On the Computation of Multi-Scalar Multiplication for Pairing-Based zkSNARKs

by

Guiwen Luo

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

Waterloo, Ontario, Canada, 2023

Examining Committee Membership

The following served on the Examining Committee for this thesis. The decision of the Examining Committee is by majority vote.

External Examiner:	Liqun Chen Professor, CS, University of Surrey
Supervisor:	Guang Gong Professor, ECE, University of Waterloo
Internal Member:	Anwar Hasan Professor, ECE, University of Waterloo
Internal Member:	Mahesh Tripunitara Professor, ECE, University of Waterloo
Internal-External Member:	Douglas Stebila Professor, C&O, University of Waterloo

Author's Declaration

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

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

Abstract

Multi-scalar multiplication refers to the operation of computing multiple scalar multiplications in an elliptic curve group and then adding them together. It is an essential operation for proof generation and verification in pairing-based trusted setup zero-knowledge succinct non-interactive argument of knowledge (zkSNARK) schemes, which enable privacy-preserving features in many blockchain applications. Pairing-based trusted setup zkSNARKs usually follow a common paradigm. A public string composed of a list of fixed points in an elliptic curve group called *common reference string* is generated in a trusted setup and accessible to all parties involved. The prover generates a zkSNARK proof by computing multi-scalar multiplications over the points in the common reference string and performing other operations. The verifier verifies the proof by computing multi-scalar multiplications and elliptic curve bilinear pairings.

Multi-scalar multiplication in pairing-based trusted setup zkSNARKs has two characteristics. First, all the points are *fixed* once the common reference string is generated. Second, the number of points n is typically *large*, with the thesis targeting at $n = 2^e$ $(10 \leq e \leq 21)$. Our goal in this thesis is to propose and implement efficient algorithms for computing multi-scalar multiplication in order to enable efficient zkSNARKs.

This thesis primarily includes three aspects. First, the background knowledge is introduced and the classical multi-scalar multiplication algorithms are reviewed. Second, two frameworks for computing multi-scalar multiplications over fixed points and five corresponding auxiliary set pairs are proposed. Finally, the theoretical analysis, software implementation, and experimental tests on the representative instantiations of the proposed frameworks are presented.

Acknowledgements

First and foremost, I would like to express my sincere gratitude to Prof. Guang Gong for her guidance and encouragement throughout my PhD journey. Her mentorship has not only shaped my academic development but also provided me with mental resilience to overcome challenges. I will always cherish the memorable Christmas celebration at her home, which brought warmth and joy to our group, especially as most of us are international students experiencing a festival without families in the cold winter. I will always cherish her regular invitations for group meals every semester, although sometimes we had to have takeout in office when dine-in options were unavailable. Additionally, she organized farewell tea gatherings for departing group members. I might be the next one, as I prepare to graduate. I deeply appreciate the sense of belonging she created within our group.

I would like to thank Shihui Fu, Chenkai Weng, and Prof. Douglas Stebila for the insightful discussions, which helped me make the following content possible, Construction IV in Section 4.4.1, Construction V in Section 4.4.2, and Algorithm 11 in Section 6.1.4, respectively.

I would like to thank the examination committee. Without their support and feedback during the two comprehensive exams, the PhD seminar and the thesis defence, I would not have been able to reach this point successfully.

I would like to thank the staffs in Faculty of Engineering, not only for their administrative assistance academically but also for organizing social events such as the Chinese New Year celebration. These events provided me with a sense of community.

I would like to thank my friends who have stood by me during the last four years. We commuted together, had meals together, traveled together, spending more time together in the past four years than with our own families.

I am also thankful for these random individuals I encountered by chance and had brief conversations with during my time in Waterloo. Whether it was in classrooms, in the swimming pool, on the walking trails, or even on public transportation, the interactions with strangers has been uplifting.

To my wife, Lijun Lyu, I am profoundly thankful for your love, understanding, support and belief in me, despite the distance that has separated us over these years.

