery ladder. A high level description of the Montgomery ladder is shown in Algorithm 35. To recover the y-coordinate of $[k]P$ one can use the Okeya-Sakurai technique [136], which extends the y-recovery formula of López-Dahab that applies to the binary elliptic curve case [121]. The Okeya-Sakurai formula calculates the y-coordinate of $[k]P$ from the y-coordinate of P, $\mathbf{x}([k]P)$ and $\mathbf{x}([k+1]P)$ (this latter value is also computed by the Montgomery ladder).

The Montgomery ladder processes the scalar k from the most significant to the least significant bit updating at each iteration the accumulators R_0 and R_1. The bits of the scalar determine which of the accumulators is updated by the results of the point doubling or the differential addition operations. Each step of the ladder performs the same number of operations preserving the relation $R_0 - R_1 = P$. This is an advantageous property, since usually the scalar k is a secret value. Therefore, a regular execution pattern helps to prevent threats caused by some simple side-channel attacks. Let k be a t-bit number. Then Algorithm 35 takes $5t$M+$4t$S, which in practice translates to a 7.6 M-per-bit cost, under the assumption that $1S \approx 0.66M$ in $\mathbb{F}_q$.

Performing all elliptic curve operations in $\mathbb{P}^1$ minimizes the use of multiplicative inverses, which tend to be quite costly. Hence, it is desirable to perform the scalar multiplication computations required by the SIDH using this strategy.

[1] This map sets $\mathcal{O} = (1 : 0)$ since $(0 : 0) \notin \mathbb{P}^1$, see [52, §3].

Algorithm 35 Montgomery ladder algorithm.

Input: $(k, \mathbf{x}(P))$, where k is a t-bit number, and $\mathbf{x}(P) \in \mathbb{P}^1$ is a representation of $P \in E_{A,B}(\mathbb{F}_q)$.
Output: $\mathbf{x}([k]P) \in \mathbb{P}^1$.

1: Initialize $R_0 \leftarrow \mathbf{x}(\mathcal{O})$, $R_1 \leftarrow \mathbf{x}(P)$, and $R_2 \leftarrow \mathbf{x}(P)$.
2: **for** $i \leftarrow t-1$ **to** 0 **do**
3: **if** $k_i = 1$ **then**
4: $(R_0, R_1) \leftarrow (R_0 +_{(R_2)} R_1, 2R_1)$
5: **else**
6: $(R_0, R_1) \leftarrow (2R_0, R_0 +_{(R_2)} R_1)$
7: **end if**
8: **end for**
9: **return** R_0 ▷ For y-coordinate recovery, return also R_1.

9.2.2. Edwards curves and their arithmetic

Introduced by H. Edwards in 2007 [122] this model of curves defined over a finite field $\mathbb{F}_q$ is defined by

$$E_d/\mathbb{F}_q : x^2 + y^2 = 1 + dx^2y^2,$$

with $d \neq 1$. Laater in 2008 Bernstein-Lange [20] proposed a generalized model called *Twisted Edwards curves* which are defined by the equation

$$E_{a,d}/\mathbb{F}_q : ax^2 + y^2 = 1 + dx^2y^2, \tag{9.3}$$

for two distinct non-zero elements of $\mathbb{F}_q$ a and $d \neq 1$. The j-invariant of $E_{a,d}$ is calculated as,

$$j(E) = \frac{16(a^2 + 14ad + d^2)^3}{ad(a-d)^4},.$$

In the same way as Montgomery curves, we can move to projective coordinates ($\mathbb{P}^2$) that implies that an affine point (x, y) is represented by $(\lambda X : \lambda Y : \lambda Z)$ such that $\lambda \neq 0$, $x = X/Z$ and $y = Y/Z$. The point at infinity is a special case that is written as $\mathcal{O} = (0 : 1 : 0)$. Points can also be mapped to $\mathbb{P}^1$ as was point out by Castrick, Galbraith and Farashahi [31]

$$\begin{array}{rcl} \mathbf{y}: \quad \mathbb{P}^2 & \to & \mathbb{P}^1 \\ (X : Y : Z) & \mapsto & (Y : Z) \\ \mathcal{O} & \mapsto & (1 : 0). \end{array} \tag{9.4}$$

Let $P, Q \in E_{a,d}(\mathbb{F}_q)$ and $R_0, R_1 \in \mathbb{P}^1$. The point doubling operation denoted as $[2]R_0$ can be computed by

$$\begin{aligned} Y_{[2]R_0} &= -c^2 dY_{R_0}^4 + 2Y_{R_0}^2 Z_{R_0}^2 - c^2 Z_{R_0}^4, \\ Z_{[2]R_0} &= dY_{R_0}^4 - 2c^2 dY_{R_0}^2 Z_{R_0}^2 + Z_{R_0}^4. \end{aligned}$$

This operation has a cost of 1M+4S [102].

On the other hand, a differential addition denoted as $R_0 +_{(R_2)} R_1$, such that $R_0 = \mathbf{y}(P)$, $R_1 = \mathbf{y}(Q)$, and $R_2 = \mathbf{y}(P - Q)$ can be computed as follows

$$\begin{aligned} Y_{R_0+_{(R_2)}R_1} &= Z_{R_2}(Y_{R_0}^2(Z_{R_1}^2 - c^2 dY_{R_1}^2) + Z_{R_0}^2(Y_{R_1}^2 - c^2 Z_{R_1}^2)), \\ Z_{R_0+_{(R_2)}R_1} &= Y_{R_2}(dY_{R_0}^2(Y_{R_1}^2 - c^2 Z_{R_1}^2) + Z_{R_0}^2(Z_{R_1}^2 - c^2 dY_{R_1}^2)). \end{aligned}$$

at a cost of 6M+4S [102].

9.2.3. Relation between Montgomery and Edwards curves

Twisted Edwards curves and Montgomery curves are strongly related in the sense that every twisted Edwards curve is birationally equivalent to a Montgomery curve over $\mathbb{F}_q$. This was proven in [20] and this equivalence is given via

$$\begin{aligned} \phi : E_{a,d} &\to E_{A,B} \\ (x, y) &\mapsto \left(\frac{1+y}{1-y}, \frac{1+y}{(1-yx)} \right), \end{aligned}$$

where $A := \frac{2(a+d)}{(a-d)}$ and $B := \frac{4}{a-d}$. Conversely

$$\begin{aligned} \psi : E_{A,B} &\to E_{a,d} \\ (x, y) &\mapsto \left(\frac{x}{y}, \frac{x-1}{x+1} \right), \end{aligned}$$

where $a := \frac{A+2}{B}$ and $d := \frac{A-2}{B}$.

In projective coordinates, particularly for the Montgomery XZ-only-coordinates and the twisted Edwards YZ-only-coordinates these maps become remarkably simple. A point $(X_M : Z_M)$ on a Montgomery curve can be transformed to the correspondent twisted Edward point (Y_E, Z_E) and vice versa by

$$\begin{aligned} (X_M : Z_M) &\to (Y_E, Z_E) = (X_M - Z_M : X_M + Z_M), \\ (Y_E : Z_E) &\to (X_M, Z_M) = (Y_E + Z_E : Y_E - Z_E). \end{aligned} \tag{9.5}$$

at a cost of only two additions in $\mathbb{F}_p$ each.

9.3. Supersingular isogeny Diffie-Hellman protocol

The main purpose of the classical Diffie-Hellman protocol is that two entities securely agree on a shared secret over a public communication channel that is considered insecure under passive attacks. In the case of the SIDH protocol this secret is obtained by computing the j-invariant of two isomorphic supersingular elliptic curves generated by Alice and Bob that happens to be isogenous to an initial supersingular curve E_0.

The SIDH domain parameters are given as follows. Choose a supersingular elliptic curve E over $\mathbb{F}_{p^2}$, where p is a large prime of the form,

$$p = (l_A)^{e_A}(l_B)^{e_B} f \pm 1,$$

and where l_A and l_B are small prime numbers, e_A and e_B are positive integers, and f is a small cofactor. Then the cardinality of E is given as, $\#E = \left(l_A{}^{e_A} l_B{}^{e_B} f\right)^2$. To simplify the notation let us define $r_A = l_A{}^{e_A}$ and $r_B = l_B{}^{e_B}$. One then chooses two pairs of independent elliptic curve points so that the subgroups $E[r_A]$ and $E[r_B]$ are generated as, $\langle P_A, Q_A \rangle = E[r_A]$ and $\langle P_B, Q_B \rangle = E[r_B]$. Notice that the prime p, the curve E and the generating points P_A, Q_A, P_B, and Q_B, are all considered public domain parameters.

The SIDH key exchange protocol consists of two main phases which are illustrated in Figure 9.1. In the first one, also known as the key generation phase, both parties proceed as follows:

- Alice selects a random number $n_A \in \mathbb{Z}_{r_A}$ and computes the isogeny $\phi_A \colon E_0 \to E_A$ with kernel $\langle P_A + [n_A]Q_A \rangle = \langle R_A \rangle$. Then Alice calculates $\{\phi_A(P_B), \phi_A(Q_B)\}$ and sends to Bob these points together with her computed curve E_A;

- analogously, Bob selects $n_B \in \mathbb{Z}_{r_B}$ randomly and computes the isogeny $\phi_B \colon E_0 \to E_B$ with kernel $\langle P_B + [n_B]Q_B \rangle = \langle R_B \rangle$. Then Bob calculates $\{\phi_B(P_A), \phi_B(Q_A)\}$ and sends to Alice these points together with his computed curve E_B.

In the second phase of the protocol, Alice and Bob compute a shared secret as follows:

- once Alice receives $\{E_B, \phi_B(P_A), \phi_B(Q_A)\}$ from Bob, she calculates the isogeny $\phi_{AB} \colon E_B \to E_{AB}$ with kernel $\langle \phi_B(P_A) + [n_A]\phi_B(Q_A) \rangle = \langle \phi_B(R_A) \rangle$. Finally Alice obtains the shared secret as the j-invariant of E_{AB};
- once Bob receives $\{E_A, \phi_A(P_B), \phi_A(Q_B)\}$ from Alice, he calculates the isogeny $\phi_{BA} \colon E_A \to E_{BA}$ that has kernel $\langle \phi_A(P_B) + [n_B]\phi_A(Q_B) \rangle = \langle \phi_A(R_B) \rangle$. Finally Bob obtains the shared secret as the j-invariant of E_{BA}.

Alice		**Bob**
Choose $n_A \in \mathbb{Z}_{r_A}$		Choose $n_B \in \mathbb{Z}_{r_B}$
$R_A = P_A + [n_A]Q_A$		$R_B = P_B + [n_B]Q_B$
$E_A,\ \phi_A = \nu(E, R_A)$		$E_B, \phi_B = \nu(E, R_B)$
	$E_A,\ \phi_A(P_B),\ \phi_A(Q_B) \longrightarrow$	
	$E_B, \longleftarrow \phi_B(P_A),\ \phi_B(Q_A)$	
$R'_A = \phi_B(P_A) + [n_A]\phi_B(Q_A)$		$R'_B = \phi_A(P_B) + [n_B]\phi_A(Q_B)$
$E_{AB}, \phi'_A = \nu(E_B, R'_A)$		$E_{BA}, \phi'_B = \nu(E_A, R'_B)$
$s_A = j(E_{BCA})$		$s_B = j(E_{ABC})$
	$s_A = s_B$	

Figure 9.1: SIDH protocol. Here ν represent the Velu's formula whose entries are an elliptic curve E and a point $P \in E$ such that generates the kernel of the output isogeny.

One can instantiate the SIDH protocol using different elliptic curve forms such as the Edwards or the Montgomery curves. In this chapter we will focus in the latter form due to its generally more efficient isogeny and elliptic curve arithmetic operations.

9.3.1. Security

The security of the Jao-De Feo Supersingular Isogeny Diffie-Hellman [55] protocol is based on the intractability of the Computational Supersingular Isogeny (CSSI) problem, which is defined as follows

Definition 9.2 (Computational Supersingular Isogeny (CSSI) problem [55])**.** *Let $\phi_A : E_0 \to E_A$ be an isogeny whose kernel is $\langle P_A + [n_A]Q_A \rangle$, where $n_A$is chosen at random from $\mathbb{Z}_{r_A}$ and not divisible by l_A. This problem consist in that given E_A, and the values $\phi_A(P_B)$ and $\phi_A(Q_B)$, find a generator R_A of $\langle P_A + [n_A]Q_A \rangle$.*

It is important to mention that, given a generator $R_A = P_A + [n_A]Q_A$ it is easy to solve for n_A, since E_0 has smooth order and thus extended discrete logarithms are easy in E_0 [159].

There are two approaches to estimate the size of the prime p necessary to offer a certain security level. According with De Feo *et al.* [55], it is believed that solve the CSSI problem

has a complexity of $O(p^{1/4})$ and $O(p^{1/6})$ against classical meet-in-the-middle and quantum Claw finding attacks, respectively. However, in the work presented by Adj *et al.* [3] in 2018, it was observed that despite the van Oorschot-Wiener golden collision finding algorithm has a higher running time, it has a lower cost in comparison with the meet-in-the-middle attack. Then, this algorithm should be used to evaluate the security of SIDH protocol against classical attacks.

In Table 9.1 we show the necessary size in bits for the prime p that offers a determined security level.

Approach	**Security level**			
	$\approx 2^{128}$	$\approx 2^{160}$	$\approx 2^{192}$	$\approx 2^{256}$
According [55]	503	-	751	964
According [3]	434	546	-	610

Table 9.1: Bit-size of p for a security level of 128, 192 and 256 bits for SIDH protocol.

9.3.2. Critical operations

In this section we present the performance-critical operations that are performed in the SIDH protocol and which should be analyzed and carefully implemented.

9.3.2.1. Computation of $P + [k]Q$

At each stage of the SIDH protocol, Alice and Bob must compute the kernel of an isogeny by calculating the point $P + [k]Q$, where P and Q are linearly independent points of order r and $k \in \mathbb{Z}_r$ is a secret.

Since it is generally more efficient to perform the SIDH scalar point multiplications using $\mathbb{P}^1$ arithmetic, we will review in the following two common strategies to compute $\mathbf{x}(P+[k]Q)$.

Method 1: Given P, Q, and k, use the classical Montgomery ladder (Algorithm 35) for computing the x-coordinate of $[k]Q$ followed by the application of the Okeya-Sakurai formula to recover the y-coordinate of $[k]Q$. Finally, perform a projective point addition of the points $(x_P : y_P : 1)$ and $(X_{[k]Q} : Y_{[k]Q} : Z_{[k]Q})$ to obtain $\mathbf{x}(P + [k]Q)$. This strategy requires the knowledge of the y-coordinate of the points P and Q. The time computational expense of this algorithm is given by the execution cost of the Montgomery ladder plus a constant number of prime field multiplications ($< 30\,M$).

Method 2 (three-point ladder): In [55], De Feo et al. proposed a three-point ladder procedure that given the x-coordinate of the points P, Q, and $Q - P$, computes $\mathbf{x}(P + [k]Q)$. This method performs two differential additions and one doubling per bit of the scalar k. This is the same number of operations as computing $[m]P + [n]Q$ using the Bernstein's two-dimensional ladder algorithm [19]. One advantage of the three-point ladder is that all elliptic curve operations are performed using only the x-coordinate of the involved points.

To improve the computation of the SIDH protocol, these two methods can be combined as follows. Notice that during the SIDH key generation phase the initial points are fixed. This situation allows us to apply the Method 1 efficiently since the y-coordinate of the points are known in advance. On the other hand, during the SIDH shared secret phase Alice and Bob exchange points in $\mathbb{P}^1$. Hence, in order to use Method 1 Alice must recover the y-coordinate of the points sent by Bob. However, this will increase the protocol's latency and/or bandwidth. Therefore, Method 2 emerges as a suitable alternative for this scenario, and in fact the three-point ladder algorithm has been adopted by most if not all state-of-the-art implementations of the SIDH protocol (see [51, 54, 116, 50]).

In §10 we introduce in the context of the SIDH protocol, novel strategies for computing $\mathbf{x}(P + [k]Q)$. Our approach performs fewer elliptic curve operations than the methods presented above. Moreover, our algorithms can be used to improve the running time of both, the key generation and the shared secret computation phases of the SIDH protocol.

9.3.2.2. Computing large degree isogenies

Another performance-critical operation is found when Alice and Bob compute and evaluate the isogenies. Let E be an elliptic curve, and let R be a point of order l^e, our goal is to compute the image curve $E/\langle R\rangle$ and evaluate the isogeny $\phi : E \to E/\langle R\rangle$ at some points of E.

The complexity of Velú's formulas scales linearly with respect to the size of the kernel subgroup. Therefore, it is convenient to decompose an l^e-isogeny into e isogenies of degree l for computational efficiency.

Given a point R of order l^e, an isogeny $\phi\colon E \to E/\langle R\rangle$ is calculated as a composition of l-degree isogenies $\phi = \phi_{e-1} \circ \cdots \circ \phi_0$, as follows[2]. Let $E_0 = E$ and $R_0 = R$, then for $0 \leq i < e$, compute: $E_{i+1} = E_i/\langle l^{e-i-1}R_i\rangle$, $\phi_i\colon E_i \to E_{i+1}$, and $R_{i+1} = \phi_i(R_i)$. Thus, $E/\langle R\rangle = E_e$. Using the point $l^{e-i-1}R_i$, the curve E_{i+1} and the isogeny ϕ_i can be readily computed in polynomial time by means of Vélu's formulas (see[164, theorem 12.16]).

Formulas for constructing and evaluate isogenies

In this section we give explicit formulas for compute and evaluate isogenies of Montgomery and twisted Edwards curves.

Lemma 9.1. *Theorem 1 of [49] establish that: Let $P \in E(\bar{\mathbb{F}}_q)$ be a point of order $d = 2l+1$ on the Montgomery curve $E_{A,B}/\mathbb{F}_q : By^2 = x(x^2+Ax+1)$ and write $\sigma = \sum_{i=1}^{l} 1/x_{[i]P} - x_{[i]P}$ and $\pi = \prod_{i=1}^{l} x_{[i]P}$. The Montgomery curve*

$$E'_{A',B'}/\mathbb{F}_q : B'y^2 = x(x^2 + A'x + 1)$$

with $A' = (6\sigma + A)\cdot\pi$ and $B' = B\cdot\pi^2$ is the codomain of the normalized d-isogeny $\phi : E \to E'$ with $\ker(\phi) = \langle P\rangle$. Moreover, we can evaluate a point $Q = (x : z)$ not in $\langle P\rangle$ via

$$\begin{aligned} x' &= x_Q \cdot \left(\prod_{i=1}^{\ell} \left[(x_Q - z_Q)(x_{[i]P} + z_{[i]P}) + (x_Q + z_Q)(x_{[i]P} - z_{[i]P})\right]\right)^2 \\ z' &= z_Q \cdot \left(\prod_{i=1}^{\ell} \left[(x_Q - z_Q)(x_{[i]P} + z_{[i]P}) - (x_Q + z_Q)(x_{[i]P} - z_{[i]P})\right]\right)^2 \end{aligned}$$

which has a cost of $4lM + 2S + (4l + 2)A$, where A stands for the cost of an addition in $\mathbb{F}_q$.

Moody presented formulas for isogenies between Edwards curves [133] and more precisely, twisted Edwards curves which are of our particular interest.

Lemma 9.2. *([133, Corollary 1]) Suppose F is a subgroup of the twisted Edwards curve $E_{a,d}$ with odd order $s = 2l + 1$, where F is the set of points*

$$F = \{(\mathcal{O}), (\pm\alpha_1, \beta_1), \ldots, (\pm\alpha_l, \beta_l)\}$$

Define

$$\psi(Q) = \left(\prod_{P\in F\setminus\{\mathcal{O}\}} \frac{x_{Q+P}}{x_P}, \prod_{P\in F\setminus\{\mathcal{O}\}} \frac{y_{Q+P}}{y_P}\right).$$

Then ψ is an s-isogeny, with kernel F from the curve $E_{a,d}$ to the curve $E_{a',d'}$ where $a' := a^s, d' = B^8 d^s$ and $B = \prod_{i=1}^{l} \beta_i$.

[2] See [55] for a comprehensive discussion on optimal approaches for computing a l^e-degree isogeny.

We can observe, as was pointed by Meyer and Reith[125] that parameters a' and d' depend only of a, d and the y-coordinates of points in F. So, we can make use of this formulas switching to Montgomery curves for curve arithmetic purposes using the maps from §9.2.3 which are almost-free-of-cost. We can consider Montgomery projective coordinates and we get that $A := 2(a + d)$ and $C := a - d$ then we can also consider Edwards YZ-coordinates, *i.e.*, consider $\mathbf{y}(P) := \beta_i = (y_{P_i} : z_{P_i})$ for $P \in F \setminus \mathcal{O}$ and obtain that $a' := B_z^8 a^s$ and $d' := B_y^8 d^s$ where $B_y := \prod_{i=1}^{l} y_{P_i}$ and $B_z := \prod_{i=1}^{l} z_{P_i}$. The cost of computing this *projective* version of a' and d' is about $(2l + 2 + \log(s)/2)\mathrm{M} + (6 + \log(s))\mathrm{S}$.

Chapter 10

A faster software implementation of the Supersingular Isogeny Diffie-Hellman protocol

Since its introduction by Jao and De Feo in 2011, the supersingular isogeny Diffie-Hellman (SIDH) key exchange protocol has positioned itself as a promising candidate for post-quantum cryptography. One salient feature of the SIDH protocol is that it requires exceptionally short key sizes. However, the latency associated to SIDH is higher than the ones reported for other post-quantum cryptosystem proposals. Aiming to accelerate the SIDH runtime performance, we present in this work several algorithmic optimizations targeting both elliptic-curve and field arithmetic operations.

10.1. Introduction

Over the last decade there has been an intense research effort to find hard mathematical problems that would be presumably hard to solve by a quantum attacker and at the same time could be used to build reasonably efficient public-key cryptoschemes. One such proposal is the hardness of finding an isogeny map between two elliptic curves, i.e., given two elliptic curves E_0 and E_1, the problem of finding a morphism $\phi\colon E_0 \to E_1$ that maps points from E_0 to E_1 while preserving $\phi(\mathcal{O}_{E_0}) = \mathcal{O}_{E_1}$. This proposal has spawned a new line of research generally known as isogeny-based cryptography.

In 2011, Jao and De Feo proposed the problem of finding the isogeny map between two supersingular elliptic curves, a setting where the attack in [39] does not apply anymore. This proposal led to the Supersingular Isogeny-based Diffie-Hellman key exchange protocol (SIDH) [95] (see also [55]). As of today, the best-known algorithms against the SIDH protocol have an exponential time complexity for both classical and quantum attackers.[1]

Although the SIDH public key size for achieving a 128-bit security level in the quantum setting was already reported as small as 564 bytes in [51], this SIDH public key size was recently further reduced in [50] to just 330 bytes. However impressive, these key size credentials have to be contrasted against SIDH relatively slow runtime performance. Indeed, the SIDH key exchange protocol has a latency in the order of milliseconds when implemented in high-end Intel processors. This timing is significantly higher than the one achieved by several other quantum-resistant cryptosystem proposals. Consequently some recent works

[1] See §9.3 for a detailed description of the SIDH protocol.

have focused on devising strategies to reduce the runtime cost of the SIDH protocol. For example, Koziel et al. presented a parallel evaluation of isogenies implemented on an FPGA architecture [115, 114], reporting important speedups for this protocol. These developments show the increasing research interest on developing techniques able to accelerate the SIDH protocol software and hardware implementations.

In order to reduce the running time of the SIDH protocol it is important to identify performance-critical operations. Upon initial inspection it is noted that this scheme computes a shared secret by performing a high number of elliptic curve and field arithmetic operations. Taking into consideration the above, our main contributions for accelerating the performance of the SIDH key exchange protocol can be summarized as follows:

- Building on the scalar multiplication procedures reported in [138], we propose a right-to-left Montgomery ladder that efficiently computes the elliptic curve scalar multiplication $P + [k]Q$ required by the two main phases of the SIDH protocol. Our strategy achieves a factor 1.4 speedup compare with the well-known three-point ladder algorithm presented in [55]. Further, when the base point Q is known in advance our algorithm can take advantage of a precomputed look-up table derived from Q, which in principle allows us to accelerate the aforementioned computation. Nevertheless, this approach led us to discover several unforeseen implementation difficulties which are somewhat related to the parameter selection used by Costello et al. in [51]. We describe how these issues were efficiently circumvented allowing us to also report a higher speedup factor for the SIDH fixed-point scalar multiplication computation.
- We present an optimized point tripling formula specialized for Montgomery elliptic curves. We consider the case when the elliptic curve parameter A that defines the elliptic curve equation is expressed as a quotient $A = A_0/A_1$. Our formulation saves one multiplication at the cost of one squaring and one addition, which are performed in the quadratic extension field $\mathbb{F}_{p^2}$. This saving is valuable if one considers that Bob has to perform hundreds of point tripling computations in both phases of the SIDH protocol.
- We developed an optimized prime field arithmetic that takes advantage of the recent instructions devoted to achieve ultra fast integer arithmetic computations, such as the Bit Manipulation Instructions (BMI2) and the addition instructions with independent carry chains (ADX) supported by high-end Intel and AMD 64-bit processors described in §3.2.

Combining all the above improvements, the execution of our library achieves a factor 1.33 speedup compared with the running time associated to the fastest SIDH software implementation reported in the literature (see §10.5 for more details about the performance achieved by our library).

10.2. A novel algorithm for computing $\mathbf{x}(P + [k]Q)$

The Montgomery ladder (Algorithm 35) is known as a *left-to-right* algorithm, since it computes $[k]P$ by scanning the bits of the scalar k from the most-significant to the least-significant bit. A *right-to-left* evaluation of the Montgomery ladder was recently introduced to accelerate the scalar multiplication operation in the fixed-point scenario. This approach was first applied in the context of binary elliptic curves [100, 137], and then, it was further extended to Montgomery curves [138]. Building on the right-to-left ladder technique of [138] we present here an algorithm that computes $\mathbf{x}(P + [k]Q)$ efficiently.

The proposed approach is shown in Algorithm 36, which given the points $\mathbf{x}(P)$, $\mathbf{x}(Q)$, and $\mathbf{x}(Q-P)$ computes $\mathbf{x}(P+[k]Q)$ provided that $P, Q-P \notin \{\mathcal{O}, (0,0)\}$. Algorithm 36 uses

Algorithm 36 Variable-point multiplication of $\mathbf{x}(P + [k]Q)$.

Input: $(k, \mathbf{x}(P), \mathbf{x}(Q), \mathbf{x}(Q-P))$, where k is a t-bit number; and $\mathbf{x}(P), \mathbf{x}(Q), \mathbf{x}(Q-P) \in \mathbb{P}^1$ are a representation of $P, Q, Q-P \in E(\mathbb{F}_q)$, respectively.
Output: $\mathbf{x}(P + [k]Q)$.

1: Initialize $R_0 \leftarrow \mathbf{x}(Q)$, $R_1 \leftarrow \mathbf{x}(P)$ and $R_2 \leftarrow \mathbf{x}(Q-P)$
2: **for** $i \leftarrow 0$ **to** $t-1$ **do**
3: **if** $k_i = 1$ **then**
4: $R_1 \leftarrow R_0 +_{(R_2)} R_1$
5: **else**
6: $R_2 \leftarrow R_0 +_{(R_1)} R_2$
7: **end if**
8: $R_0 \leftarrow [2]R_0$
9: **end for**
10: **return** R_1

three accumulators, namely $R_0, R_1, R_2 \in \mathbb{P}^1$, and scans the bits of k from the least-significant to the most-significant bit. At the i-th iteration, the k_i bit value determines whether R_0 must be accumulated in R_1 or in R_2. Thereafter R_0 is doubled unconditionally. Accumulators are updated by one differential addition and one point doubling and they always preserve the relation $R_0 - R_1 = R_2$. This is the same invariant relation that holds for the classical Montgomery ladder of Algorithm 35. However notice that in the case of Algorithm 36, the value stored in R_2 may vary between iterations, unlike Algorithm 35 where R_2 is always fixed to $\mathbf{x}(P)$. In summary, Algorithm 36 performs $6t$M+$4t$S, which for a practical software implementation implies a cost of approximately 8.6 M-per-bit.

As observed in [138], in the case that Q is a fixed point known in advance, one can construct a look-up table $T(Q)$ by precomputing constants that are obtained from the x-coordinate of multiples of the point Q as,

$$T(Q) = (T_0, \ldots, T_{t-1}), \text{ where } T_i = \frac{x_i + 1}{x_i - 1}, \text{ and } (x_i, y_i) = [2^i]Q, \tag{10.1}$$

for $0 \leq i < t$, where t is the size in bits of the scalar k. Using this approach, the point doubling computation in line 8 of Algorithm 36 can be replaced by a query to the look-up table T. For completeness, we show in Algorithm 37 the fixed-point version of Algorithm 36. The differential additions in lines 5 and 7 of Algorithm 37 are computed more efficiently (using 3M+2S) using the precomputed value T_i as input [138]. It is worth to mention that the look-up table queries of Algorithm 37 use non-secret indexes. This is in stark contrast with other fixed-point multiplication algorithms where protecting look-up table accesses is mandatory, a measure that unavoidably introduces performance overheads. The computational cost of Algorithm 37 for computing $\mathbf{x}(P + [k]Q)$ drops to $3t$M+$2t$S, which is around 4.3 M-per-bit.

As in the case of the classical Montgomery ladder we show how to recover the y-coordinate of $P + [k]Q$ from the values computed by the right-to-left ladder algorithm. This method will be discussed at length in §10.2.2.

10.2.1. Applying the new algorithm to the SIDH protocol

As already mentioned, the $P + [k]Q$ operation must be performed in both phases of the SIDH protocol. During the key generation phase, this operation uses points that are known in advance. Conversely, during the shared secret generation phase operations are performed over unknown points. In the remaining of this section we describe the application of our algorithms to these scenarios and we also discuss some relevant issues that appeared on their implementations. We present first the description of the variable point case.

Algorithm 37 Fixed-point multiplication of $\mathbf{x}(P+[k]Q)$.

Input: $(k, \mathbf{x}(P), \mathbf{x}(Q-P))$, where k is a t-bit number; and $\mathbf{x}(P), \mathbf{x}(Q-P) \in \mathbb{P}^1$ are a representation of $P, Q-P \in E(\mathbb{F}_q)$, respectively.
Output: $\mathbf{x}(P+[k]Q)$.

Precomputation:
1: $T(Q)$ is a look-up table defined as in Equation (10.1).
Computation:
2: Initialize $R_1 \leftarrow \mathbf{x}(P)$, and $R_2 \leftarrow \mathbf{x}(Q-P)$.
3: **for** $i \leftarrow 0$ **to** $t-1$ **do**
4: **if** $k_i = 1$ **then**
5: $R_1 \leftarrow T_i +_{(R_2)} R_1$
6: **else**
7: $R_2 \leftarrow T_i +_{(R_1)} R_2$
8: **end if**
9: **end for**
10: **return** R_1 ▷ For y-coordinate recovery, return also R_2.

10.2.1.1. Computing $P+[k]Q$ in the variable-point scenario

During the secret generation phase, Alice[2] receives $\mathbf{x}(\phi_B(P_A))$, $\mathbf{x}(\phi_B(Q_A))$ from Bob. Afterwards, Alice must calculate $\mathbf{x}(\phi_B(P_A)+[n_A]\phi_B(Q_A))$. To that end, Method 1 could be applied in this scenario. Nevertheless, this would require that Alice must know the y-coordinate values of the points $\phi_B(P_A)$ and $\phi_B(Q_A)$. To circumvent this difficulty, Bob could send the y-coordinate of these two points, but this would increase the public key sizes considerably. Alternatively, Bob could encode the y-coordinate of each point into one bit. However, this would force Alice to decompress a point using time-consuming square-roots over $\mathbb{F}_q$. We conclude that Method 1 becomes an inadequate choice for this scenario.

On the other hand if Bob additionally sends $\mathbf{x}(\phi_B(P_A-Q_A))$, then Alice can perform the three-point ladder algorithm [55] (corresponding to Method 2 described in the previous section). This is the mechanism followed by most of the state-of-the-art implementations, such as [51, 54, 116, 50]. Nevertheless notice that the three-point ladder algorithm has a higher computational cost as compared to Method 1.

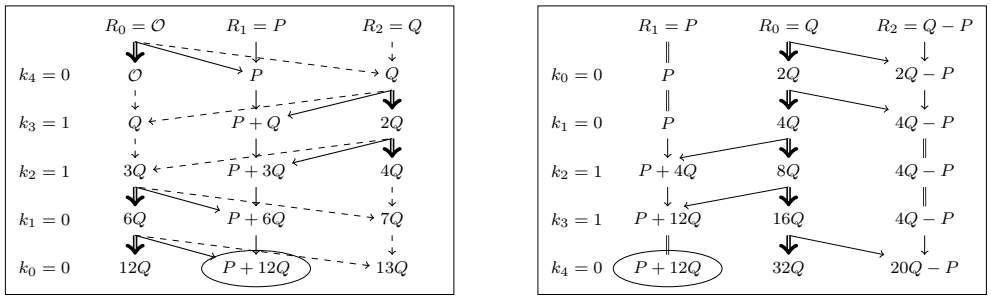
A more efficient approach consists of applying Algorithm 36 since it provides a significant saving of field arithmetic operations when compared to Method 1 or Method 2. Furthermore given the same input values, Algorithm 36 and the three-point ladder procedure produce the same output. This implies that both algorithms can share the same interface, which is especially valuable for minimizing the changes of existent software implementations. An extra advantage of adopting Algorithm 36 is that it does not increase the public key size. Figure 10.1 shows an example contrasting the execution of Algorithm 36 and the three-point ladder procedure when processing the same scalar $k = 12$.

By replacing the three-point ladder algorithm with Algorithm 36, we estimate to achieve a factor 1.38 speedup (Table 10.1). In §10.5, this prediction is experimentally verified through the benchmarking of our SIDH protocol implementation.

10.2.1.2. Computing $P+[k]Q$ in the fixed-point scenario

In this phase, one can exploit more avenues for further optimizations. First of all, the involved points are fixed, allowing us to use precomputed look-up tables that could possibly accelerate operations. In this scenario it is clear that Method 1 is more efficient than

[2]Since the same analysis applies to Bob by just swapping sub-indexes, in this section we only summarize the operations performed by Alice.



(a) Three-point ladder from [55]. **(b)** Algorithm 36 (our work)

Figure 10.1: Calculating $P + [12]Q$, where the scalar is a 5-bit number $(12)_{10} = (01100)_2$. In Fig.10.1(a) we show the steps for the three-point ladder algorithm, and in Fig.10.1(b) the steps for the ladder of Algorithm 36. If we remove the central column in Fig.10.1(a), it becomes clear that the three-point ladder procedure is in essence a classical Montgomery ladder. Also note that the column in the center of Fig.10.1(b) shows a sequence of consecutive point doublings of Q. When Q is a fixed point, this column can be precomputed.

Method 2. However, it is not obvious how these two strategies could benefit from precomputation techniques.

Taking advantage of the fact that the points are known in advance one can directly apply Algorithm 37. Using this procedure one can expect a factor 1.77 speedup when compared with Method 1 (Table 10.1). Hence, the SIDH key generation phase can be also accelerated in a similar way as it happens in the ECDH protocol implementation reported in [138].

Algorithm 37 was designed to perform arithmetic operations over $\mathbb{F}_q$, where in the general setting $q = p^2$. However, Algorithm 37 can also be useful in the setting $q = p$. For the rest of this section, we adhere to the elliptic curve parameters proposed by Costello *et al.* [51].

Let $P_A = (x_{P_A}, y_{P_A})$ and $P_B = (x_{P_B}, y_{P_B})$ be the base points of Alice and Bob, respectively. By construction, the authors of [51] selected P_A and P_B in such a way that their affine coordinates lie in $\mathbb{F}_p$. Using the distortion map action, they obtained the points $Q_A = (-x_{P_A}, y_{P_A} i)$ and $Q_B = (-x_{P_B}, y_{P_B} i)$, which happen to be linearly independent to P_A and P_B, respectively. Let us recall that Alice's points have order $r_A = 2^{e_A}$, while Bob's points have order $r_B = 3^{e_B}$. Under these conditions, Costello *et al.* [51] employed the following method to compute $\mathbf{x}(P + [k]Q)$ in the fixed-point scenario.

Method 3: Compute the Montgomery ladder algorithm in $\mathbb{F}_p$ to obtain $\mathbf{x}([k]Q) \in \mathbb{P}^1(\mathbb{F}_p)$. Then recover the y-coordinate of $[k]Q$, and finally perform a projective point addition to obtain $\mathbf{x}(P + [k]Q)$. The cost of Method 3 is of about 8.2m-per-bit, where m denotes a multiplication in $\mathbb{F}_p$. Assuming $1M = 3m$, one concludes that Method 3 takes around one third of the cost of Method 1.

Sticking to the same conditions, we consider to perform operations of Algorithm 37 over $\mathbb{F}_p$. To that end, this procedure requires $\mathbf{x}(Q - P) \in \mathbb{P}^1(\mathbb{F}_p)$. Unfortunately, this is not the case for the points selected using the parameter generation of [51]. Indeed, note that one of the projective coordinates of $\mathbf{x}(Q - P) = ((x_P^2 + 1)i : 2x_P)$ is not in $\mathbb{F}_p$. As a consequence, it would appear that the point selection method used in [51] restricts the use of Algorithm 37.

However, not all is lost. We prevent these issues and propose an alternative solution that combines the efficiency offered by Algorithm 37 and the parameter selection described above.

Our idea consists of employing Algorithm 37 to compute $\mathbf{x}(S + [k']Q)$ for an order-d point $S \in E(\mathbb{F}_{p^2})$, such that $\mathbf{x}(Q - S) \in \mathbb{P}^1(\mathbb{F}_p)$ and $k' \equiv k/d \mod r$. Thereafter one can compute $\mathbf{x}([k]Q) = [d]\mathbf{x}(S + [k']Q)$, followed by the recovery of the y-coordinate of $[k]Q$, and finally the addition of the point P (as in Method 3) to end up with $\mathbf{x}(P + [k]Q)$. We must ensure

that $S \notin \langle Q \rangle$, and for efficiency reasons we also impose the restriction that the point S must have a low order $d \neq 2$. These steps are summarized in Algorithm 38.

Algorithm 38 Proposed algorithm to compute $\mathbf{x}(P + [k]Q)$ in the fixed-point scenario and adapted to the elliptic curve parameters defined in [51]. Let $I \in \{A = \text{Alice}, B = \text{Bob}\}$ denote the SIDH protocol participant.

Input: (k, t_I, P_I, Q_I, S_I), where k is an t_I-bit number ($t_A = 372$ and $t_B = 379$), P_I and Q_I are points of order r_I (defined as in [51]), and S_I is a point of order d_I (defined as in Equation (10.2)).

Output: $\mathbf{x}(P_I + [k]Q_I)$.

Precomputation:

Compute a look-up table $T(Q_I)$ defined as in Equation (10.1). Compute U_0, U_1, V_0, V_1 $\in E(\mathbb{F}_{p^2})$ defined as in Equation (10.5).