Lastly, I would like to express my deepest appreciation to my parents, whose unconditional love and unwavering support have been my anchor in life. Regardless of where I go, what I do, or the outcomes I achieve, I always know that home is my last sanctuary. With this belief, I fear no failure.

Dedication

To my family.

Table of Contents

List of Figures

List of Tables

List of Algorithms

Nomenclature

$a := b$	Define a as b.
$aP, a \cdot P$	Scalar multiplication or single-scalar multiplication, where a is an integer and P is a point in an elliptic curve group.
$S_{n,r}$	Notation for $\sum_{i=1}^{n} a_i P_i$, where a_i's are scalars such that $0 \leq a_i < r$ and P_i's are fixed points in an elliptic curve group.
$\lceil x \rceil$	The smallest integer that is no less than x.
$s.t.$	Such that.
$[a, b]$	Closed interval. The set of integers x satisfying $a \leq x \leq b$.
$\mathbb{Z}_p$	The ring of p elements or the finite field of p elements, where p is a prime.
$O \leftarrow \mathsf{fun}(X)$	A function named fun that takes X as input, and then outputs O.
$\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T$	Three groups over which a pairing is defined. $\mathbb{G}_1$ and $\mathbb{G}_2$ are elliptic curve groups, and $\mathbb{G}_T$ is a subgroup of a finite field. When discussing computational complexity, we also refer to the twist group $\mathbb{G}_2'$, which is defined over a lower extension field, as $\mathbb{G}_2$.
$\mathbb{F}^*$	Finite field $\mathbb{F}$ excluding 0.
$\lfloor x \rfloor$	The largest integer that is no more than x.
$\omega_2(b)$	The exponent of factor 2 in b. If $b = 2^e k,\ 2 \nmid k$, then $\omega_2(b) = e$.

Chapter 1

Introduction

1.1 Motivation

The goal in this thesis is to establish efficient methods for computing multi-scalar multiplication over fixed points, which is an essential and time-consuming operation in many pairing-based trusted setup zero-knowledge succinct non-interactive argument of knowledge (zkSNARK) schemes.

zkSNARKs have been a hot topic lately because they can be used to build privacy-preserving blockchains. Known for the successful deployment in Zcash [Zca], zkSNARKs provide a convenient tool to construct blockchains that support shielded transactions, in which the sender, the receiver, and the coin being transacted remain confidential when verifying the validity of transactions.

zkSNARKs enable one party (the prover) to prove to another party (the verifier) that a statement is true, without revealing other information apart from the fact that the statement is true. Among the many zkSNARK schemes, there is a popular category of them, which are trusted setup zkSNARKs built upon elliptic curve bilinear pairings [GGPR13, DFGK14, Gro16, MBKM19, GWC19, CHM+20]. The representatives in this category are Groth16 [Gro16] and PlonK [GWC19].

These pairing-based trusted setup zkSNARKs typically follow a common paradigm. There is a public string composed of a list of fixed points in an elliptic curve group called common reference string, which is generated in a trusted setup[1] and accessible to all par-

[1] Trusted setup refers to a pre-processing phase that generates a trapdoor hidden in the common reference string.

ties. The prover generates the zkSNARK proof by computing multi-scalar multiplications over the points in the common reference string and performing other operations. The verifier verifies the proof by computing multi-scalar multiplications and elliptic curve bilinear pairings.

Multi-scalar multiplication, which is also called n-scalar multiplication if the number of points involved is n, refers to the operation of computing multiple scalar multiplications in an elliptic curve group and then adding them together. In pairing-based trusted setup zkSNARKs, cryptographic pairings can be quickly evaluated within several milliseconds, while the computation of multi-scalar multiplications is time-consuming, usually measured in seconds. The number of points n involved in multi-scalar multiplication can be very large. For examples, given an image of SHA256, if a prover wants to prove the knowledge of a 512-bit preimage by utilizing Groth16 [Gro16] in a zero-knowledge manner, n in the corresponding n-scalar multiplication is tens of thousands [CGGN17]. To hide the identity of a coin, Zcash originally proposed to prove the membership of the coin's commitment in a 64-layer Merkle tree built upon SHA256, and n in the corresponding n-scalar multiplication is more than a million [BCTV13, BCG+14].

Multi-scalar multiplication in pairing-based trusted setup zkSNARKs exhibits two characteristics. First, all points are fixed once the common reference string is generated. Second, the number of points n is large. This thesis targets at $n = 2^e$ $(10 \leq e \leq 21)$.

In light of the importance of multi-scalar multiplication in the process of proof generation and verification for pairing-based trusted setup zkSNARKs, some new techniques that utilize precomputation to speed up the computation of multi-scalar multiplication over fixed points are proposed and analyzed in this thesis.

1.2 Related work

1.2.1 Multi-scalar multiplication

The most popular method for computing scalar multiplication in elliptic curve groups is the binary algorithm, known as the doubling and addition method. It is also known as the square and multiplication method in the exponentiation setting [Knu97, Section 4.6.3]. GLV method [GLV01] and GLS method [GLS11] decompose the scalar into dimensions 2, 4, 6 and 8, then compute the corresponding multi-scalar multiplication. When the point for scalar multiplication is fixed, precomputation can be used to reduce the computational cost. Knuth's 5 window algorithm utilizes the precomputation table of 16 points

to speed up scalar multiplication [Knu97, BC89]. If a bigger window and more memory for precomputed points are used, window method can be even faster. Pippenger's bucket method and its variants decompose the scalar, then sort all points into buckets with respect to their scalars, and finally utilize an accumulation algorithm to add them together [Pip76, BDLO12]. Another line of research lies in constructing new number systems to represent the scalar, such as basic digit sets [Mat82, BGMW92] and multi-base number systems [DKS09, SIM12, YWLT13]. Researchers also try to make the addition arithmetic more efficient by using different curve representations, such as projective coordinates and Jacobian coordinates that eliminate the inversion operations, and Montgomery form that utilizes the x-only-coordinate systems [Mon87]. Differential addition chains are used in conjunction with x-only-coordinate systems, for example, PRAC chains [Mon92], DJB chains [Ber06] and other multi-dimensional differential addition chains [Bro15, Rao15].

Most of the aforementioned techniques can be applied to n-scalar multiplication when the number of points n is small. When n is large, which is the case in pairing-based trusted setup zkSNARK schemes, Pippenger's bucket method and its variants are the state-of-the-art algorithms that outperform other competitors. Published in 1976 [Pip76], Pippenger's bucket method has recently become an essential tool for privacy-preserving blockchains due to its effectiveness for computing multi-scalar multiplication in pairing-based zkSNARKs, pairing-based cryptographic commitments, BLS signature aggregations, and so on. A variant of Pippenger's bucket method that utilizes precomputation to speed up single-scalar multiplication was introduced by Brickell, Gordon, McCurley and Wilson [BGMW92]. This variant can be applied to n-scalar multiplication with large n. In order to achieve fast batch forgery signature verification, Bernstein *et al.* [BDLO12] investigated Bos-Coster method [DR94, Section 4], Straus method [Str64] and Pippenger's bucket method, then chose Pippenger's bucket method for implementation, which marked the start of extensive employment of Pippenger's bucket method for computing n-scalar multiplication with large n.

By now, some popular zkSNARK applications, such as Zcash [Zca], Aztec [Azt], TurboPLONK [GJW20] and gnark [gna], have already adopted Pippenger's bucket method for computing multi-scalar multiplication.

1.2.2 Pairing-based trusted setup zkSNARKs

There are several classic approaches to constructing zkSNARKs, one of which involves utilizing elliptic curve bilinear pairings. In 2010, Groth made the first attempt to introduce elliptic curve pairing into zero-knowledge proofs by constructing an argument scheme for

circuit satisfiability [Gro10]. This scheme does not rely on probabilistic checkable proofs or Fiat-Shamir heuristic. Following that, Gennaro, Gentry, Parno and Raykova [GGPR13] introduced the new characterizations of the NP class called quadratic span program (QSP) and quadratic arithmetic program (QAP). They constructed a zkSNARK scheme for circuit satisfiability with a proof consisting of only 7 group elements. The common reference string size is linear in the circuit size, and the prover computation is quasi-linear, which makes this scheme seemingly practical. Since then, many pairing-based trusted setup zkSNARKs following a similar construction paradigm have emerged, including Pinocchio [PHGR13], DFGK scheme [DFGK14], Groth16 [Gro16], etc. Among this line of work, Groth16 stands out with a proof consisting of only 3 elliptic curve points.