Computation:

$k' \leftarrow k/d_I \mod r_I$
if $I = \text{Alice}$ **then** ▷ §10.2.2.1.
    $(\alpha, \beta, k') \leftarrow (k'_{e_A-1}, k'_{e_A-2}, k' \mod 2^{e_A-2})$
end if
$R_1, R_2 \leftarrow \text{ALGORITHM37}_{q=p,T(Q_I)}(k', \mathbf{x}(S_I), \mathbf{x}(Q_I - S_I))$ ▷ $R_1 = \mathbf{x}(S_I + [k']Q_I)$
$R_1 \leftarrow [d_I]R_1$, $R_2 \leftarrow [d_I]R_2$
$(X_{R_1} : Y_{R_1} : Z_{R_1}) \leftarrow y\text{-RECOVER}(R_1, R_2)$ ▷ §10.2.2.
if $I = \text{Alice}$ **then** ▷ §10.2.2.1.
    $U \leftarrow \text{CMOVE}(\alpha, U_0, U_1)$
    $V \leftarrow \text{CMOVE}(\beta, V_0, V_1)$
    $R_3 \leftarrow (X_{R_1} : Y_{R_1} : Z_{R_1}) + U + V$
else if $I = \text{Bob}$ **then**
    $R_3 \leftarrow (X_{R_1} : Y_{R_1} : Z_{R_1}) + P_B$
end if
return $\mathbf{x}(R_3)$

Looking for a suitable point S the most natural choice to obtain low order points is that Alice uses Bob's points and vice versa. Hence, let us define the following points,

$$S = \begin{cases} S_A = [3^{e_B-1}]Q_B\,, & \text{for Alice;} \\ S_B = [2^{e_A-2}]Q_A\,, & \text{for Bob.} \end{cases} \tag{10.2}$$

By construction S_A and S_B were chosen such that both $\mathbf{x}(Q_A - S_A)$ and $\mathbf{x}(Q_B - S_B)$ are in $\mathbb{P}^1(\mathbb{F}_p)$.

The cost of Algorithm 38 is similar to the cost of Algorithm 37 plus a constant number of multiplications (< 30M). However, the scalar multiplication operations are performed over $\mathbb{F}_p$ resulting in a cost of approximately 4.6m-per-bit. Thus, Algorithm 38 provides an acceleration of a factor 1.78 speedup compared to the performance of Method 3.

Table 10.1 summarizes the computational costs of the algorithms discussed in this section. We considered two scenarios: the first one is when points are fixed and known in advance; and the second one when dealing with unknown points. For both scenarios our methods outperform the techniques used in state-of-the-art implementations [51, 54, 116, 50]. In §10.5, we report the impact yielded by these algorithms on the SIDH protocol overall performance.

10.2.2. Recovering the y-coordinate of $P + [k]Q$

As in the classical Montgomery ladder, one can recover the y-coordinate of $P+[k]Q$ using the values computed in the last iteration of the right-to-left algorithm. This can be done by

Scenario	Field	Mult-per-bit	AF	Algorithms
Fixed-point	$\mathbb{F}_{p^2}$	7.6M		Method 1
		4.3M	1.77	**Alg. 37 (our work)**
	$\mathbb{F}_p$	8.2m		Method 3 [51]
		4.6m	1.78	**Alg. 38 (our work)**
Variable-point	$\mathbb{F}_{p^2}$	11.9M		3-point ladder [55]
		8.6M	1.38	**Alg. 36 (our work)**

Table 10.1: Algorithms for computing $\mathbf{x}(P+[k]Q)$ in the fixed- and variable-point scenario. The third column shows ladder step arithmetic operation costs and the fourth column shows the predicted acceleration factor. We assume that 1M=3m, 1S=0.66M, and 1s=0.8m.

restating the formula given in Okeya-Sakurai's paper [136, Corollary 2] as discussed next.

Let us consider an affine point (x, y) with $y \neq 0$, and the points $P_i = (X_i : Z_i) \in \mathbb{P}^1$, for $i = 1, 2, 3$ such that, $(X_2 : Z_2) = (X_1 : Z_1) - (x, y)$, and $(X_3 : Z_3) = (X_1 : Z_1) + (x, y)$. Then one can compute,

$$\begin{aligned} X_1' &= 4ByZ_1Z_2Z_3X_1 \\ Y_1' &= (X_2Z_3 - Z_2X_3)(X_1 - Z_1x)^2 \\ Z_1' &= 4ByZ_1Z_2Z_3Z_1\,, \end{aligned} \tag{10.3}$$

such that the point $(X_1' : Y_1' : Z_1') \in \mathbb{P}^2$ belongs to the same equivalence class of the point $(X_1 : Z_1) \in \mathbb{P}^1$.

Recall that the loop-invariant of the right-to-left ladder is $R_0 - R_1 = R_2$. Thus, the accumulators in the i-th iteration hold the values $R_0 = [2^i]Q$, $R_1 = P + [k \mod 2^i]Q$, and $R_2 = [2^i - (k \mod 2^i)]Q - P$, respectively. Since k is a t-bit number, then after t iterations one can compute $R_3 = R_0 +_{(R_2)} R_1$ and use the points stored in those four accumulators to apply Equation (10.4) as: $(x, y) \leftarrow R_0$; $(X_1 : Z_1) \leftarrow R_1$; $(X_2 : Z_2) \leftarrow R_2$; and $(X_3 : Z_3) \leftarrow R_3$. This allows the recovery of the y-coordinate of the point $R_1 = P + [k]Q$. The cost of the y-coordinate recovering just described is one differential addition more than the original Okeya-Sakurai technique. Thus, the only requirement is to have a previous knowledge of the point $[2^t]Q$.

In the fixed-point scenario, the point $[2^t]Q$ can be saved together with the look-up table constants. This enables the usage of Algorithm 37 as a subroutine of Algorithm 38 for accelerating the $P + [k]Q$ operation in the fixed-point scenario. Nonetheless, the fact that Alice uses points of 2-smooth order produces some troubles for recovering the y-coordinate of $[k]Q_A$. We dedicate the next subsection for exposing this issue and the solution that we found to it.

10.2.2.1. An implementation issue: Alice's y-coordinate recovering

We found a subtle issue when Alice tries to recover the y-coordinate of $[k]Q_A$. Since Q_A has order 2^{e_A} then $R_0 = [2^i]Q_A = \mathcal{O}$ for all $i \geq e_A$. Note that for a t-bit scalar k, the point $R_0 = [2^t]Q$ is directly involved in the recovery of the projective coordinates of the point $P + [k]Q$. Hence, after running e_A steps of the right-to-left ladder we end up having $y_{R_0} = 0$, which makes the usage of Equation (10.4) impossible. In order to overcome this problem we propose the solution described in Algorithm 38. The main idea consists of running only t' iterations of Algorithm 37, where t' is the largest number such that $t' < e_A$ and the y-coordinate of $R_0 = [2^{t'}]Q$ is different than 0. This allows us to recover the y-coordinate

using Equation (10.4). However notice that if we set $t' = e_A - 1$, then R_0 becomes a point of order two, i.e. $y_{R_0} = 0$. For this reason, we chose $t' = e_A - 2$, since then $R_0 = [2^{t'}]Q_A$, and $y_{R_0} \neq 0$. The points corresponding to the last two missing steps of the ladder can be conditionally added together with the point P_A.

Referring to Algorithm 38, in step 1 the scalar k' is computed. Then in steps 2-4 the values of the two most significant bits of k' are saved as $\alpha = k'_{e_A-1}$ and $\beta = k'_{e_A-2}$. Also k' is updated to consider only its t' least significant bits. Then, Algorithm 37 computes $S_A + [k']Q_A$ performing exactly t' iterations. After clearing S_A, the accumulators hold $R_0 = R_1 + R_2 = [2^{e_A-2}]Q_A$ and $y_{R_0} \neq 0$. This allows to recover the y-coordinate of $[3(k' \bmod 2^{e_A-2})]Q_A$ using Equation (10.4). Thereafter, in steps 9-11 the points $3k'_{e_A-1}2^{e_A-1}Q_A$ and $3k'_{e_A-2}2^{e_A-2}Q_A$ are conditionally added to obtain $[3k']Q_A = [k]Q_A$. Finally the procedure returns $\mathbf{x}(P_A + [k]Q_A)$.

The conditional point additions of steps 9-11 must be computed in a secure way. One common technique is to conditionally select $U \in E(\mathbb{F}_q)$ from $\{U, \mathcal{O}\}$ according to the bit value (this can be securely implemented using a conditional move or conditional swap). However, there is an issue when the bit chooses $\mathcal{O}$ due to the projective point addition is not complete, i.e. it can not handle the point at infinity. To remedy this situation, we precompute the following points:

$$\begin{aligned} U_0 &= -P_A\,, \quad U_1 = U_0 + [3 \times 2^{e_A-1}]Q_A\,, \\ V_0 &= [2]P_A\,, \quad V_1 = V_0 + [3 \times 2^{e_A-2}]Q_A. \end{aligned} \tag{10.4}$$

Steps 10-12 of Algorithm 38 show how to select these points by using the auxiliary function CMOVE, which conditionally moves the points according to the input bit value. Note that regardless the bit values, our procedure always add P_A.

The overhead caused by these modifications in Alice's side is negligible in comparison with Bob's method. By using this approach, both Alice and Bob can benefit from the usage of a precomputation table to accelerate the key generation phase.

10.3. Optimization of point tripling in Montgomery curves

As we see in §9.3.2.2, the calculation of large-degree isogenies requires to compute either $[2^i]\mathbf{x}(P)$ or $[3^i]\mathbf{x}(P)$ for some point $P \in E(\mathbb{F}_q)$ and some integer i. These operations are computed repeatedly applying point doubling or tripling algorithms using projective formulas in $\mathbb{P}^1$. For the sake of efficiency, we look for an optimized formula that computes the point tripling operation faster.

A common technique to compute $[3]P$ consists of performing a point doubling followed by a differential point addition, i.e. $[3]P = [2]P +_{(P)} P$. This method has a cost of 7M + 4S + 8A field arithmetic operations. Recently, Subramanya Rao [157] showed a more efficient formula to compute a point tripling. Given $P = (X_1 : Z_1)$ and let A be the Montgomery curve parameter, such a formula calculates $[3]P = (X_3 : Z_3)$ as follows:

$$\begin{aligned} \lambda &= \left(X_1{}^2 - Z_1{}^2\right)^2 \\ \gamma &= 4(X_1{}^2 + Z_1{}^2 + AX_1Z_1) \\ X_3 &= X_1\left(\lambda - \gamma Z_1{}^2\right)^2 \\ Z_3 &= Z_1\left(\lambda - \gamma X_1{}^2\right)^2. \end{aligned} \tag{10.5}$$

This formula is derived by coalescing the point doubling and differential addition and its computational cost is 6M + 5S + 9A field operations.

In the SIDH context, the parameter A of the Montgomery elliptic curve (see Equation (9.1)) is not fixed, since it may change due to the computation of isogenies. Because of this, the parameter A is represented as a quotient $A = A_0/A_1$. This representation, which was introduced in [51], avoids the usage of inversions for the computation of large-degree isogenies. Therefore, tripling formulas must be modified to operate with A_0 and A_1. Table 10.2 shows the cost of several tripling formulas reported in the literature.

$A = A_0/A_1$	Cost	Precomputation	Reference
$A_1 = 1$	7M + 4S + 8A	$\{(A+2)/4\}$	[132]
	6M + 5S + 9A	$\emptyset$	[157]
A_1 arbitrary	8M + 4S + 8A	$\{A_0 + 2A_1, 4A_1\}$	[51]
	7M + 5S + 10A	$\{A_0 \pm 2A_1\}$	[49]
	7M + 5S + 9A	$\{A_0 - 2A_1, 2A_1\}$	**Our work**

Table 10.2: Cost of point tripling formulas (in $\mathbb{P}^1$) for a Montgomery elliptic curve with parameter $A = A_0/A_1$.

We optimize the calculation of the tripling formula observing that $2X_1Z_1$ can be calculated from ${X_1}^2$, ${Z_1}^2$, and $(X_1 + Z_1)^2$ as:

$$2X_1Z_1 = (X_1 + Z_1)^2 - ({X_1}^2 + {Z_1}^2)\,; \tag{10.6}$$

thus, λ from (10.5) can be also calculated using (10.6) as follows:

$$\begin{aligned} \lambda &= (X_1 + Z_1)^2(X_1 - Z_1)^2 \\ &= (X_1 + Z_1)^2\left[({X_1}^2 + {Z_1}^2) - 2X_1Z_1\right]; \end{aligned} \tag{10.7}$$

likewise, γ from (10.5) is given as:

$$\begin{aligned} \gamma &= 4\left({X_1}^2 + {Z_1}^2 + AX_1Z_1\right) \\ &= 2\left[2({X_1}^2 + {Z_1}^2 + AX_1Z_1)\right] \\ &= 2\left[2(X_1 + Z_1)^2 + (A-2)(2X_1Z_1)\right]. \end{aligned} \tag{10.8}$$

Using Equations (10.6))-((10.8) and considering that $A = A_0/A_1$, we calculate $[3]P = (X_3 : Z_3)$ as follows:

$$\begin{aligned} \lambda &= (2A_1)(X_1 + Z_1)^2\left[({X_1}^2 + {Z_1}^2) - 2X_1Z_1\right] \\ \gamma &= 4\left[(2A_1)(X_1 + Z_1)^2 + (A_0 - 2A_1)(2X_1Z_1)\right] \\ X_3 &= X_1\left(\lambda - \gamma {Z_1}^2\right)^2 \\ Z_3 &= Z_1\left(\lambda - \gamma {X_1}^2\right)^2. \end{aligned} \tag{10.9}$$

Assuming $A_0' = A_0 - 2A_1$ and $A_1' = 2A_1$ are precomputed, then our point tripling formula (Equation (10.9)) requires 7M + 5S + 9A using the following sequence of operations:

$$\begin{array}{lll} 1 : t_0 \leftarrow (X_1)^2 & 8 : t_2 \leftarrow A_1' \times t_2 & 15 : t_2 \leftarrow t_2 \times t_4 \\ 2 : t_1 \leftarrow (Z_1)^2 & 9 : t_5 \leftarrow t_2 + t_5 & 16 : t_0 \leftarrow t_2 - t_0 \\ 3 : t_2 \leftarrow X_1 + Z_1 & 10 : t_5 \leftarrow t_5 + t_5 & 17 : t_1 \leftarrow t_2 - t_1 \\ 4 : t_2 \leftarrow (t_2)^2 & 11 : t_5 \leftarrow t_5 + t_5 & 18 : t_0 \leftarrow (t_0)^2 \\ 5 : t_3 \leftarrow t_0 + t_1 & 12 : t_0 \leftarrow t_0 \times t_5 & 19 : t_1 \leftarrow (t_1)^2 \\ 6 : t_4 \leftarrow t_2 - t_3 & 13 : t_1 \leftarrow t_1 \times t_5 & 20 : X_3 \leftarrow X_1 \times t_1 \\ 7 : t_5 \leftarrow A_0' \times t_4 & 14 : t_4 \leftarrow t_3 - t_4 & 21 : Z_3 \leftarrow Z_1 \times t_0 \end{array}$$

It can be seen that our formula improves point tripling computation by 1M - 1S - 1A with respect to the formula used by Costello *et al.* in [51] (Table 10.2). Independent work of Costello and Hisil [49] gives formulas for point tripling; however, our formulas are one field addition faster.

Bob's isogeny computations require the frequent computation of point tripling operations to calculate points of the form $[3^i]P$. Therefore, one can see that any improvement in the tripling formula impacts directly the calculation of large-degree isogenies, which are by far the most time-consuming operations in the SIDH protocol.

10.4. Finite field arithmetic implementation

The instantiation of the SIDH protocol by Costello *et al.* [51] uses a prime modulus of the form, $p_{\text{CLN}} = 2^{372}3^{239} - 1$. Notice that this prime can be represented using twelve 64-bit words.

Since the SIDH protocol computes isogenies of supersingular elliptic curves defined over the field $\mathbb{F}_{p^2}$, a sensible implementation of the SIDH protocol must implement fast arithmetic in the quadratic field $\mathbb{F}_{p^2}$. Quadratic field arithmetic can be performed more efficiently by means of a field towering approach that relies on an optimized implementation of the base field arithmetic $\mathbb{F}_p$. For example, the multiplication and squaring operations in the quadratic extension field translate to the computation of three and two field multiplications in the base field $\mathbb{F}_p$ as discussed next.

Let $\mathbb{F}_{q=p^2} = \mathbb{F}_p[i]/(i^2+1)$, where i^2+1 is an irreducible binomial[3] in $\mathbb{F}_p[i]$. The field elements $a, b \in \mathbb{F}_q$ can be written as $a = a_0 + a_1 \cdot i$ and $b = b_0 + b_1 \cdot i$. Using a Karatsuba approach the field multiplication $c = a \cdot b = c_0 + c_1 \cdot i$ can be computed as,

$$\begin{aligned} c_0 &= a_0 \cdot b_0 - a_1 \cdot b_1, \\ c_1 &= (a_0 + a_1)(b_0 + b_1) - a_0 \cdot b_0 - a_1 \cdot b_1, \end{aligned} \tag{10.10}$$

which can be performed at a cost of three integer multiplications, five integer additions and two modular reductions. Similarly the field squaring operation, for $a \in \mathbb{F}_q$, is computed as $a^2 = (a_0 + a_1 \cdot i)^2 = (a_0 + a_1) \cdot (a_0 - a_1) + 2a_0a_1 \cdot i$ at a cost of two integer multiplications, two modular reductions and three integer additions.

In order to get a fast implementation of the field multiplication and squaring operations in the base field $\mathbb{F}_p$, we used the techniques explained in §3.3 taking advantage of the novel instruction sets recently introduced in modern Intel and AMD processors, which have been especially designed for achieving a faster execution of multi-precision integer arithmetic. For the sake of concreteness our description will be mainly focused on the popular prime modulus p_{CLN} striving to exploit its very special form, which allows us to use the technique introduced in §3.3.4.1.

10.4.1. Exploiting the special form of the SIDH moduli

The main algorithmic idea of the REDC multi-precision version shown in Algorithm 5 is that of calculating a quotient q that makes $T + q \cdot p$ divisible by 2^w. This allows to update T as $(T + q \cdot p)/2^w$, which implies that at each iteration of Algorithm 5, the size of T is decreased by one word. Notice that the value of q in step 3 directly depends on the updated value of T. This situation is commonly known as a loop-carried dependency that prevents a further parallelization of Algorithm 5. Therefore, this procedure can only process one $q \cdot p$ product per iteration with an associated cost of one $1 \times n$ digit multiplication.

Nevertheless, when Algorithm 5 is executed using a λ-Montgomery-friendly modulus the loop-carried dependency can be avoided in up to λ iterations of the main loop. To see how

[3] Always true whenever $p \bmod 4 = 3$.

this trick works notice that in step 2 the value t is assigned with the least significant word of T. If p is a λ-Montgomery-friendly modulus and $p' = 1$. This implies that in step 3 there is no multiplication to be performed but a simple assignment $q = t$. It follows that step 4 can be computed as,

$$\frac{(T + q \cdot p)}{2^w} = \frac{(T + t \cdot p)}{2^w} = \frac{(T + t \cdot (p-1) - t)}{2^w}.$$

Since $p + 1$ can be represented as $(p_{n-1}, \ldots, p_0)$, where $p_i = 0$ for $0 \leq i < \lambda$, the λ least-significant words of the product $t \cdot (p + 1)$ are all equal to zero. This implies that we can compute step 4 of Algorithm 5 by multiplying t with the $n - \lambda$ most-significant words of $p+1$, adding the resulting product with T, and completely ignoring the least-significant word of this computation. In other words,

$$\frac{(T + q \cdot p)}{2^w} = \frac{(T + t \cdot (p+1) - t)}{2^w} = \left\lfloor \frac{T + t \cdot (p+1)}{2^w} \right\rfloor.$$

We observe that since the least-significant words of T are not modified, then the value of q for the next iteration can be obtained in advance, thus breaking the loop-carried dependency. In general, for a λ-Montgomery-friendly prime one can calculate the value of q for λ iterations without the knowledge of the values that T will be getting in those iterations.

We illustrate in Figure 10.2 the execution of the multi-precision REDC algorithm using as a modulus $p = p_{\text{CLN}}$, which is a 5-Montgomery-friendly modulus that has a size of $n = 12$ words. At the i-th iteration of the REDC algorithm an updated value of q is calculated and multiplied by $p + 1$. Then, the result is added to T (on top), and the least-significant word of T is removed. After n iterations, the final result is stored in C, which is composed of the twelve most-significant words of T.

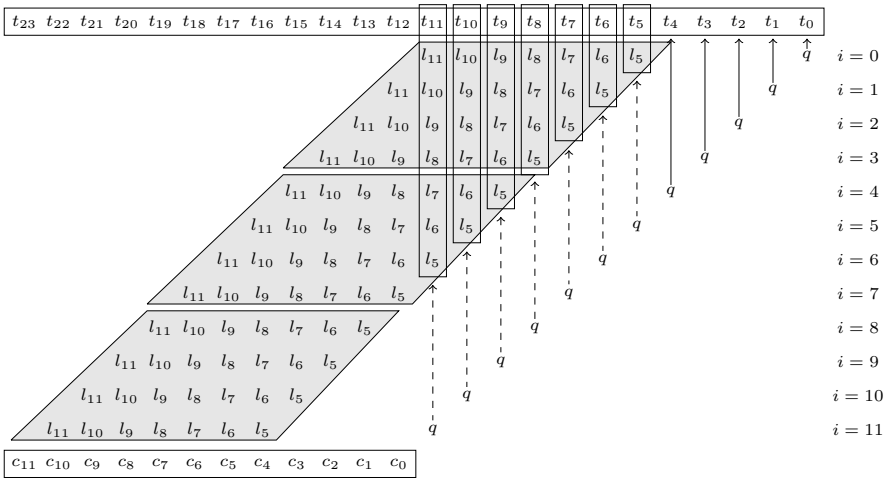


Figure 10.2: Multi-precision execution of $C = \text{REDC}(T)$ for $n = 12$. Given the input $T = (t_0, \ldots, t_{23})$, REDC calculates n times the product $q \cdot (p + 1)$, where p is a 5-Montgomery-friendly prime. This implies that $p + 1$ can be expressed as $(p_{11}, p_{10}, p_9, p_8, p_7, p_6, p_5, 0, 0, 0, 0, 0)$. In order to update T, at each iteration the partial products $l_j = p_j q$, for $5 \leq j < 12$ are computed. The dependency for calculating q at each iteration is highlighted with arrows. Notice that the first five values of q only depends on the unmodified value of T (this fact is represented by solid arrows).

The vertical arrows denote the dependencies associated to the computation of q. Since p is a 5-Montgomery-friendly prime, from the first to the fifth iteration q only depends on

the original value of T. However in the sixth iteration q depends on T and on the value of $q \cdot (p+1)$ from the first iteration (this fact is highlighted by the dashed arrows and the vertical rectangles). As can be seen, no loop-carried dependencies appear during the first five iterations, allowing to compute up to five $q \cdot (p+1)$ products before updating T becomes necessary. These products can be viewed as a 7×4 digit multiplication followed by a 64-bit left-shift in the case of p_{CLN} modulus (these operations are highlighted in the shadowed area). Thus, after performing three 7×4 digit multiplications and three 64-bit left-shifts the modular reduction is completed. This latter observation inspired us to come out with the modified version of Algorithm 5 shown in Algorithm 39.

Algorithm 39 Modified modular reduction algorithm for a λ-Montgomery-friendly modulus.

Input: T, an integer such that $0 \leq T < Rp$, $R = 2^{wn}$, p is a λ-Montgomery-friendly modulus, and $0 < B \leq \lambda$.

Output: C, an integer such that $C = TR^{-1} \mod p$.

1: $\lambda_0 \leftarrow \lfloor n/B \rfloor$
2: $\lambda_0' \leftarrow n \mod B$
3: $M \leftarrow \lfloor (p+1)/2^{\lambda \cdot w} \rfloor$
4: **for** $i \leftarrow 1$ **to** λ_0 **do**
5: $\quad Q \leftarrow T \mod 2^{B \cdot w}$
6: $\quad T \leftarrow \lfloor T/2^{B \cdot w} \rfloor + 2^{(\lambda - B) \cdot w} Q \cdot M$
7: **end for**
8: **if** $\lambda_0' \neq 0$ **then**
9: $\quad Q \leftarrow T \mod 2^{\lambda_0' \cdot w}$
10: $\quad T \leftarrow \lfloor T/2^{\lambda_0' \cdot w} \rfloor + 2^{(\lambda - \lambda_0') \cdot w} Q \cdot M$
11: **end if**
12: $C \leftarrow T$
13: **if** $C \geq p$ **then**
14: $\quad C \leftarrow C - p$
15: **end if**
16: **return** C

The modified REDC procedure is presented in Algorithm 39. Given a λ-Montgomery-friendly modulus p, the number of iterations without loop-carried dependency can be chosen as, $0 < B \leq \lambda$ (this value B is equals four for the example in Figure 10.2). In step 5 of Algorithm 39 the value $Q = T \mod 2^{B \cdot w}$ is computed. Thereafter Q is multiplied by the $\lambda_1 = n - \lambda$ most-significant words of $p+1$ given by $M = \lfloor (p+1)/2^{\lambda \cdot w} \rfloor$. It is noticed that Q is a B-digit number. Hence, $Q \cdot M$ can be calculated as a $B \times \lambda_1$ digit product. At this point, the value $2^{(\lambda - B) \cdot w} Q \cdot M$ is added to $\lfloor T/2^{B \cdot w} \rfloor$. Hence, the B least-significant digits of T are discarded. Repeating this procedure $\lambda_0 = \lfloor \frac{n}{B} \rfloor$ times, the size of T is decreased by $B\lambda_0$ words. In the case that $n \mod B = 0$, the modular reduction has been completed. Otherwise, $\lambda_0' = n \mod B$ digits of T must still be reduced by applying one extra iteration using $B = \lambda_0'$ (lines 8-11 of Algorithm 39).

Algorithm 39 shares similar ideas as the ones presented by Bos and Friedberger in [25]. In particular, the strategy named *shifted* (SH) in [25] that allows to trade multiplications by right-shift operations can be easily adapted to our setting.

10.4.1.1. Correctness

From the previous discussion, it follows that the first B iterations of the multi-precision REDC Algorithm 5 do not show loop-carried dependencies. Thus, the first B values of q can be computed at once, by setting $Q = T \mod 2^{B \cdot w}$ (line 5 of Algorithm 39). Then T is updated with $\lfloor T/2^{\lambda_0' \cdot w} \rfloor + 2^{(\lambda - \lambda_0') \cdot w} Q \cdot M$. In this way the B least-significant words of

T are removed. After λ_0 iterations, the size of T will be reduced $B\lambda_0$ words exactly as it would happen after $n - \lambda'_0$ iterations of a regular execution of the multi-precision REDC algorithm. Whenever $B \nmid n$, one additional iteration is processed to reduce the remaining λ'_0 digits. Since Algorithm 39 performs the same reduction that Algorithm 5 computes, the final conditional subtraction step of lines 13-15 is also required.

10.4.1.2. Case Study $p_{\mathbf{CLN}} = 2^{372}3^{239} - 1$

Here we focus our attention to the problem of fine-tuning the design parameters of Algorithm 39 when dealing with the modulus p_{CLN}.

We performed several experiments with different values of the parameter B trying to determine the optimal value of this parameter that yields the modular reduction with the smallest latency. In order to provide a fair comparison, we performed the implementation of this operation using three different variants, which mainly differ in the type of x86_64 arithmetic instructions that were used. The benchmarked timings obtained from our experiments are reported in Table 10.3. It can be seen that the best results were obtained using the combination of `MULX` and `ADCX/ADOX` instructions and setting $B = 4$, along with the shifted technique, this latter technique was proposed in [25]. Using this design choice, the modular reduction has a cost of three 6×4 64-bit multiplications, three 52-bit right-shifts over 10-word operands, and three additions over 11-word operands. The measured latency is of 156 clock cycles.

Ref.	B	Instr. Set	Operation Counts				Clock Cycles
			Mul	Add	Mov	Other	
This work	1	mul/adc	84	251	204	8	281
		mulx/adc	84	191	24	8	232
		mulx/adx	84	191	24	8	230
	2	mul/adc	84	289	207	10	244
		mulx/adc	84	257	27	10	208
		mulx/adx	84	149	27	16	187
	3	mul/adc	84	301	210	10	227
		mulx/adc	84	281	34	10	210
		mulx/adx	84	137	34	18	193
	4	mul/adc	84	307	210	10	218
		mulx/adc	84	292	36	10	191
		mulx/adx	84	130	42	17	162
	4 + (sh)	mul/adc	72	265	186	46	204
		mulx/adc	72	253	36	46	189
		mulx/adx	**72**	**118**	**36**	**55**	**156**
[25]	1	mul/adc	84	332	157	41	254
	2	mul/adc	84	358	202	61	275
	1+(SH)	mul/adc	72	299	223	86	240

Table 10.3: Performance comparison of different modular reduction algorithms. For Algorithm 39, the admissible values of B for the prime $p_{\mathrm{CLN}} = 2^{372}3^{239} - 1$ were measured. The timings are reported in clock cycles measured on a Skylake micro-architecture. SH stands for the *shifted* technique as proposed in [25].

Our fastest modular reduction timing (reported in Table 10.3) is more efficient by a factor 1.6 than the one achieved in [25] with $B = 1$, which corresponds to the modular reduction based in the product scanning multiplication as presented in Costello *et al.* [51]. Moreover, we obtained a modular reduction that is faster by a factor 1.5 than the one reported in [25] with $B = 1$ and the shifted technique (SH). This latter result somewhat contradicts the conjecture that a value $B > 1$, may lead to a lower performance than the one associated with the choice $B = 1$, adopted by Bos and Friedberger in [25].

10.5. Implementation and benchmark results

We benchmarked our software on an Intel Core i7-4770 processor supporting the Haswell micro-architecture and on an Intel Core i7-6700K processor that supports the Skylake micro-architecture. To guarantee the reproducibility of our measurements, the Intel Hyper-Threading and Intel Turbo Boost technologies were disabled. Our source code was compiled using the GNU C Compiler (gcc) v6.1.0 with the `-O3` optimization flag and using the options `-mbmi2 -fwrapv -fomit-frame-pointer` and `-mbmi2 -madx -fwrapv -fomit-frame-pointer` for the Haswell and Skylake micro-architectures, respectively. Our code is available at: [http://github.com/armfazh/flor-sidh-x64].

10.5.1. Related works

Due to the novelty of the SIDH protocol only a few software and hardware implementations have so far been reported. Several of these implementations use different elliptic curve parameters, which makes it difficult to come out with a fair comparison. The publicly-available implementation of Costello *et al.* [51] is a portable software library called SIDH v2.0. This library includes optimized 64-bit code for field arithmetic, public key compression algorithms and an instantiation of the Diffie-Hellman protocol. SIDH v2.0 is widely considered the state-of-the-art software library for implementing the SIDH protocol. Other SIDH publicly available software libraries include [54, 8]. In [54], De Feo reports an implementation of the SIDH protocol supporting several prime sizes [54]. His implementation relies on the GMP library [78] as a modular arithmetic back-end. The implementation by Azarderakhsh *et al.* [8] is also publicly available. However, the performance of this library is significant slower than the library presented in [51].

In this chapter, we rely on the software library of Costello *et al.* [51], since it is the fastest one reported in the open literature. Further, in order to report a more complete picture of the SIDH protocol acceleration provided by the techniques presented in this chapter, we plugged-in our elliptic curve and field arithmetic functions in that library.

10.5.2. Prime field arithmetic

In Table 10.4 and Table 10.5, the running time of relevant prime field and elliptic curve operations for the Haswell and Skylake micro-architectures are reported.

Comparing with the implementation of Costello *et al.*, the multiplication in the quadratic extension field $\mathbb{F}_{p^2}$, which is a performance-critical operation, was consistently accelerated by a factor 1.32-1.34 speedup in both platforms. This improvement produces an immediate acceleration of all elliptic curve operations, yielding a factor 1.13-1.25 speedup in the Haswell micro-architecture. For Skylake, the impact of our implementation is higher, since our library benefits from more specialized multi-precision arithmetic instructions. In Skylake, the elliptic curve operations achieved a factor 1.14-1.32 speedup.

Domain	Operation	CLN [51]	**Our work**	AF
$\mathbb{F}_p$	Modular reduction	279	242	1.15
	Multiplication	670	605	1.11
	Squaring	724	526	1.38
	Inversion	622,761	462,099	1.35
$\mathbb{F}_{p^2}$	Multiplication	2,143	1,626	1.32
	Squaring	1,420	1,256	1.13
	Inversion	625,904	463,773	1.35
$E(\mathbb{F}_{p^2})$	Dif. Addition	10,160	8,316	1.22
	Point Doubling	12,019	9,619	1.25
	Point Tripling	24,024	19,247	1.25
	Ladder Step ($\mathbb{F}_{p^2}$)	19,715	16,123	1.22
	Ladder Step ($\mathbb{F}_p$)	7,403	6,085	1.22
	Iso. Gen. 3-degree	11,678	9,737	1.19
	Iso. Gen. 4-degree	8,174	7,252	1.13
	Iso. Eval. 3-degree	15,817	12,842	1.23
	Iso. Eval. 4-degree	21,480	17,154	1.25

Table 10.4: Timing Performance of selected base field, quadratic and elliptic-curve arithmetic operations. The last column shows the acceleration factor that our library obtained in comparison with the SIDH v2 library [51]. All timings are reported in clock cycles measured in Haswell micro-architectures.

Domain	Operation	CLN [51]	**Our work**	AF
$\mathbb{F}_p$	Modular reduction	212	156	1.36
	Multiplication	486	415	1.17
	Squaring	523	395	1.32
	Inversion	456,621	354,373	1.29
$\mathbb{F}_{p^2}$	Multiplication	1,582	1,183	1.34
	Squaring	1,026	880	1.16
	Inversion	458,706	355,889	1.29
$E(\mathbb{F}_{p^2})$	Dif. Addition	7,371	5,896	1.25
	Point Doubling	8,855	6,969	1.27
	Point Tripling	17,799	13,528	1.32
	Ladder Step ($\mathbb{F}_{p^2}$)	14,384	11,802	1.22
	Ladder Step ($\mathbb{F}_p$)	5,259	4,327	1.21
	Iso. Gen. 3-degree	8,537	6,873	1.24
	Iso. Gen. 4-degree	5,980	5,241	1.14
	Iso. Eval. 3-degree	11,864	9,369	1.27
	Iso. Eval. 4-degree	15,932	12,377	1.29

Table 10.5: Timing Performance of selected base field, quadratic and elliptic-curve arithmetic operations. The last column shows the acceleration factor that our library obtained in comparison with the SIDH v2 library [51]. All timings are reported in clock cycles measured in Skylake micro-architectures.

10.5.3. Impact of the $P + [k]Q$ optimization

We measured the performance rendered by the ladder algorithms presented in §9.3.2.1. To that end, we take as a baseline the original SIDH v2 library and plugged in our algorithms using the same prime field arithmetic interface.

The benchmarked timings are summarized in Table 10.6. In all the cases, we were able to corroborate the theoretical predictions summarized in Table 10.1. For example for the variable-point scenario, the SIDH v2 library computes the three-point ladder in 11.2×10^6 Haswell clock cycles. Our software accelerates this timing by a factor 1.38 speedup to compute the same operation. Thank to this, Alice and Bob shared-secret time performance are accelerated by around 6-7% (Table 10.7). In the case of the fixed-point scenario it can be observed that using either Algorithm 37 or Algorithm 38 our approach is ≈ 1.7 faster than the methods implemented in the SIDH v2 library. Once again these results confirm the theoretical estimates given in Table 10.1. The pre-computed look-up tables have a size of around 35 KB. This relatively moderate size permits that a large part of the look-up tables can fit in the Level-1 Data cache memory of the target platforms (which have a size of 32 KB).

Regarding side-channel protection, we want to note that the right-to-left algorithms were implemented considering classic countermeasures; for example, using a straight and a regular execution of instructions. Moreover, no secret values were used to index look-up tables or to bifurcate the execution of any function.

Scenario	Field	Haswell	Skylake	Algorithm
Fixed-point	$\mathbb{F}_{p^2}$	6.7	4.9	Method 1
		3.9	2.9	**Alg. 37 (our work)**
	$\mathbb{F}_p$	2.5	1.7	Method 3
		1.5	1.0	**Alg. 38 (our work)**
Variable-point	$\mathbb{F}_{p^2}$	11.2	8.1	3-point ladder
		8.0	5.9	**Alg. 36 (our work)**

Table 10.6: Performance comparison of different methods to compute $\chi(P + [k]Q)$. The implementation of Methods 1, 2 and 3 were taken from the SIDH-v2 library [51]. All timings are given in 10^6 clock cycles and were measured on a Haswell and on a Skylake micro-architecture.

10.5.4. Point tripling impact

Clearly, the most time consuming SIDH operation is the calculation of large-degree isogenies. In the case of Bob, this process implies to perform a large number of point tripling computations.

Our implementation of the point tripling formula proposed in §10.3 saves up to 400 clock cycles, corresponding to the difference 1M - 1S - 1A (Tables 10.4 and 10.5). This reduction in the cost of the point tripling computation yields a small but noticeable acceleration of the whole protocol. More concretely, replacing the tripling formula implemented in the SIDH v2 library by our proposed formula yields a speedup of around 1-2% in the SIDH protocol execution.

10.5.5. Performance comparison of the SIDH protocol

In Table 10.7, the running timings associated with the execution of both phases of the SIDH protocol are reported. It is noted that the achieved speedups are highly correlated with the ones obtained for the multiplication operation in the quadratic extension field $\mathbb{F}_{p^2}$. This confirms the high-impact of this operation in the performance of the whole protocol. For all of the SIDH operations, the performance measured on Skylake was between 1.38 to 1.41 times faster than the one measured on the Haswell processor (Table 10.7). This acceleration can be seen as a consequence of the higher performance achieved by the latest integer arithmetic instruction sets (which are available in Skylake but not in Haswell).

Protocol Phase		Haswell			Skylake		
		CLN [51]	**This work**	AF	CLN [51]	**This work**	AF
Key Gen.	Alice	48.3	38.0	1.27	35.7	26.9	1.33
	Bob	54.5	42.8	1.27	39.9	30.5	1.31
Shared Secret	Alice	45.7	34.3	1.33	33.6	24.9	1.35
	Bob	52.8	39.6	1.33	38.4	28.6	1.34

Table 10.7: Performance comparison of the SIDH protocol. The running time is reported in 10^6 clock cycles to compute the two phases of the SIDH protocol. Additionally, the speedup factor with respect to the SIDH v2 library [51] is also reported.

10.6. Conclusions

In this chapter we presented a number of optimizations targeting the supersingular isogeny-based Diffie-Hellman protocol. We focused our attention on optimizing both the finite field and the elliptic curve arithmetic layers.

We accelerated operations in the base field $\mathbb{F}_p$ and in its quadratic extension $\mathbb{F}_{p^2}$, using the newest arithmetic instruction sets available in modern Intel processors and also, by taking advantage of the special form of the p_{CLN} prime chosen in [51]. The combination of these techniques allowed us to compute finite field arithmetic about 1.38 faster than the performance obtained by running the library of [51] on the same Intel processor architectures.

Building on [138], we adapted a right-to-left Montgomery ladder variant to the context of the SIDH protocol, where the elliptic curve operation $P + [k]Q$ must be computed. In the case when the involved points are known in advance, our algorithm enables for the first time the usage of precomputed look-up tables to accelerate the SIDH key generation phase. We also presented an improved formula for elliptic curve point tripling. Our formula permits to save one multiplication at the cost of one extra squaring and one extra addition performed in the quadratic extension $\mathbb{F}_{p^2}$.