However, the aforementioned schemes have a drawback in that the common reference string is specific to the circuit. If the circuit changes, the common reference string needs to be recomputed by a costly multi-party computation protocol. In 2018, Groth, Kohlweiss, Maller, Meiklejohn and Miers [GKM+18] proposed a new pairing-based trusted setup zkSNARK scheme that introduces the concept of *universal* and *updatable* common reference string[2]. Here *universal* means that the same common reference string can be used for all size-bounded circuits, and *updatable* means that the common reference string can be updated by a new party. As long as at least one party from all updaters is honest, the soundness of the scheme is ensured. The scheme proposed by [GKM+18] has inspired a lot of current work, including Sonic [MBKM19], PlonK [GWC19], Marlin [CHM+20], and others. PlonK has become the representative of this line of work and continues to inspire the creation of more schemes to this day.

These zkSNARK schemes can be instantiated over pairing-friendly elliptic curves such as Barreto-Naehrig (BN) curves [BN05] and Barreto-Lynn-Scott (BLS) curves [BLS02].

1.3 Contributions

This thesis proposes new ideas for computing multi-scalar multiplication over fixed points, driven by the objective of supporting efficient pairing-based trusted setup zkSNARK applications. The term *n-scalar multiplication over fixed points* refers to the arithmetic

$$S_{n,r} := \sum_{i=1}^{n} a_i P_i,$$

[2]Common reference string (CRS) is also called structured reference string (SRS) in some papers mentioned here in order to emphasize that there is a special structure in the string.

where every a_i $(1 \leq i \leq n)$ is an integer such $0 \leq a_i < r$, and every P_i is a fixed point in an elliptic curve group. This thesis targets at the case where n is *large*. The main contributions of the thesis can be summarized in the following three aspects.

I. Two frameworks for computing multi-scalar multiplication over fixed points and an associated accumulation algorithm.

These two frameworks are the extensions of BGMW method and Pippenger's bucket method respectively. The key idea behind the proposed frameworks is to decompose a scalar a into the following radix q representation

$$a = \sum_{j=0}^{h-1} m_j b_j q^j, \ m_j \in M, b_j \in B,$$

where q is a proper radix, h is the length of the decomposition, M is a set of integers referred to as *multiplier set*, and B is a set of integers called *bucket set*. The first framework computes n-scalar multiplication using at most approximately

$$nh + |B|$$

point additions, with the help of $|M|nh$ precomputed points. The second framework computes n-scalar multiplication using at most approximately

$$h(n + |B|)$$

point additions, with the help of $|M|n$ precomputed points.

After all points are sorted into buckets with respect to their scalars, an accumulation algorithm is utilized to add all subsums together. The original subsum accumulation Algorithm 1 employed in Pippenger's bucket method is only applicable when the scalars in the bucket set are consecutive. However, for the proposed frameworks, the scalars in the bucket set may be nonconsecutive, rendering Algorithm 1 less efficient. To address this issue, this thesis proposes a new subsum accumulation Algorithm 4 that accumulates m intermediate subsums using at most $(2m + d - 3)$ point additions, where d is the maximum difference between two neighboring elements in the bucket set.

II. Five constructions of bucket set and multiplier set that yield efficient algorithms for computing multi-scalar multiplication over fixed points when combined with the proposed frameworks.

The proposed multiplier set M is symmetric, i.e.,

$$M = \{i \mid i \in M'\} \cup \{-i \mid i \in M'\},$$

where the set M' only contains positive integers. In this case, the precomputation size can be halved by computing the inverse of a point on the fly when needed. The proposed bucket set is carefully constructed to make its size as small as possible, because a smaller bucket set would yield a faster algorithm for computing multi-scalar multiplication. The proposed constructions are summarized in Table 4.1.