Executing our software on an Intel Skylake Core i7-6700 processor we are able to compute the two phases of the SIDH protocol, namely, key generation and shared secret, in less than 51.8 and 59.1 millions of clock cycles for Alice's and Bob's computations, respectively. This gives us a 1.33 times speedup against the software implementation of Costello *et al.*

Chapter 11

A parallel approach for the Supersingular Isogeny Diffie-Hellman protocol

In the same way as in the previous section, our objective is to improve the running time performance of the SIDH protocol developed by David Jao *et al.* [55]. Because, the latency associated to SIDH is higher in comparison with other post-quantum proposals. In order to reduce its running time, in this section we present a fast SIDH variant called eSIDH that uses primes of the form $p = 2^{e_A} l_B^{e_B} l_C^{e_C} f - 1$. This new version allow us to take advantage of software parallelism. Besides, we propose some improvements that allow us reduce the cost of construction and evaluation of isogenies, which contribute to decrease the overall performance of SIDH.

11.1. Introduction

Nowadays, the isogeny-based cryptosystems have become popular among the post-quantum candidates. A strong isogeny-based competitor is the SIDH cryptosystem proposed by David Jao and Luca de Feo [55] described in the previous section, whose security is based on the CSSI problem which is hard to compute in classical and quantum computers [3]. The main operation in both cryptosystems is the computation of isogenies between curves, when the kernel of the isogeny is known. This operation can be performed thanks to Velú [161], who developed a formula to compute isogenies between Weierstrass curves using a given subgroup which will be the kernel of the isogeny. As far as we know, there are few works focused on compute isogenies in other curve models such as the Moody studying Huff and the Edwards models [133]. Costello and Hisil [49] developed a general formula to compute odd-degree isogenies in Montgomery curves which is basically the main core of this section.

In order to reduce the running time of the SIDH protocol we focused on performance-critical operations, such as the elliptic curve scalar multiplication $P + [k]Q$ required by the two main phases of the SIDH protocol, the isogeny constructions and evaluations, and their inherent field arithmetic operations. With that in mind, our main contributions for accelerating the performance of the SIDH protocol can be summarized as follows:

- We propose a particular way to perform SIDH protocol using a non-prime power degree isogenies in the Bob's side. This construction of the SIDH protocol is called eSIDH.

This variant at first sight looks more costly than the original SIDH, but by improve the isogeny constructions and evaluations and using software parallelism, it is possible achieve a considerable speedup.

- We propose λ-Montgomery-friendly primes as an alternative to those recommended in the current state-of-the-art, which maintain the same security level. This new primes allow us to perform a better modular reduction which yields a better performance of whole protocol.

11.2. Extended SIDH

Here, we present a novel technique that allows us to improve the Bob's computations in the SIDH protocol [55], but without altering the format and length of the public keys. The main core of this technique is the use of what we call a *composite-isogeny* which is a non-prime power degree isogeny.

The eSIDH domain parameters are a supersingular elliptic curve $E/\mathbb{F}_{p^2}$, where p is a large prime of the form

$$p = (l_A)^{e_A}(l_B)^{e_B}(l_C)^{e_C} f \pm 1,$$

where l_A, l_B and l_C are different small prime numbers; e_A , e_B and e_C are positive integers such that $l_A^{e_A} \approx (l_B^{e_B} l_C^{e_C})$; and f is a small cofactor.

In addition to the form of the prime p, the main difference of the eSIDH in comparison with the Jao's *et al.* protocol is that Bob now needs to compute an isogeny of degree $(l_B)^{e_B}(l_C)^{e_C}$. However, if Bob choose public points P_{BC} and Q_{BC} such that $E[l_B^{e_B} l_C^{e_C}] = \langle P_{BC}, Q_{BC} \rangle$, then the protocol is the same as that devised by Jao *et al.* In the following, we propose three ways to compute this isogeny, which are denoted as eSIDH, PeSIDH and CRTeSIDH.

11.2.1. eSIDH

At first glance if $e_B = e_C$ then Bob should compute a chain of $(l_B l_C)$-degree isogenies, but this could be different in practice. For the sake of simplicity let us assume that $e_C < e_B$ and $k = e_B - e_C$. With these settings, Bob can compute an l_B^k-isogeny ϕ_{B_0} followed by an $(l_B l_C)^{e_C}$-isogeny ϕ_{B_1} or vice versa. At this point the protocol turns a little bit complicated, because Bob needs to calculate the kernel of ϕ_{B_0} and once Bob computes this l_B^k-isogeny, then he needs to compute kernel of the $(l_B l_C)^{e_C}$-isogeny ϕ_{B_1}.

With the aim of generating the kernel of his secret isogeny ϕ_B, Bob choose a random integer $n_B \in [1,\ l_B^{e_B} l_C^{e_C}]$ and computes the point $R = P_B + [n_B]Q_B$ whose order is $(l_B^{e_B} l_C^{e_C})$. After that, he calculates $R_B = [(l_B l_C)^{e_B}]R$ whose order is l_C^k. In this way, the isogeny ϕ_B in Figure 9.1 is generated as follows: Bob calculates the point R_B and constructs the isogeny ϕ_{B_0}, then he evaluates $\phi_{B_0}(R)$ that produces a point of order $(l_B l_C)^{e_C}$ that is the kernel of ϕ_{B_1}, and finally $\phi_B = \phi_{B_0} \circ \phi_{B_1}$. (the same procedure is used to compute ϕ'_B in Figure 9.1).

We sketch a brief analysis on the cost of performing the above procedure. As we see, it is necessary to perform two scalar multiplications one to get the value of R and other to get R_B. The cost of this scalar multiplication is shown in next using as unit of measurement:

- A *full-ladder* if the scalar has about $\log(p^{1/2}) \approx \log(l_A^{e_A}) \approx \log(l_B^{e_B}\ l_C^{e_C})$,
- A *demi-ladder* if the scalar has about $\log(p^{1/4}) \approx \log(l_B^{e_B}) \approx \log(l_C^{e_C})$.

Thus, the cost associated to compute the point R is of a full-ladder, while the cost of compute R_B is a little over a demi-ladder. More precisely, the cost of compute R_B is about a full-ladder *minus* a ladder of $\log(l_B^k)$ bits. One way of saving operations during

the computation of the chain of $(l_B l_C)$-degree isogenies, is changing the order in that these isogenies are computed, *i.e.* computing the $(l_B l_C)^{e_C}$-isogeny first and then the l_B^k-isogeny, however, doing this implies that e_C evaluations of $(l_B l_C)$-degree isogenies must be performed instead of k l_B-degree isogeny evaluations in the first approach described before.

11.3. Parallel eSIDH

The new construction of the SIDH protocol shown in previous section looks very simple. In the sense that (almost) follows the structure of the protocol devised by Jao *et al.* [55]. However, eSIDH allows us to improve the performance of the protocol by doing more changes in its structure. The main core of PeSIDH (Parallel eSIDH) construction is to use two private keys, and two bases instead of one in the Bob's side. Then, our new Bob's public setting will be the points P_B, Q_B, P_C, Q_C such that $\langle P_B, Q_B\rangle = E[r_B]$ and $\langle P_C, Q_C\rangle = E[r_C]$ where $r_B = l_B^{e_B}$ and $r_C = l_C^{e_C}$. With this new parameters, now we are able to sketch the PeSIDH:

- Public Parameters
  - Prime $p = (l_A)^{e_A}(l_B)^{e_B}(l_C)^{e_C} f \pm 1$,
  - Curve E_0,
  - Points P_A, Q_A, P_B, Q_B, P_C, and Q_C.
- Key Generation
  - Alice randomly choose $n_A \in [1,\ l_A^{e_A}]$ and computes $R_A = P_A + [n_A]Q_A$. Then, she computes the public curve E_A and the isogeny $\phi_A : E_0 \to E_A$ such that $\ker(\phi_A) = \langle R_A\rangle$ and sends to Bob $\phi_A(P_B)$, $\phi_A(Q_B)$, $\phi_A(P_C)$ and $\phi_A(Q_C)$.
  - Bob randomly chooses $n_B \in [1,\ r_B]$ and $n_C \in [1,\ r_C]$, then he computes $R_B = P_B + [n_B]Q_B$ and $R_C = P_C + [n_C]Q_C$. After that, Bob computes the private isogenous curve E_B and the isogeny $\phi_B : E_0 \to E_B$ such that $\ker(\phi_B) = \langle R_B\rangle$. Finally, Bob computes the public curve E_{BC} and the isogeny $\phi_{BC} : E_B \to E_C$ such that $\ker(\phi_{BC}) = \langle \phi_B(R_C)\rangle$, and send to Alice $\phi_{BC}(P_A)$ and $\phi_{BC}(Q_A)$.
- Key Agreement
  - Alice *recovers* her private key as $R'_A = \phi_{BC}(P_A) + [n_A]\phi_{BC}(Q_A)$ and using it computes the curve E_{BCA}.
  - Bob *recovers* his both private keys as $R'_B = \phi_A(P_B) + [n_B]\phi_A(Q_B)$ and $R'_C = \phi_A(P_C) + [n_C]\phi_A(Q_C)$. Then, he computes the private curve E_{AB} and the isogeny $\phi'_B : E_A \to E_{AB}$ such that $\ker(\phi'_B) = \langle R'_B\rangle$. At the end, Bob computes the curve E_{ABC} using $\phi'_B(R'_C)$ as kernel of the isogeny $\phi'_C : E_{AB} \to E_{ABC}$.
  - The Secret shared is $j(E_{ABC}) = j(E_{BCA})$.

Now we have four public points for Bob instead of two, this could be a problem because this implies a bigger size in the public settings at beginning and in the middle part of the protocol. We solved this issue by fixing the points

$$S = P_B + P_C, \qquad T = Q_B + Q_C.$$

This overcome the size problem because Alice just needed to evaluate two points for Bob, but brings a new problem that we call the *Key-Recovering* problem. This problem consist of recovering the image of the Bob's isogeny kernel through the Alice isogeny ϕ_A, by using

only the knowledge of $\phi_A(S)$ and $\phi_A(T)$. Just looking at the order of the points involved, we can solve this problem in an easy (but not cheap) way. We have the following equations

$$[r_B]S = [r_B]P_C, \quad [r_C]S = [r_C]P_B, \quad [r_B]T = [r_B]Q_C, \quad [r_C]T = [r_C]Q_B.$$

Then making use of those equalities we have that:

$$[r_C](S + [n_B]T) = [r_C](P_B + [n_B]Q_B) \qquad [r_B](S + [n_C]T) = [r_B](P_C + [n_C]Q_C).$$

Using the fact that if R is an n-order point and is the kernel of an isogeny, then for an integer m such that $\gcd(m, n) = 1$ we have that $[m]R$ generates *the same* isogeny (up to isomorphism). Then, we are able to recover Bob's private keys or more precisely, a multiple of both keys which generate the same isogenies (up to isomorphism).

As we can see, this key-recovering method is effective but maybe not be efficient, because in order to recover the key we need 4 demi-ladders making this more costly than the eSIDH approach. On the other hand, we can see that computing $R'_B = [r_C](S + [n_B]T)$ is independent of computing $R'_C = [r_B](S + [n_C]T)$. This fact, bring us the possibility of compute R'_B and R'_C in parallel. If we consider that invoking parallelism is cost-free, then computing R'_B and R'_C has the cost of two demi-ladders, which is approximate the cost of one full-ladder. In the same way, for the Key agreement phase we can compute R_B and R_C in parallel at a cost of one demi-ladder, which directly improves the cost of computing the kernel in the original SIDH [55]. Figure 11.1 shown how this variant of eSIDH could be performed. Here we only need two cores to perform the computations, however, in §11.5.2 we show other *places* which could also benefit by parallelism.

Alice		**Bob**
Choose $n_A \in \mathbb{Z}_{r_A}$		Choose $n_B \in \mathbb{Z}_{r_B}$ and $n_C \in \mathbb{Z}_{r_C}$
$R_A = P_A + [n_A]Q_A$		$R_B = P_B + [n_B]Q_B$, $R_C = P_C + [n_C]Q_C$
$E_A,\ \phi_A = \nu(E, R_A)$		$E_B, \phi_B = \nu(E, R_B)$
		$E_{BC}, \phi_C = \nu(E_B, \phi_B(R_C))$
	$E_A,\ \phi_A(S),\ \phi_A(T)$ $\longrightarrow$	
	$\longleftarrow$ $E_{BC},\ \phi_C(\phi_B(P_A)),\ \phi_C(\phi_B(Q_A))$	
$R'_A = \phi_C(\phi_B(P_A)) + [n_A]\phi_C(\phi_B(Q_A))$		$R'_B = [r_C](\phi_A(S) + [n_B]\phi_A(T))$
$E_{BCA}, \phi'_A = \nu(E_{BC}, R'_A)$		$R'_C = [r_B](\phi_A(S) + [n_C]\phi_A(T))$
		$E_{AB}, \phi'_B = \nu(E_A, R'_B)$
		$E_{ABC}, \phi'_C = \nu(E_{AB}, \phi'_B(R'_C))$
$s_A = j(E_{BCA})$		$s_B = j(E_{ABC})$
	$s_A = s_B$	

Figure 11.1: Parallel version of the eSIDH protocol called PeSIDH. Here ν represent the Velu's formula whose entries are an elliptic curve E and a point $P \in E$ such that generates the kernel of the output isogeny. Notice that, computing R_B, R_C, R'_B and R'_C can be performed in parallel.

11.3.1. eSIDH meets Chinese Remainder Theorem

In the PeSIDH construction we exploited the usage of parallelism to achieve a faster kernel generation and to maintain the cost same to the original key agreement phase of SIDH. We are aware that not all devices have two cores, having this in mind we propose a

better single-core version, more precisely, an eSIDH construction whose cost is less than 2 full-ladders in the key agreement phase.

Our strategy is to use the ancient Chinese Remainder Theorem (Theorem 3.1) to modify the private key generation phase. Unlike the previous approach where we only choose a pair of random values for Bob to compute the kernels, in our CRT based construction it is necessary to perform some calculations on those random values. Thus, the secret key used in the key generation phase and the secret key used in the key agreement phase must be constructed in the following special way:

- Randomly choose $n_B \in [1, r_B]$ and $n_C \in [1, r_C]$ such that $\gcd(n_B, r_C) = \gcd(n_C, r_B) = 1$.
- Compute $\hat{n}_B = n_B^{-1} \mod r_C, \quad \hat{n}_C = n_C^{-1} \mod r_B$
- Finally compute the private keys
  - $(\bar{n}_B = n_B \cdot \hat{n}_B \mod r_B,\ \bar{n}_C = n_C \cdot \hat{n}_C \mod r_C)$ for the key generation phase,
  - $n_{BC} = n_B \cdot \hat{n}_B \cdot n_C \cdot \hat{n}_C \mod (r_B \cdot r_C)$ for the key agreement phase.

Remark 11.1. *By construction and by the Chinese Remainder Theorem(CRT) we have that* $n_{BC} \equiv \bar{n}_B \mod r_B$ *and* $n_{BC} \equiv \bar{n}_C \mod r_C$.

For the key generation phase we should compute the points $R_B = P_B + [\bar{n}_B]Q_B$ and $R_C = P_C + [\bar{n}_C]Q_C$, whose orders are r_B and r_C, respectively. Once we compute both points, it is possible compute the isogenies ϕ_B and ϕ_C, such that $\phi_B = \langle R_B \rangle$ and $\phi_C = \langle \phi_B(R_C) \rangle$. As we can observe, in this part of the protocol we have computed two demi-ladders, just like in the previous approach.

The Bob's Key-recovering problem in this case is a little bit different because we want to use this new key-style to save some operations. For this aim we proposed a solution based in the following theorem.

Theorem 11.1. *Let* P_B, Q_B, P_C, Q_C, $\bar{n}_B$, $\bar{n}_C$, n_{BC}, R_B, R_C, S *and* T *as before, then* $[r_C]R_B = [r_C](S + [n_{BC}]T)$ *and* $[r_B]R_C = [r_B](S + [n_{BC}]T)$.

Proof. By straightforward computation we have that:

$$\begin{aligned}
[r_C](S + [n_{BC}]T) =& [r_C](P_B + P_C + [n_{BC}]Q_B + [n_{BC}]Q_C) \\
=& [r_C](P_B + [n_{BC}]Q_B) \\
=& [r_C](P_B + [n_{BC} \mod r_B]Q_B) \\
=& [r_C](P_B + [\bar{n}_B]Q_B) \\
=& [r_C]R_B.
\end{aligned}$$

The proof for $[r_B]R_C = [r_B](S + [n_{BC}]T)$ can be realized in an analogous way. □

Using the Theorem 11.1 we can solve the Key-Recovering problem setting $R'_B = [r_C](\phi_A(S) + [n_{BC}]\phi_A(T)) = \phi_A([r_C]R_B)$ and $R'_C = [r_B](\phi_A(S) + [n_{BC}]\phi(T)) = \phi_A([r_B]R_C)$. Despite the fact that we solve the problem, we compute one full-ladder and two demi-ladders which is more costly than the the first approach. However, we can exploit one more time the CRT property mentioned in the Remark 11.1 to overcome this overhead.

Theorem 11.2. *Fixing* $R' = \phi_A(S) + [n_{BC}]\phi_A(T)$ *and* R'_C *as before, the isogeny* ϕ'_C *such that* $\ker(\phi'_C) = \langle R'_C \rangle$ *produces a point* $\phi'_C(R')$ *with order* r_B, *moreover* $\phi'_C(R') = \phi'_C((\phi_A(R_B))$.

Proof. By straightforward computation and using Theorem 11.1 we get that

$$\begin{aligned} R' =& \phi_A(S + [n_{BC}]T) \\ =& \phi_A(P_B + [n_{BC}]Q_B + P_C + [n_{BC}]Q_C) \\ =& \phi_A(R_B + R_C). \end{aligned}$$

Then $\phi'_C(R') = \phi'_C((\phi_A(R_B))$ which is a point with order r_B. □

Using Theorem 11.2 we can save one demi-ladder because the evaluation of ϕ'_C performs it automatically. Now, we have the tools to describe how this changes can be applied to the eSIDH protocol, in Figure 11.2 we shown this construction, that we call CRTeSIDH, which could be used in practice in a similar way as the SIDH protocol.

Alice		**Bob**
Choose $n_A \in \mathbb{Z}_{r_A}$		Choose $n_B \in \mathbb{Z}_{r_B}$ and $n_C \in \mathbb{Z}_{r_C}$
$R_A = P_A + [n_A]Q_A$ $E_A,\ \phi_A = \nu(E, R_A)$		Such that $(n_B, r_C) = (n_C, r_B) = 1$. Compute: $\hat{n}_B = n_B^{-1} \mod r_C$, $\hat{n}_C = n_C^{-1} \mod r_B$, $n_{BC} = n_B \cdot \hat{n}_B \cdot n_C \cdot \hat{n}_C \mod (r_B \cdot r_C)$. $\bar{n}_B = n_B \cdot \hat{n}_B$ and $\bar{n}_C = n_C \cdot \hat{n}_C$. $R_B = P_B + [\bar{n}_B]Q_B\ ,\ R_C = P_C + [\bar{n}_C]Q_C$
		$E_B, \phi_B = \nu(E, R_B)$
		$E_C, \phi_C = \nu(E_B, \phi_B(R_C))$
	$E_A,\ \phi_A(S),\ \phi_A(T) \longrightarrow$	
	$\longleftarrow E_C,\ \phi_C(\phi_B(P_A)),\ \phi_C(\phi_B(Q_A))$	
$R'_A = \phi_C(\phi_B(P_A)) + [n_A]\phi_C(\phi_B(Q_A))$ $E_{BA}, \phi'_A = \nu(E_B, R'_A)$		$R' = (\phi_A(S) + [n_{BC}]\phi_A(T)),\ R'_B = [r_C]R'$ $E_{AB}, \phi'_B = \nu(E_A, R'_B)$ $E_{ABC}, \phi'_C = \nu(E_{AB}, \phi'_B(R'))$
$s_A = j(E_{BA})$		$s_B = j(E_{ABC})$

Figure 11.2: CRTeSIDH description. The parameters are the same as in the Theorems 11.1 and 11.2. The function ν is only to represent the Velu's isogeny construction which given a curve and a subgroup return an isogenous curve and an isogeny.

11.4. Improving the construction and evaluation of isogenies

In this section we propose two strategies that allow reduce the cost of compute and evaluate isogenies. We based these strategies in the work of Meyer *et al.* [125, 126], where the authors proposed to switch between Montgomery and Edwards curves whenever it is possible to obtain a speedup in the computations. In the following we present how to exploit the Edwards isogeny-construction and some tweaks on it.

11.4.1. Tweaks for Isogeny construction

In §9.2 we estimate the complexity of computing the image of the curve using Moody's formula, but we can make some tiny tweaks in order to reduce the operations count (ignoring the obvious parallel approach of compute a' and d'). We have two basic cases divided by the

isogeny degree, in the following the degree is expressed as $s = 2l + 1$. First, if $s > 8$ then we simply can compute

$$a' = (B_z a)^8 a^{s-8} \quad \text{and} \quad d' = (B_y d)^8 d^{s-8},$$

saving 6S, more precisely changing 6S by 2M. In this way, the new complexity is about $(2l + 2 + \log(s-8)/2)\text{M} + (6 + \log(s-8))\text{S}$; The other case is for odd-degree isogenies with $s < 8$, in this case we only have 3 cases but we will ignore the case when $s = 3$ because the particular formula in Montgomery curves [49] is better. For $s = 5$ we can compute

$$a' = (B_z^2 a)^4 a \quad \text{and} \quad d' = (B_y^2 d)^4 d,$$

and for $s = 7$ we have

$$a' = (B_z a)^8 d \quad \text{and} \quad d' = (B_y d)^8 a.$$

In both cases the cost of compute a' and d' is 6S + 4M.

From §9.2.2 we can also observe that given a twisted Edwards curve $E_{a,d}$, there exists the following correspondence with its equivalent Montgomery Curve $E_{A,C}$

$$A24p = A + 2C = a, \qquad A24m = A - 2C = d \quad \text{and} \quad C24 = 4C = a - d,$$

which is useful for eSIDH because through the isogeny computation it is better to maintain this constants instead of $(A : C)$ (For scalar multiplication purposes).

11.4.2. Using yDBL and yADD

Looking at the Costello-Hisil [49] formula for isogeny evaluation, we observe that for each point P_i in the Kernel we have the quantities $X_i - Z_i$ and $X_i + Z_i$ which corresponds to the Edwards YZ-coordinates (see §9.2.3). Moreover, we observed that once we *send* the kernel points to its respective Edwards coordinates is not necessary turning back to Montgomery. Then, we can easily modify xDBL into an yDBL version and also the xADD into a yADD version preserving all operation counts (including additions) using maps presented in §9.2.2 and by replacing $A24p$ and $C24$ in xDBL by a and $a - d$.

Therefore, if we have the kernel points in YZ-coordinates we can rewrite the Costello-Hisil evaluation formula as

$$\begin{aligned} x' &= \mathbf{x_Q} \cdot \left(\prod_{i=1}^{\ell} \left[z_Q y_{[i]P} + y_Q z_{[i]P} \right] \right)^2 \\ z' &= \mathbf{z_Q} \cdot \left(\prod_{i=1}^{\ell} \left[z_Q y_{[i]P} - y_Q z_{[i]P} \right] \right)^2 . \end{aligned}$$

Where $\mathbf{x_Q}$ and $\mathbf{z_Q}$ corresponds to XZ-coordinates of point Q (to be evaluated), and y_Q, z_Q are the Edwards YZ-coordinates of Q. Notice that x' and z' correspond to the XZ-coordinates of the image of Q, therefore, we can save $2l$ additions in the construction and evaluation of isogenies.

11.5. Implementation and benchmarks results

In this section we present some considerations for the implementation of the PeSIDH construction, we also propose new λ-Montgomery-friendly primes p that allow us to perform a better base field arithmetic. Besides, we show the timings of our software library and its behavior in comparison with the state-of-the-art works.

11.5.1. eSIDH prime Selection

The common choice of primes for SIDH are those of the form $p = 2^{e_A}3^{e_B}f - 1$. There are at least two reasons for this choice, one is the fast arithmetic that can be achieved using a specialized Montgomery reduction algorithm [62]. The second one is that there are efficient formulas to compute 4- and 3-degree isogenies [49]. In our work the primes for the PeSIDH are of the form $p = l_A^{e_A} l_B^{e_B} l_C^{e_C} f - 1$ consecuently we have many forms to choose p. However, we preserve the classical $l_A = 2$ in order to preserve the fast arithmetic. Moreover, as we have many other options for l_B and l_C, we look for primes such that if $N = \lceil \lceil \log_2(p) \rceil / 64 \rceil$ is the minimum number of 64-bit-words needed to represent p, then the value $p + 1$ have $N/2$ 64-bit-words in zero. This choice allows to achieve a better reduction algorithm using the Algorithm 39.

Now for Bob's side there exists a trade-off between the size of the base-prime (l_B and l_C in our case) and the exponent (e_B and e_C respectively) because the base-prime defines the *size of the step* and the exponent defines how many steps we must compute. Then we can make a few big steps or many small steps. In order to take advantage of the parallelism in kernel generation and key recovery, we try to balance the pair $(l_B^{e_B}, l_C^{e_C})$. Because, if $l_B^{e_B} >> l_C^{e_C}$ then the value of l_B^k is such that $\log(l_B^k) \approx \log(l_B^{e_B} l_C^{e_C})$, and then we should compute a full-ladder for kernel generation and a bit more than one full ladder for the key recovering.

For all instances used in this section, we use primes computed as $p = 2^{e_A} l_B^{e_B} l_C^{e_C} f - 1$ such that $e_A \approx \log(l_B^{e_B} l_C^{e_C})$. The value of e_A is the same as in the SIKE specifications sent to NIST and the ones proposed in [3]. While, the value $f = 2^k c$ was chosen so that k represents the value that allows contstruct $N/2$-Montgomery friendly primes. Table 11.1 shown our prime selection, which preserve the security proposed in [94] and [3] (see Table 9.1).

Our proposals	[94] proposals
$P_{508} = 2^{258}3^{74}5^{57} - 1$	$P_{503} = 2^{250}3^{159} - 1$
$P_{764} = 2^{391}3^{121}5^{78} - 1$	$P_{751} = 2^{372}3^{239} - 1$
$P_{1013} = 2^{512}3^{157}5^{108} - 1$	$P_{964} = 2^{486}3^{301} - 1$
	[3] proposals
$P_{443} = 2^{222}3^{73}5^{45} - 1$	$P_{434} = 2^{216}3^{137} - 1$
$P_{557} = 2^{280}3^{86}5^{61} - 1$	$P_{546} = 2^{273}3^{172} - 1$

Table 11.1: Our proposals for PeSIDH primes in comparison with the current state-of the art

For the primes proposed in table Table 11.1 we implemented the base field arithmetic. In Table 11.2 and Table 11.3 we shown the cost of the quadratic finite field and the elliptic curve main operations. That results were measured in an Intel core i7-6700K processor with micro-architecture Skylake, using the Clang-3.9 compiler and the flags `-Ofast -fwrapv -fomit-frame-pointer -march=native -madx -mbmi2`. On these tables we can observe that our arithmetic operations using the $N/2$-Montgomery-friendly primes have better performance that those proposed in [94] and [3].

11.5.2. Parallelization of large-degree isogeny computation

On our eSIDH scenario, we need to compute 2^{e_2}- , 3^{e_3}- and 5^{e_5}-degree isogenies. In order to get an efficient computation of those isogenies, it is possible to use an efficient dynamic programming strategies as in [51, 94]. The main idea of using those strategies is to keep a points list L which will be evaluated through the d-degree isogenies to get an d-order point *easily*. The number of points in the lists L are very much related to the isogeny degree

Operation	[94] p_{503}	Ours p_{509}	[94] p_{751}	Ours p_{765}	Ours p_{1013}
Mult $\mathbb{F}_{p^2}$	557	500	1,054	972	1,610
Sqr $\mathbb{F}_{p^2}$	411	370	769	711	1,217
Inv $\mathbb{F}_{p^2}$	110,927	102,530	314,354	250,131	675,623
Doubling	3,404	3,111	6,168	5,789	9,412
Tripling	6,502	5,948	11,941	11,232	18,349
Quintupling	-	8,555	-	16,192	26,589
3-IsoGen	3,256	3,053	5,700	5,398	8,536
3-IsoEval	3,191	2,922	5,935	5,591	9,220
4-IsoGen	2,060	1,914	3,646	3,405	5,559
4-IsoEval	4,588	4,202	8,392	7,912	12,879
5-IsoGen	-	9,074	-	17,160	27,941
5-IsoEval	-	5,171	-	9,890	16,363

Table 11.2: Arithmetic cost comparison. Timings are reported in clock cycles measured over a Skylake processor at 4.0GHz.

Operation	[3] p_{434}	Ours p_{443}	[3] p_{546}	Ours p_{557}
Mult $\mathbb{F}_{p^2}$	509	467	774	680
Sqr $\mathbb{F}_{p^2}$	345	340	519	515
Inv $\mathbb{F}_{p^2}$	79,018	80,253	207,854	154,931
Doubling	3,084	2,920	4,627	4,145
Tripling	6,002	5,551	8,848	8,003
Quintupling	-	7,995	-	11,487
3-IsoGen	3,103	2,836	4,664	3,944
3-IsoEval	3,793	2,717	5,773	3,959
4-IsoGen	2,271	1,758	3,284	2,502
4-IsoEval	5,334	3,921	8,136	5,627
5-IsoGen	-	8,472	-	12,135
5-IsoEval	-	4,835	-	7,004

Table 11.3: Timings are reported in clock cycles measured over a Skylake processor at 4.0GHz.

and there are about $\log(d^{e_d})$, althougth usually is a bit bigger than that. In the Parallel eSIDH scenario, we exploit, if it is possible, the parallelism on software to compute and recover the private key. Besides, we parallelize in an intuitive way all point evaluations of L in parallel. These strategies together with the parallel eSIDH allows us to obtain a considerable speed up in comparison with the SIDH. In Table 11.4 we show the results of our software implementation of the PeSIDH protocol and we compare our results with the state-of-the-art works.

11.6. Conclusion

Using the techniques and algorithms presented in above sections we implemented the variant of SIDH protocolo PeSIDH, because it is the most promising version of SIDH. We can observe from Table 11.4 that our implementations achieve an acceleration factor of up to 1.73 and 1.44 against the Bob's key gaeneration and Bob's key agreement from the original

8-word primes	Alice KeyGen			Bob KeyGen			Alice KeyAgr			Bob KeyAgr		
	NP	P	AF	NP	P	AF	NP	P	AF	NP	P	AF
P503 [94]		8.24			9.13			6.70			7.71	
$2^{256} \cdot 3^{79} \cdot 5^{54} \cdot 1 - 1$	7.51	5.97	1.38	8.09	5.44	1.67	6.12	5.49	1.40	7.63	5.59	1.37
$2^{258} \cdot 3^{74} \cdot 5^{57} \cdot 1 - 1$	7.50	5.92	1.39	8.04	5.46	1.67	6.11	5.38	1.43	7.58	5.55	1.38
$2^{256} \cdot 3^{78} \cdot 7^{45} \cdot 5 - 1$	7.49	5.92	1.39	8.44	5.65	1.61	6.11	5.36	1.43	7.89	5.76	1.33

12-word primes	Alice KeyGen			Bob KeyGen			Alice KeyAgr			Bob KeyAgr		
	NP	P	AF	NP	P	AF	NP	P	AF	NP	P	AF
P751 [94]		23.72			26.70			19.38			22.81	
$2^{388} \cdot 3^{119} \cdot 5^{81} \cdot 1 - 1$	22.28	16.74	1.42	24.36	15.87	1.68	18.37	15.42	1.25	23.00	16.36	1.39
$2^{391} \cdot 3^{121} \cdot 5^{78} \cdot 1 - 1$	22.27	16.72	1.42	24.10	15.43	1.73	18.35	15.32	1.26	22.77	15.78	1.44
$2^{389} \cdot 3^{116} \cdot 7^{68} \cdot 1 - 1$	22.27	16.73	1.42	25.53	16.58	1.61	18.36	15.30	1.27	23.84	16.91	1.35

16-word primes	Alice KeyGen		Bob KeyGen		Alice KeyAgr		Bob KeyAgr	
	NP	P	NP	P	NP	P	NP	P
$2^{512} \cdot 3^{157} \cdot 5^{108} \cdot 1 - 1$	49.27	36.44	54.79	34.57	40.84	33.26	51.78	35.40
$2^{520} \cdot 3^{148} \cdot 7^{90} \cdot 5 - 1$	49.28	36.48	56.16	35.53	40.85	33.15	52.58	36.26
$2^{524} \cdot 3^{159} \cdot 7^{84} \cdot 1 - 1$	49.27	36.57	55.96	35.59	40.87	33.28	52.56	36.56

7-word primes	Alice KeyGen			Bob KeyGen			Alice KeyAgr			Bob KeyAgr		
	NP	P	AF	NP	P	AF	NP	P	AF	NP	P	AF
P434 [3]		5.3			5.9			5.0			5.8	
$2^{222} \cdot 3^{73} \cdot 5^{45} \cdot 1 - 1$	5.93	4.68	1.13	6.60	4.61	1.28	4.79	4.27	1.17	6.17	4.69	1.23

9-word primes	Alice KeyGen			Bob KeyGen			Alice KeyAgr			Bob KeyAgr		
	NP	P	AF	NP	P	AF	NP	P	AF	NP	P	AF
P546 [3]		10.6			11.6			9.9			11.3	
$2^{280} \cdot 3^{86} \cdot 5^{61} \cdot 1 - 1$	11.17	8.63	1.23	12.45	8.29	1.40	9.09	7.83	1.26	11.65	8.48	1.33

Table 11.4: Performance comparison of the PeSIDH against the proposed in [94] and [3]. The running time is reported in 10^6 clock cycles measured in an Intel Skylake proccessor at 4.0 GHz.Parallel version performance using 3 cores. The AF column refers to the acceleration factor of the parallel version that is our fastest implementation.

SIDH protocol, respectively. Besides, we report our results for the prime p_{1013} as an option that can be compared with the p_{964} proposed in [94], however we have no access to an implementation of this prime to realize a comparison.

Chapter 12

Conclusions and future work

In this chapter we present a general analysis of the work presented in this thesis and we consider the problems left for future research.

12.1. Conclusions

In this thesis we design software libraries that implement integer and finite field arithmetic. In order to obtain the best possible performance, we experiment with different settings. First, we implemented the arithmetic using the fastest assembly instructions `ADX/ADOX` and `MULX` that were designed specifically for arithmetic over large integers. Then, we implemented the arithmetic based on the Residue Number System using the AVX2 instructions that follow the Single Instruction Multiple Data paradigm and also using Graphical Processing Units (GPUs).

These software libraries were used to implement the RSA signature algorithm. The performance achieved by our libraries yield faster timings than previous works that implemented the RSA signature on CPU and GPU platforms. However, our results allow us to assess that when is desirable to compute arithmetic operations over large integers with lengths of $\{1024, 2048, 3072\}$ bits. A better performance is achieved by using the assembly instructions `ADX/ADOX` and `MULX`. Although, the usage of the RNS arithmetic for GPU implementation enjoys a sub-quadratic complexity in the cost of the RSA exponentiation with respect to the size of its key. Thus, we believe that for those multiplication applications where extremely large operands are required.

On the other hand, we propose practical, efficient, and secure algorithms for hashing values to elliptic curve subgroups used in pairing-based cryptography protocols, which have the indifferentiability property. We also detailed the implementation of two paring-based two-factor authentication protocols, which is secure against simple side-channel attacks. Our implementation takes advantage of processors found in recent mobiles devices, and also of the desktop processors found in contemporary laptops models. From these works, we conclude that it is possible to carry out an efficient implementations of pairing-based protocols, which are protected against time analysis attacks without sacrificing efficiency using a relatively low computational cost.

We also propose the use of pairings over elliptic curves with embedding degree one, since the improvements in algorithms for computing discrete logarithms repotred by Kim and Barbulescu in [109] do not apply to the DLP in prime-order fields $\mathbb{F}_p$. This because, the prime p does not have a special form. Moreover, we present the results of the implementation of the BLS signature algorithm for a 3072-bit prime p.

Besides, we proposed several algorithmic optimizations targeting both elliptic curve and finite field arithmetic operations, in order to accelerate the SIDH runtime performance. We accelerated the finite field operations, using the newest arithmetic instruction sets available in modern Intel processors and also, by taking advantage of the special form of the p_{CLN} prime chosen in [51]. We adapted the right-to-left Montgomery ladder variant presented in [138] to the context of the SIDH protocol. We proposed an algorithm that enables for first time the usage of precomputed look-up tables to accelerate the SIDH key generation phase. Finally, we presented an improved formula for elliptic curve point tripling.

Finally, we proposed a new construction of the SIDH protocol using non-prime power degree isogenies in the Bob's side. At first sight, this construction of the SIDH protocol called eSIDH looks more costly than the original SIDH, but by improve the isogeny constructions and evaluations, and using software parallelism it is possible achieve a considerable speedup. Moreover, We propose new λ-Montgomery-friendly primes as an alternative to those recommended in the current state-of-the-art, which maintain the same security level. These new primes allow us to perform a better modular reduction, which yields a better performance of whole protocol.

12.2. Future work

In this section we present some problems that we consider left for future research.

- Perform an efficient and secure implementation of the integer and finite field arithmetic based on RNS, using the `AVX512` instruction set. We believe that this implementation could achieve competitive or even better timings than those reported in Chapter §3.
- Perform a protected version of the BLS signature protocol implemented in Section §8. For this purpose, it is necessary to analyze if it is possible to construct a constant-time encoding that allows to define a hash function to the group $E[r]$ used in the pairings defined over elliptic curves with embedding degree one.
- Perform a security analysis of the pairings over elliptic curves with embedding degree one, when they are constructed using a Montgomery-friendly prime. In order to verify if it is possible or not apply a special version of the NFS algorithm for compute discrete logarithms.
- Build an eSIDH version using primes $p = l_A^{e_A} \prod_{i=3}^{n} l_i^{e_i} \cdot f \pm 1$, with the aim of verify the maximum value of n for which there is still a significantly reduction of the running time of the protocol.
- Design a parallel strategy that allows to efficiently compute large degree isogenies.

12.3. List of publications

- **Book chapter:** Eduardo Ochoa-Jiménez, Mehdi Tibouchi, and Francisco Rodríguez-Henríquez. Guide to Pairing-Based Cryptography, chapter Hashing into elliptic curves. Chapman & Hall/CRC, 2016.
- **Article:** A. Faz-Hernández, J. López, E. Ochoa-Jiménez, and F. Rodríguez-Henríquez. A faster software implementation of the supersingular isogeny diffie-hellman key exchange protocol. IEEE Transactions on Computers, pages 1–1, 2018.
- **Article:** J. E. Ochoa Jiménez and F. Rodríguez Henríquez. Protected implementation of pairing based two factor authentication protocols. IEEE Latin America Transactions, 14(9):4173–4180, 2016.

- **Article:** Nareli Cruz Cortés, Eduardo Ochoa-Jiménez, Luis Rivera-Zamarripa, and Francisco Rodríguez-Henríquez. A GPU parallel implementation of the RSA private operation. In High Performance Computing - Third Latin American Conference, CARLA, volume 697 of Communications in Computer and Information Science, pages 188–203, 2016.