Out of all the five constructions, the first three constructions would provide different trade-offs between precomputation size and time complexity for computing multi-scalar multiplication. The fourth and fifth constructions are primarily of theoretical interest, which explore the ultimate limits that the proposed frameworks could reach.

III. Two concrete methods for computing multi-scalar multiplication over fixed points in the BLS12-381 groups.

These two methods, summarized in Propositions 4 and 5, are obtained by instantiating the two proposed frameworks together with the first proposed construction of bucket set and multiplier set. We analyzed the performance of the two methods theoretically, then implemented and tested them based on the BLS12-381 curve library `blst` [bls]. When computing n-scalar multiplication over fixed points in the BLS12-381 groups, the theoretical analysis indicates that

- The proposed Method I saves 21.05%–39.77% of the point additions compared to Pippenger's bucket method for $n = 2^e$ $(10 \leq e \leq 21)$, and it saves 2.08%–9.65% of the point additions compared to BGMW method for $n = 2^e$ $(10 \leq e \leq 21)$.
- By utilizing a smaller precomputation size than Method I, the proposed Method II saves 2.59%–12.26% of the point additions compared to Pippenger's bucket method for $n = 2^e$ $(10 \leq e \leq 21)$.

The experimental results show that

- The proposed Method I saves 17.20%–40.89% of the computational time compared to Pippenger's bucket method for $n = 2^e$ $(10 \leq e \leq 21)$, and it saves 0.73%–10.21% of the computational time compared to BGMW method for $n = 2^e$ $(10 \leq e \leq 21, e \neq 16, 17)$.
- The proposed Method II saves 2.48%–11.27% of the computational time compared to Pippenger's bucket method for $n = 2^e$ $(10 \leq e \leq 21)$.

The experiment confirms the feasibility of speeding up the computation of multi-scalar multiplication over fixed points by utilizing large precomputation tables.

Original work declaration

Sections 3.1 and 3.2 in Chapter 3, all the sections in Chapter 4, all the sections in Chapter 5, and Sections 6.2 and 6.3 in Chapter 6 are my original work. Most of these work appears in the following three papers that I co-authored[3], explicitly,

- The work on the algorithm in Section 3.1, Framework I in Section 3.2, Construction I in Section 4.1, Method I in Section 5.2 and the test for Method I in Section 6.2 appears in [LFG23]
  - Guiwen Luo, Shihui Fu, and Guang Gong. Speeding Up Multi-Scalar Multiplication over Fixed Points Towards Efficient zkSNARKs. In *IACR Transactions on Cryptographic Hardware and Embedded Systems (2023)*, pages 358-380.

  This paper proposes a method for computing multi-scalar multiplication over fixed points by taking advantage of large precomputation tables.
- The work on Framework II in Section 3.2, Method II in Section 5.3, and the test for Method II in Section 6.3 appears in [LG23]
  - Guiwen Luo and Guang Gong. Fast Computation of Multi-Scalar Multiplication for Pairing-Based zkSNARK Applications. In *IEEE International Conference on Blockchain and Cryptocurrency (2023)*, pages 1-5.

  This paper provides an alternative to [LFG23] when devices' memory size is relatively limited.

[3]In these papers, the authorship is not arranged in alphabetical order, and Guiwen Luo is the first author.

- The work on Constructions II and V in Chapter 4 appears in
  - Guiwen Luo and Guang Gong. On the Optimization of Pippenger's Bucket Method with Precomputation. In *Stinson66 - New Advances in Designs, Codes and Cryptography*, accepted paper.

  This paper explores the theoretical potential of the proposed frameworks.

1.4 Outline

The thesis is organized as follows,

- Chapter 1 introduces the motivation, related work and contribution of this thesis.
- Chapter 2 reviews several popular methods for computing multi-scalar multiplication over fixed points, two pairing-based schemes that can benefit from the ideas presented in this thesis, and other fundamental knowledge.
- Chapter 3 proposes two frameworks and discusses popular techniques for computing multi-scalar multiplication over fixed points.
- Chapter 4 proposes five constructions of multiplier set and bucket set pairs that can be used with the proposed frameworks.
- Chapter 5 proposes two methods for computing multi-scalar multiplication over fixed points in the BLS12-381 groups.
- Chapter 6 implements and tests these two methods proposed in Chapter 5.
- Chapter 7 concludes the thesis and suggests possible future research topics.