12.3.1. Works in preparation

- Protected implementation of RSA signature algorithm. In this work we performed an efficient and secure implementation of the RSA signature algorithm using CPU and GPU platforms, experimenting with Montgomery and RNS based arithmetic. This work was realized along with Nareli C. Cortéz, Luis A. Rivera-Zamarripa and Francisco Rodríguez-Henríquez.
- Implementation of BLS signature protocol over curves with embedding degree one. In this work we design a software library that implements the digital signature algorithm BLS, constructed over elliptic curves with embedding degree one. Our library offers a security level of 128-bits by using a prime number of 3072 bits to define the finite field $\mathbb{F}_p$. This work was realized along with Francisco Rodríguez-Henríquez.
- A parallel approach for the Supersingular Isogeny Diffie-Hellman protocol. In this work we propose a new construction of the SIDH protocol using non-prime power degree isogenies in the Bob's side. This construction allow us to benefit from software parallelism and allows us to achieve a considerable speedup in the computation of the SIDH protocol. This work was performed with Daniel Cervantes-Vazquez and Francisco Rodríguez-Henríquez.

Bibliography

[1] Tolga Acar, Kristin E. Lauter, Michael Naehrig, and Daniel Shumow. Affine pairings on ARM. In *Pairing-Based Cryptography - Pairing 2012 - 5th International Conference*, volume 7708 of *Lecture Notes in Computer Science*, pages 203–209. Springer, 2012.

[2] Tolga Acar and Dan Shumow. Modular Reduction without Pre-computation for Special Moduli. Technical report, Microsoft Research, 2010.

[3] Gora Adj, Daniel Cervantes-Vázquez, Jesús-Javier Chi-Domínguez, Alfred Menezes, and Francisco Rodríguez-Henríquez. On the cost of computing isogenies between supersingular elliptic curves. Cryptology ePrint Archive, Report 2018/313, 2018. https://eprint.iacr.org/2018/313.

[4] Gora Adj and Francisco Rodríguez-Henríquez. Square root computation over even extension fields. *IEEE Trans. Computers*, 63(11):2829–2841, 2014.

[5] David Adrian, Karthikeyan Bhargavan, Zakir Durumeric, Pierrick Gaudry, Matthew Green, J. Alex Halderman, Nadia Heninger, Drew Springall, Emmanuel Thomé, Luke Valenta, Benjamin VanderSloot, Eric Wustrow, Santiago Zanella Béguelin, and Paul Zimmermann. Imperfect forward secrecy: How diffie-hellman fails in practice. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 5–17, 2015.

[6] Diego F. Aranha, Koray Karabina, Patrick Longa, Catherine H. Gebotys, and Julio López. Faster explicit formulas for computing pairings over ordinary curves. In *Advances in Cryptology - EUROCRYPT 2011 - 30th Annual International Conference on the Theory and Applications of Cryptographic Techniques*, volume 6632 of *Lecture Notes in Computer Science*, pages 48–68. Springer, 2011.

[7] A. Atkin. Probabilistic primality testing, summary by F. Morain. *Research Report 1779, INRIA*, pages 159–163, 1992.

[8] Reza Azarderakhsh, Dieter Fishbein, and David Jao. Efficient Implementation of a Quantum-Resistant Key-Exchange Protocol on Embedded Systems. Technical Report CACR 2014-20, Center of Applied Cryptographic Research (CACR), 2014.

[9] Eric Bach and Klaus Huber. Note on taking square-roots modulo N. *IEEE Trans. Information Theory*, 45(2):807–809, 1999.

[10] Joonsang Baek and Yuliang Zheng. Identity-based threshold decryption. In *Public Key Cryptography - PKC 2004, 7th International Workshop on Theory and Practice*

in Public Key Cryptography, volume 2947 of *Lecture Notes in Computer Science*, pages 262–276. Springer, 2004.

[11] Elaine Barker. Recommendation for key management, NIST special publication 800-57 part 1 revision 4. Technical report, NIST, Gaithersburg, MD, United States, January 2016.

[12] Paulo Barreto, Ben Lynn, and Michael Scott. *Constructing Elliptic Curves with Prescribed Embedding Degrees. Security in Communication Networks*, 2576:257–267, 2003.

[13] Paulo Barreto and Michael Naehrig. *Pairing-friendly elliptic curves of prime order. Selected Areas in Cryptography – SAC 2005*, 3897:319–331, 2006.

[14] Paulo S. L. M. Barreto, Ben Lynn, and Michael Scott. On the selection of pairing-friendly groups. In *Selected Areas in Cryptography, 10th Annual International Workshop, SAC*, volume 3006 of *Lecture Notes in Computer Science*, pages 17–25. Springer, 2003.

[15] Paul Barrett. Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor. In *Proceedings of Advances in Cryptology*, volume 263 of *Lecture Notes in Computer Science CRYPTO '86*, pages 311–323. Springer, 1987.

[16] Mihir Bellare and Phillip Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In *CCS '93, Proceedings of the 1st ACM Conference on Computer and Communications Security*, pages 62–73. ACM, 1993.

[17] Naomi Benger and Michael Scott. *Constructing Tower Extensions of Finite Fields for Implementation of Pairing-Based Cryptography. Arithmetic of Finite Fields*, 6087:180–195, 2010.

[18] Daniel J. Bernstein. Multidigit modular multiplication with the explicit chinese remainder theorem. Technical report, 1995.

[19] Daniel J. Bernstein. Differential addition chains, February 2006.

[20] Daniel J. Bernstein, Peter Birkner, Marc Joye, Tanja Lange, and Christiane Peters. Twisted edwards curves. In Serge Vaudenay, editor, *Progress in Cryptology – AFRICACRYPT 2008*, pages 389–405, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[21] Alexandra Boldyreva. Threshold signatures, multisignatures and blind signatures based on the gap-diffie-hellman-group signature scheme. In *Public Key Cryptography - PKC 2003, 6th International Workshop on Theory and Practice in Public Key Cryptography*, volume 2567 of *Lecture Notes in Computer Science*, pages 31–46. Springer, 2003.

[22] Dan Boneh and Matthew K. Franklin. Identity-based encryption from the weil pairing. *SIAM J. Comput.*, 32(3):586–615, 2003.

[23] Dan Boneh, Craig Gentry, Ben Lynn, and Hovav Shacham. Aggregate and verifiably encrypted signatures from bilinear maps. In *Advances in Cryptology - EUROCRYPT 2003, International Conference on the Theory and Applications of Cryptographic Techniques*, volume 2656 of *Lecture Notes in Computer Science*, pages 416–432. Springer, 2003.

[24] Dan Boneh, Ben Lynn, and Hovav Shacham. Short signatures from the weil pairing. In *Advances in Cryptology - ASIACRYPT 2001, 7th International Conference on the Theory and Application of Cryptology and Information Security*, volume 2248 of *Lecture Notes in Computer Science*, pages 514–532. Springer, 2001.

[25] J. W. Bos and S. Friedberger. Fast Arithmetic Modulo $2^x p^y - 1$. In *2017 IEEE 24th Symposium on Computer Arithmetic (ARITH)*, pages 148–155, July 2017.

[26] Joppe W. Bos, Peter L. Montgomery, Daniel Shumow, and Gregory M. Zaverucha. Montgomery multiplication using vector instructions. In *Selected Areas in Cryptography - SAC 2013 - 20th International Conference, Burnaby, BC, Canada, August 14-16, 2013, Revised Selected Papers*, pages 471–489, 2013.

[27] Colin Boyd, Paul Montague, and Khanh Quoc Nguyen. Elliptic curve based password authenticated key exchange protocols. In *Information Security and Privacy, 6th Australasian Conference, ACISP*, volume 2119 of *Lecture Notes in Computer Science*, pages 487–501. Springer, 2001.

[28] Xavier Boyen. Multipurpose identity-based signcryption (A swiss army knife for identity-based cryptography). In *Advances in Cryptology - CRYPTO 2003, 23rd Annual International Cryptology Conference*, volume 2729 of *Lecture Notes in Computer Science*, pages 383–399. Springer, 2003.

[29] Friederike Brezing and Annegret Weng. *Elliptic Curves Suitable for Pairing Based Cryptography. Designs, Codes and Cryptography*, 37:133–141, 2005.

[30] Eric Brier, Jean-Sébastien Coron, Thomas Icart, David Madore, Hugues Randriam, and Mehdi Tibouchi. Efficient indifferentiable hashing into ordinary elliptic curves. In *Advances in Cryptology - CRYPTO 2010, 30th Annual Cryptology Conference*, volume 6223 of *Lecture Notes in Computer Science*, pages 237–254. Springer, 2010.

[31] Wouter Castryck, Steven Galbraith, and Reza Rezaeian Farashahi. Efficient arithmetic on elliptic curves using a mixed edwards-montgomery representation. Cryptology ePrint Archive, Report 2008/218, 2008. https://eprint.iacr.org/2008/218.

[32] Ç. K. Koç. High-speed RSA implementation. Technical report, TR 201, RSA Laboratories, November 1994. http://cryptocode.net/docs/r01.pdf.

[33] Jae Choon Cha and Jung Hee Cheon. An identity-based signature from gap diffie-hellman groups. In *Public Key Cryptography - PKC 2003, 6th International Workshop on Theory and Practice in Public Key Cryptography*, volume 2567 of *Lecture Notes in Computer Science*, pages 18–30. Springer, 2003.

[34] Denis X. Charles, Kristin E. Lauter, and Eyal Z. Goren. Cryptographic Hash Functions from Expander Graphs. *Journal of Cryptology*, 22(1):93–113, Jan 2009.

[35] S. Chatterjee, A. Menezes, and F. Rodríguez-Henríquez. On instantiating pairing-based protocols with elliptic curves of embedding degree one. *IEEE Transactions on Computers*, 66(6):1061–1070, June 2017.

[36] Sanjit Chatterjee, Darrel Hankerson, and Alfred Menezes. On the efficiency and security of pairing-based protocols in the type 1 and type 4 settings. In *Arithmetic of Finite Fields, Third International Workshop, WAIFI*, volume 6087 of *Lecture Notes in Computer Science*, pages 114–134. Springer, 2010.

[37] Sanjit Chatterjee and Alfred Menezes. On cryptographic protocols employing asymmetric pairings - the role of Ψ revisited. *Discrete Applied Mathematics*, 159(13):1311–1322, 2011.

[38] Liqun Chen, Zhaohui Cheng, and Nigel P. Smart. Identity-based key agreement protocols from pairings. *Int. J. Inf. Sec.*, 6(4):213–241, 2007.

[39] Andres Childs, David Jao, and Vladimir Soukharev. Constructing elliptic curve isogenies in quantum subexponential time. *Journal of Mathematical Cryptology*, 8(1):1–29, February 2014.

[40] D.V Chudnovsky and G.V Chudnovsky. Sequences of numbers generated by addition in formal groups and new primality and factorization tests. *Adv. Appl. Math.*, 7(4):385–434, December 1986.

[41] Chitchanok Chuengsatiansup, Michael Naehrig, Pance Ribarski, and Peter Schwabe. Panda: Pairings and arithmetic. In *Pairing-Based Cryptography - Pairing 2013 - 6th International Conference*, volume 8365 of *Lecture Notes in Computer Science*, pages 229–250. Springer, 2013.

[42] Henri Cohen, Gerhard Frey, Roberto Avanzi, Christophe Doche, Tanja Lange, Kim Nguyen, and Frederik Vercauteren. *Handbook of Elliptic and Hyperelliptic Curve Cryptography, Second Edition*. Chapman & Hall/CRC, 2nd edition, 2012.

[43] Henri Cohen, Atsuko Miyaji, and Takatoshi Ono. Efficient elliptic curve exponentiation using mixed coordinates. In *Proceedings of the International Conference on the Theory and Applications of Cryptology and Information Security: Advances in Cryptology*, ASIACRYPT '98, pages 51–65, London, UK, UK, 1998. Springer-Verlag.

[44] D. Coppersmith. Fast evaluation of logarithms in fields of characteristic two. *IEEE Transactions on Information Theory*, 30(4):587–594, July 1984.

[45] Jean-Sébastien Coron. On the exact security of full domain hash. In *Advances in Cryptology - CRYPTO 2000, 20th Annual International Cryptology Conference, Santa Barbara, California, USA, August 20-24, 2000, Proceedings*, pages 229–235, 2000.

[46] Intel Corporation. *Intel 64 and IA-32 Architectures Optimization Reference Manual*, November 2018. https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf.

[47] Intel Corporation. *Intel Advanced Vector Extensions Programming Reference*, November 2018. https://software.intel.com/sites/default/files/4f/5b/36945.

[48] Nareli Cruz Cortés, Eduardo Ochoa-Jiménez, Luis Rivera-Zamarripa, and Francisco Rodríguez-Henríquez. A GPU parallel implementation of the RSA private operation. In *High Performance Computing - Third Latin American Conference, CARLA*, volume 697 of *Communications in Computer and Information Science*, pages 188–203, 2016.

[49] Craig Costello and Hüseyin Hisil. A Simple and Compact Algorithm for SIDH with Arbitrary Degree Isogenies. In *Advances in Cryptology – ASIACRYPT 2017: 23nd International Conference on the Theory and Application of Cryptology and Information Security, Hong Kong, China, December 3-7, 2017, Proceedings*, December 2017.

[50] Craig Costello, David Jao, Patrick Longa, Michael Naehrig, Joost Renes, and David Urbanik. Efficient Compression of SIDH Public Keys. In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *Advances in Cryptology – EUROCRYPT 2017: 36th*

Annual International Conference on the Theory and Applications of Cryptographic Techniques, Paris, France, April 30 – May 4, 2017, Proceedings, Part I, pages 679–706, Cham, 2017. Springer International Publishing.

[51] Craig Costello, Patrick Longa, and Michael Naehrig. Efficient Algorithms for Supersingular Isogeny Diffie-Hellman. In Matthew Robshaw and Jonathan Katz, editors, *Advances in Cryptology – CRYPTO 2016: 36th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2016, Proceedings, Part I*, pages 572–601, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

[52] Craig Costello and Benjamin Smith. Montgomery curves and their arithmetic. *Journal of Cryptographic Engineering*, pages 1–14, 2017.

[53] Jean-Marc Couveignes. Hard Homogeneous Spaces. Cryptology ePrint Archive, Report 2006/291, 2006. http://eprint.iacr.org/2006/291.

[54] Luca De Feo. Software for "Quantum-Resistant Cryptosystems from Supersingular Elliptic Curve Isogenies", 2011. http://github.com/defeo/ss-isogeny-software.

[55] Luca De Feo, David Jao, and Jérôme Plût. Towards Quantum-Resistant Cryptosystems from Supersingular Elliptic Curve Isogenies. *Journal of Mathematical Cryptology*, 8(3):209–247, september 2014.

[56] T. Dierks and E. Rescorla. The transport layer security (TLS) protocol version 1.2, RFC 5246. *Network Working Group, IETF*, 2008.

[57] Whitfield Diffie and Martin E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, 1976.

[58] Jiankuo Dong, Fangyu Zheng, Wuqiong Pan, Jingqiang Lin, Jiwu Jing, and Yuan Zhao. Utilizing the double-precision floating-point computing power of GPUs for RSA acceleration. *Security and Communication Networks*, 2017, September 2017.

[59] Heba Mohammed Fadhil and Mohammed Issam Younis. Parallelizing RSA algorithm on multicore CPU and GPU. *International Journal of Computer Applications*, 87(6):15–22, February 2014.

[60] Reza Rezaeian Farashahi, Pierre-Alain Fouque, Igor E. Shparlinski, Mehdi Tibouchi, and José Felipe Voloch. Indifferentiable deterministic hashing to elliptic and hyperelliptic curves. *Math. Comput.*, 82(281):491–512, 2013.

[61] Reza Rezaeian Farashahi, Igor E. Shparlinski, and José Felipe Voloch. On hashing into elliptic curves. *J. Mathematical Cryptology*, 3(4):353–360, 2009.

[62] A. Faz-Hernández, J. López, E. Ochoa-Jiménez, and F. Rodríguez-Henríquez. A faster software implementation of the supersingular isogeny diffie-hellman key exchange protocol. *IEEE Transactions on Computers*, pages 1–1, 2018.

[63] Armando Faz-Hernández, Patrick Longa, and Ana H. Sánchez. Efficient and secure algorithms for glv-based scalar multiplication and their implementation on GLV-GLS curves (extended version). *J. Cryptographic Engineering*, 5(1):31–52, 2015.

[64] Pierre-Alain Fouque and Mehdi Tibouchi. Estimating the size of the image of deterministic hash functions to elliptic curves. In *Progress in Cryptology - LATINCRYPT 2010, First International Conference on Cryptology and Information Security in Latin America*, volume 6212 of *Lecture Notes in Computer Science*, pages 81–91. Springer, 2010.

[65] Pierre-Alain Fouque and Mehdi Tibouchi. Indifferentiable hashing to barreto-naehrig curves. In *Progress in Cryptology - LATINCRYPT 2012 - 2nd International Conference on Cryptology and Information Security in Latin America*, volume 7533 of *Lecture Notes in Computer Science*, pages 1–17. Springer, 2012.

[66] John B. Fraleigh and Victor J. Katz. *A first course in abstract algebra*. Addison-Wesley, 7 edition, 2003.

[67] David Freeman, Michael Scott, and Edlyn Teske. *A Taxonomy of Pairing-Friendly Elliptic Curves. Journal of Cryptology*, 23:224–280, 2010.

[68] Gerhard Frey and Hans-Georg Rück. A remark concerning m-divisibility and the discrete logarithm in the divisor class group of curves. *Math. Comput.*, 62(206):865–874, April 1994.

[69] Laura Fuentes-Castañeda, Edward Knapp, and Francisco Rodríguez-Henríquez. Faster hashing to ${\mathbb G}_2$. In *Selected Areas in Cryptography - 18th International Workshop, SAC*, volume 7118 of *Lecture Notes in Computer Science*, pages 412–430. Springer, 2011.

[70] Steven D. Galbraith, Xibin Lin, and Michael Scott. Endomorphisms for faster elliptic curve cryptography on a large class of curves. *J. Cryptology*, 24(3):446–469, 2011.

[71] Steven D. Galbraith, Kenneth G. Paterson, and Nigel P. Smart. Pairings for cryptographers. *Discrete Applied Mathematics*, 156(16):3113–3121, september 2008.

[72] Robert P. Gallant, Robert J. Lambert, and Scott A. Vanstone. Faster point multiplication on elliptic curves with efficient endomorphisms. In *Advances in Cryptology - CRYPTO 2001, 21st Annual International Cryptology Conference*, volume 2139 of *Lecture Notes in Computer Science*, pages 190–200. Springer, 2001.

[73] Craig Gentry and Alice Silverberg. Hierarchical id-based cryptography. In *Advances in Cryptology - ASIACRYPT 2002, 8th International Conference on the Theory and Application of Cryptology and Information Security*, volume 2501 of *Lecture Notes in Computer Science*, pages 548–566. Springer, 2002.

[74] C. C. F. Pereira Geovandro, Marcos A. Simplício Jr., Michael Naehrig, and Paulo S. L. M. Barreto. A family of implementation-friendly BN elliptic curves. *Journal of Systems and Software*, 84(8):1319–1326, 2011.

[75] Shafi Goldwasser and Joe Kilian. *Primality Testing Using Elliptic Curves. J. ACM*, 46:450–472, 1999.

[76] V. Gopal, J.D. Guilford, G.M. Wolrich, W.K. Feghali, E. Ozturk, M.G. Dixon, S.P. Mirkes, M.C. Merten, T. Li, and T.T.I. Bret. Addition instructions with independent carry chains, January 9 2014. US Patent App. 13/993,483.

[77] Daniel M. Gordon. Discrete logarithms in gf(p) using the number field sieve. *SIAM J. Discret. Math.*, 6(1):124–138, February 1993.

[78] Torbjörn Granlund and the GMP development team. *GNU MP: The GNU Multiple Precision Arithmetic Library*, 6.1.0 edition, 2016. http://gmplib.org/.

[79] Gurleen Grewal, Reza Azarderakhsh, Patrick Longa, Shi Hu, and David Jao. Efficient implementation of bilinear pairings on ARM processors. In *Selected Areas in Cryptography, 19th International Conference, SAC 2012*, volume 7707 of *Lecture Notes in Computer Science*, pages 149–165. Springer, 2012.

[80] Shay Gueron and Vlad Krasnov. Fast prime field elliptic-curve cryptography with 256-bit primes. *Journal of Cryptographic Engineering*, pages 1–11, 2014.

[81] Shay Gueron and Vlad Krasnov. Speed records for multi-prime RSA using AVX2 architectures. In *Information Technology: New Generations: 13th International Conference on Information Technology*, pages 237–245. Springer International Publishing, 2016.

[82] Darrel Hankerson, Alfred J. Menezes, and Scott Vanstone. *Guide to Elliptic Curve Cryptography*. Professional Computing. Springer, 2004.

[83] M. Harris. Optimizing parallel reduction in CUDA. Technical report, nVidia, 2008.

[84] William Hasenplaugh, Gunnar Gaubatz, and Vinodh Gopal. Fast modular reduction. In *Proceedings of the 18th IEEE Symposium on Computer Arithmetic*, ARITH '07, pages 225–229, Washington, DC, USA, 2007. IEEE Computer Society.

[85] Florian Hess, Nigel Smart, and Frederik Vercauteren. *The Eta Pairing Revisited. IEEE Transactions on Information Theory*, 52:4595–4602, 2006.

[86] Jeffrey Hoffstein, Jill Pipher, and J.H. Silverman. *An Introduction to Mathematical Cryptography*. Springer Publishing Company, Incorporated, 1 edition, 2008.

[87] Jeremy Horwitz and Ben Lynn. Toward hierarchical identity-based encryption. In *Advances in Cryptology - EUROCRYPT 2002, International Conference on the Theory and Applications of Cryptographic Techniques*, volume 2332 of *Lecture Notes in Computer Science*, pages 466–481. Springer, 2002.

[88] Zhi Hu, Lin Wang, Maozhi Xu, and Guoliang Zhang. *Generation and Tate Pairing Computation of Ordinary Elliptic Curves with Embedding Degree One*, pages 393–403. Springer International Publishing, 2013.

[89] Thomas Icart. How to hash into elliptic curves. In *Advances in Cryptology - CRYPTO 2009, 29th Annual International Cryptology Conference*, volume 5677 of *Lecture Notes in Computer Science*, pages 303–316. Springer, 2009.

[90] David P. Jablon. Strong password-only authenticated key exchange. *Computer Communication Review*, 26(5):5–26, 1996.

[91] David P. Jablon. Extended password key exchange protocols immune to dictionary attacks. In *6th Workshop on Enabling Technologies (WET-ICE '97), Infrastructure for Collaborative Enterprises*, pages 248–255. IEEE, 1997.

[92] Keon Jang, Sangjin Han, Seungyeop Han, Sue Moon, and KyoungSoo Park. SSLShader: Cheap ssl acceleration with commodity processors. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, pages 1–14, Berkeley, CA, USA, 2011. USENIX Association.

[93] D. Jao and R. Venkatesan. Use of isogenies for design of cryptosystems, May 5 2005. US Patent App. 10/816,083.

[94] David Jao, Reza Azarderakhsh, Matthew Campagna, Craig Costello, Luca De Feo, Basil Hess, Amir Jalali, Brian Koziel, Brian LaMacchia, Patrick Longa, Michael Naehrig, Joost Renes, Vladimir Soukharev, and David Urbanik. Supersingular isogeny key encapsulation, 2017. sike.org.

[95] David Jao and Luca De Feo. Towards Quantum-Resistant Cryptosystems from Supersingular Elliptic Curve Isogenies. In Bo-Yin Yang, editor, *Post-Quantum Cryptography: 4th International Workshop, PQCrypto 2011, Taipei, Taiwan. Proceedings*, pages 19–34, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[96] Hamza Jeljeli. *Accélérateurs logiciels et matériels pour l'algèbre linéaire creuse sur les corps finis*. PhD thesis, Inria Nancy - Grand Est, LORIA - ALGO - Department of Algorithms, Computation, Image and Geometry, available at: https://hal.inria.fr/tel-01178931, 2015.

[97] Hamza Jeljeli. *Accelerating Iterative SpMV for the Discrete Logarithm Problem Using GPUs*, volume 9061 of *Lecture Notes in Computer Science*, pages 25–44. Springer International Publishing, 2015.

[98] J. E. Ochoa Jimenez and F. Rodriguez Henriquez. Protected implementation of pairing based two factor authentication protocols. *IEEE Latin America Transactions*, 14(9):4173–4180, 2016.

[99] Antoine Joux. A one round protocol for tripartite diffie-hellman. *J. Cryptology*, 17(4):263–276, 2004.

[100] Marc Joye. Highly Regular Right-to-Left Algorithms for Scalar Multiplication. In Pascal Paillier and Ingrid Verbauwhede, editors, *Cryptographic Hardware and Embedded Systems - CHES 2007: 9th International Workshop, Vienna, Austria, September 10-13, 2007. Proceedings*, pages 135–147, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[101] Marc Joye and Michael Tunstall. Exponent recoding and regular exponentiation algorithms. In *Proceedings of the 2nd International Conference on Cryptology in Africa, 2009*, AFRICACRYPT '09, pages 334–349, Gammarth, Tunisia, 2009. Springer.

[102] Benjamin Justus and Daniel Loebenberger. Differential addition in generalized edwards coordinates. Cryptology ePrint Archive, Report 2009/523, 2009. https://eprint.iacr.org/2009/523.

[103] Ezekiel Kachisa, Edward Schaefer, and Michael Scott. *Constructing Brezing-Weng Pairing-Friendly Elliptic Curves Using Elements in the Cyclotomic Field. Pairing-Based Cryptography - Pairing 2008*, 5209:126–135, 2008.

[104] B. S. Kaliski. The montgomery inverse and its applications. *IEEE Transactions on Computers*, 44(8):1064–1065, Aug 1995.

[105] Anatolii Karatsuba and Yuri Ofman. Multiplication of Multidigit Numbers on Automata. *Soviet Physics-Doklady*, 7:595–596, 1963.

[106] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography, Second Edition*. Chapman & Hall/CRC, 2014.

[107] Shinichi Kawamura, Masanobu Koike, Fumihiko Sano, and Atsushi Shimbo. Cox-rower architecture for fast parallel montgomery multiplication. In Bart Preneel, editor, *Advances in Cryptology — EUROCRYPT 2000*, pages 523–538, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.

[108] Shinichi Kawamura, Yuichi Komano, Hideo Shimizu, and Tomoko Yonemura. RNS montgomery reduction algorithms using quadratic residuosity. *Journal of Cryptographic Engineering*, September 2018.

[109] Taechan Kim and Razvan Barbulescu. Extended tower number field sieve: A new complexity for the medium prime case. In Matthew Robshaw and Jonathan Katz, editors, *Advances in Cryptology – CRYPTO 2016*, pages 543–571, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

[110] Edward Knapp. *On the Efficiency and Security of Cryptographic Pairings*. PhD thesis, University of Waterloo, Ontario, Canada, 2012.

[111] Miroslav Knežević, Frederik Vercauteren, and Ingrid Verbauwhede. Speeding Up Bipartite Modular Multiplication. In M. Anwar Hasan and Tor Helleseth, editors, *Arithmetic of Finite Fields: Third International Workshop, WAIFI 2010, Istanbul, Turkey, June 27-30, 2010. Proceedings*, pages 166–179, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[112] Neal Koblitz. Elliptic curve cryptosystems. *Mathematics of Computation*, 48(177):203–209, 1987.

[113] Paul C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '96, pages 104–113, London, UK, 1996. Springer-Verlag.

[114] B. Koziel, R. Azarderakhsh, M. Mozaffari Kermani, and D. Jao. Post-Quantum Cryptography on FPGA Based on Isogenies on Elliptic Curves. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 64(1):86–99, Jan 2017.

[115] Brian Koziel, Reza Azarderakhsh, and Mehran Mozaffari-Kermani. Fast Hardware Architectures for Supersingular Isogeny Diffie-Hellman Key Exchange on FPGA. In Orr Dunkelman and Somitra Kumar Sanadhya, editors, *Progress in Cryptology – INDOCRYPT 2016: 17th International Conference on Cryptology in India, Kolkata, India, December 11-14, 2016, Proceedings*, pages 191–206, Cham, 2016. Springer International Publishing.

[116] Brian Koziel, Amir Jalali, Reza Azarderakhsh, David Jao, and Mehran Mozaffari-Kermani. NEON-SIDH: Efficient Implementation of Supersingular Isogeny Diffie-Hellman Key Exchange Protocol on ARM. In Sara Foresti and Giuseppe Persiano, editors, *Cryptology and Network Security: 15th International Conference, CANS 2016, Milan, Italy, November 14-16, 2016, Proceedings*, pages 88–103, Cham, 2016. Springer International Publishing.

[117] Peter L. Montgomery and Robert D. Silverman. An FFT extension to the P - 1 factoring algorithm. 54:839–854, 04 1990.

[118] A. K. Lenstra, H. W. Lenstra, and L. Lovasz. Factoring polynomials with rational coefficients. *Mathematische Annalen*, 261(4):515–534, 1982.

[119] Arjen K. Lenstra. Generating RSA Moduli with a Predetermined Portion. In Kazuo Ohta and Dingyi Pei, editors, *Advances in Cryptology — ASIACRYPT'98: International Conference on the Theory and Application of Cryptology and Information Security Beijing, China, October 18–22, 1998 Proceedings*, pages 1–10, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg.

[120] Benoît Libert and Jean-Jacques Quisquater. Efficient signcryption with key privacy from gap diffie-hellman groups. In *Public Key Cryptography - PKC 2004, 7th International Workshop on Theory and Practice in Public Key Cryptography*, volume 2947 of *Lecture Notes in Computer Science*, pages 187–200. Springer, 2004.

[121] Julio López and Ricardo Dahab. Fast Multiplication on Elliptic Curves Over GF(2m) without precomputation. In Çetin K. Koç and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems: First International Workshop, CHES'99 Worcester, MA, USA, August 12–13, 1999 Proceedings*, pages 316–327, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[122] Harold M. Edwards. A normal form for elliptic curves. 44:393–423, 07 2007.

[123] Ueli M. Maurer, Renato Renner, and Clemens Holenstein. Indifferentiability, impossibility results on reductions, and applications to the random oracle methodology. In *Theory of Cryptography, First Theory of Cryptography Conference, TCC*, volume 2951 of *Lecture Notes in Computer Science*, pages 21–39. Springer, 2004.

[124] Alfred Menezes, Tatsuaki Okamoto, and Scott A. Vanstone. Reducing elliptic curve logarithms to logarithms in a finite field. *IEEE Trans. Information Theory*, 39(5):1639–1646, 1993.

[125] Michael Meyer and Steffen Reith. A faster way to the csidh. Cryptology ePrint Archive, Report 2018/782, 2018. https://eprint.iacr.org/2018/782.

[126] Michael Meyer, Steffen Reith, and Fabio Campos. On hybrid sidh schemes using edwards and montgomery curve arithmetic. Cryptology ePrint Archive, Report 2017/1213, 2017. https://eprint.iacr.org/2017/1213.

[127] Victor S. Miller. Use of elliptic curves in cryptography. In *Advances in Cryptology - CRYPTO '85, Proceedings*, volume 218 of *Lecture Notes in Computer Science*, pages 417–426. Springer, 1986.

[128] Victor S. Miller. The weil pairing, and its efficient calculation. *Journal of Cryptology*, 17(4):235–261, September 2004.

[129] Hermann Minkowski. *Geometrie der Zahlen*. Leipzig und Berlin, Druck ung Verlag von B.G. Teubner, 1910.

[130] Shigeo Mitsunari. A fast implementation of the optimal ate pairing over BN curve on Intel Haswell processor. *IACR Cryptology ePrint Archive*, 2013:362, 2013.

[131] Peter L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, April 1985.

[132] Peter L. Montgomery. Speeding the Pollard and Elliptic Curve Methods of Factorization. *Mathematics of Computation*, 48(177):243–264, 1987.

[133] Dustin Moody and Daniel Shumow. Analogues of velu's formulas for isogenies on alternate models of elliptic curves. Cryptology ePrint Archive, Report 2011/430, 2011. https://eprint.iacr.org/2011/430.

[134] S. Müller. On the computation of square roots in finite fields. *J. Design, Codes and Cryptography*, 31:301–312, 2004.

[135] Eduardo Ochoa-Jiménez, Mehdi Tibouchi, and Francisco Rodríguez-Henríquez. *Guide to Pairing-Based Cryptography*, chapter Hashing into elliptic curves. Chapman & Hall/CRC, 2016.

[136] Katsuyuki Okeya and Kouichi Sakurai. Efficient Elliptic Curve Cryptosystems from a Scalar Multiplication Algorithm with Recovery of the y-Coordinate on a Montgomery-Form Elliptic Curve. In Çetin K. Koç, David Naccache, and Christof Paar, editors,

Cryptographic Hardware and Embedded Systems — CHES 2001: Third International Workshop Paris, France, May 14–16, 2001 Proceedings, pages 126–141, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[137] Thomaz Oliveira, Diego F. Aranha, Julio López, and Francisco Rodríguez-Henríquez. Fast Point Multiplication Algorithms for Binary Elliptic Curves with and without Precomputation. In Antoine Joux and Amr Youssef, editors, *Selected Areas in Cryptography – SAC 2014: 21st International Conference, Montreal, QC, Canada, August 14-15, 2014, Revised Selected Papers*, pages 324–344, Cham, 2014. Springer International Publishing.

[138] Thomaz Oliveira, Julio López, Hüseyin Hışıl, Armando Faz-Hernández, and Francisco Rodríguez-Henríquez. How to (pre-)compute a ladder. In Jan Camenisch and Carlisle Adams, editors, *Selected Areas in Cryptography – SAC 2017: 24th International Conference, Ottawa, Ontario, Canada, August 16 - 18, 2017, Revised Selected Papers*. Springer International Publishing, August 2017.

[139] Thomaz Oliveira, Julio López, and Francisco Rodríguez-Henríquez. Software implementation of koblitz curves over quadratic fields. In Benedikt Gierlichs and Axel Y. Poschmann, editors, *Cryptographic Hardware and Embedded Systems – CHES 2016*, pages 259–279, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

[140] K. G. Paterson. *Cryptography from Pairings*, volume 317 of *London Mathematical Society Lecture Notes*, chapter X, pages 215–251. Cambridge University Press, 2005.

[141] John Pollard. *Monte Carlo methods for Index Computation (mod p). Mathematics of Computation*, 32:918–924, 1978.

[142] K. C. Posch and R. Posch. Modulo reduction in residue number systems. *IEEE Transactions on Parallel and Distributed Systems*, 6(5):449–454, May 1995.

[143] Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126, 1978.

[144] Ryuichi Sakai, Kiyoshi Ohgishi, and Masao Kasahara. Cryptosystems based on pairing.

[145] Ana Helena Sánchez and Francisco Rodríguez-Henríquez. NEON implementation of an attribute-based encryption scheme. In *Applied Cryptography and Network Security - 11th International Conference, ACNS 2013*, volume 7954 of *Lecture Notes in Computer Science*, pages 322–338. Springer, 2013.

[146] René Schoof. Elliptic curves over finite fields and the computation of square roots mod p. *Mathematics of Computation*, 44(170):483–494, 1985.

[147] Michael Scott. Implementing cryptographic pairings. In *Pairing-Based Cryptography - Pairing 2007*, volume 4575 of *Lecture Notes in Computer Science*, pages 177–196. Springer, 2007.

[148] Michael Scott. Client-server authentication using pairings. *IACR Cryptology ePrint Archive*, 2012:148, 2012.

[149] Michael Scott. Unbalancing pairing-based key exchange protocols. *IACR Cryptology ePrint Archive*, 2013:688, 2013.

[150] Michael Scott, Naomi Benger, Manuel Charlemagne, Luis J. Dominguez Perez, and Ezekiel J. Kachisa. Fast hashing to G_2 on pairing-friendly curves. In *Pairing-Based Cryptography - Pairing 2009, Third International Conference*, volume 5671 of *Lecture Notes in Computer Science*, pages 102–113. Springer, 2009.

[151] Mike Scott. Missing a trick: Karatsuba revisited. *IACR Cryptology ePrint Archive*, 2015:1247, 2015.

[152] Andrew Shallue and Christiaan van de Woestijne. Construction of rational points on elliptic curves over finite fields. In *Algorithmic Number Theory, 7th International Symposium, ANTS-VII*, volume 4076 of *Lecture Notes in Computer Science*, pages 510–524. Springer, 2006.

[153] Adi Shamir. Identity-based cryptosystems and signature schemes. *Proceedings of CRYPTO'84 on Advances in cryptology*, pages 47–53, 1984.

[154] D. Shanks. Five number-theoretic algorithms. *Proceedings of the second Manitoba conference on numerical mathematics*, pages 51–70, 1972.

[155] Victor Shoup. *A Computational Introduction to Number Theory and Algebra*. Cambridge University Press, New York, NY, USA, 2 edition, 2009.

[156] Anton Stolbunov. Constructing public-key cryptographic schemes based on class group action on a set of isogenous elliptic curves. *Advances in Mathematics of Communications*, 4(2):215–235, 2010.

[157] Srinivasa Rao Subramanya Rao. Three Dimensional Montgomery Ladder, Differential Point Tripling on Montgomery Curves and Point Quintupling on Weierstrass' and Edwards Curves. In David Pointcheval, Abderrahmane Nitaj, and Tajjeeddine Rachidi, editors, *Progress in Cryptology – AFRICACRYPT 2016: 8th International Conference on Cryptology in Africa, Fes, Morocco, April 13-15, 2016, Proceedings*, pages 84–106, Cham, 2016. Springer International Publishing.

[158] John Tate. Wc-groups over p-adic fields. Exposé 156, Séminaire Bourbaki, 1957.

[159] Edlyn Teske. The pohlig—hellman method generalized for group structure computation. *J. Symb. Comput.*, 27(6):521–534, June 1999.

[160] A. Tonelli. Bemerkung uber die auflosung quadratischer congruenzen. *Götinger Nachrichten*, pages 344–346, 1891.

[161] Jacques Vélu. *C. R. Acad. Sci. Paris Sér. A-B*, 273:A238–A241, 1971.

[162] F. Vercauteren. Optimal pairings. *IEEE Transactions on Information Theory*, 56(1):455–461, 2010.

[163] C. D. Walter. Montgomery exponentiation needs no final subtractions. *Electronics Letters*, 35(21):1831–1832, Oct 1999.

[164] Lawrence C. Washington. *Elliptic Curves: Number Theory and Cryptography*. Chapman & Hall/CRC, second edition, 2008.

[165] André Weil. *Sur les fonctions algebriques á corps de constantes finis*, volume I, pages 257–259. Oeuvres Scientifiques, Paris, 1940.

[166] André Weimerskirch and Christof Paar. Generalizations of the karatsuba algorithm for efficient implementations. Cryptology ePrint Archive, Report 2006/224, 2006. http://eprint.iacr.org/2006/224.

[167] Yang Yang, Zhi Guan, Huiping Sun, and Zhong Chen. Accelerating RSA with fine-grained parallelism using GPU. In *Proceedings of the Information Security Practice and Experience: 11th International Conference, 2015*, ISPEC'15, pages 454–468, Beijing, China, 2015. Springer.

[168] Eric Zavattoni, Luis J. Dominguez Perez, Shigeo Mitsunari, Ana H. Sánchez-Ramírez, Tadanori Teruya, and Francisco Rodríguez-Henríquez. Software implementation of an attribute-based encryption scheme. *IEEE Trans. Computers*, 64(5):1429–1441, 2015.

[169] Fangguo Zhang and Kwangjo Kim. Id-based blind signature and ring signature from pairings. In *Advances in Cryptology - ASIACRYPT 2002, 8th International Conference on the Theory and Application of Cryptology and Information Security*, volume 2501 of *Lecture Notes in Computer Science*, pages 533–547. Springer, 2002.