Chapter 2

Preliminaries

In this chapter, we will explain the background knowledge that underpins the subsequent chapters. We will begin by introducing elliptic curve groups and bilinear pairings. Then we will explain the popular algorithms used for multi-scalar multiplication, followed by introducing two classical pairing-based schemes where the computation of multi-scalar multiplication is essential. Lastly, we will introduce the influential BLS12-381 curve, over which the experiment is conducted.

2.1 Elliptic curve group

Elliptic curves can be expressed by general Weierstrass equations [Sil09, Chapter III]. In this thesis, we always work with pairing-friendly curves, which are non-singular elliptic curves defined over a field $\mathbb{F}$ whose characteristic is neither 2 nor 3. In this case, an elliptic curve can be expressed by its short Weierstrass equation

$$y^2 = x^3 + Ax + B, \tag{2.1}$$

where $A, B \in \mathbb{F}$ are two constants such that $4A^3 + 27B^2 \neq 0$.

Although a curve is usually specified by an affine equation, we are actually working with the corresponding projective curve, which can be expressed as

$$Y^2Z = X^3 + AXZ^2 + BZ^3.$$

If $Z = 0$, then so does X, implying that the projective curve and the projective line $Z = 0$ has a unique intersection point $\infty = (0 : 1 : 0)$. Any point on the curve that is not ∞

has a nonzero Z-coordinate, which can be scaled to 1, allowing us to investigate the group operation in the affine space.

In the affine space, the elements of an elliptic curve include every point (x, y) that satisfies the affine equation in Equation (2.1), as well as the infinity point ∞ that cannot be expressed in the affine equation. For an affine curve defined over the real number field, one can treat ∞ as a formal notation and loosely think of ∞ as a point located at the end of the y-axis. The points ∞ and $-\infty$ are considered as the same point. A line passes through ∞ when it is perpendicular to the x-axis.

2.1.1 Group operation

Let $E(\mathbb{F})$ be an elliptic curve defined over the field $\mathbb{F}$,

$$E(\mathbb{F}) := \{(x, y) \in \mathbb{F}^2 \mid y^2 = x^3 + Ax + B\} \cup \{\infty\}. \tag{2.2}$$

For $P = (x, y) \in E(\mathbb{F})$, it is important to notice that the point $(x, -y)$ also lies on the curve. If we define $-P = (x, -y)$ and treat ∞ as the identity element, the group operation can be characterized by the following rule [Sil09, Chapter III]:

For $P, Q, R \in E(\mathbb{F})$, $P + Q = R$ if and only if $P, Q, -R$ lie on the same line.

It follows immediately that this operation is commutative. For $P, Q \in E(\mathbb{F})$, $P+Q = Q+P$, since the line passing through P and Q is the same as the line passing through Q and P.

A group is defined as a set of elements along with a binary operation that combines two elements to produce a third element in the set, in such a way that the set contains an identity element, every element in the set has an inverse, and the operation is associative. It can be checked that $E(\mathbb{F})$ along with the given addition operation is a commutative group, explicitly,

- ∞ is the identity element. For $P \in E(\mathbb{F})$, $P, \infty, -P$ lie on the same line, so
$$\infty + P = P.$$
- Every element has an inverse. For $P = (x, y) \in E(\mathbb{F})$, we have $-P = (x, -y) \in E(\mathbb{F})$. For $P = \infty$, we have $-P = \infty$.

- The addition operation is associative. For $P_1, P_2, P_3 \in E(\mathbb{F})$,

$$(P_1 + P_2) + P_3 = P_1 + (P_2 + P_3).$$

  Associativity is not immediately clear, therefore we would refrain from presenting the proof here. One can check [ST92, Chapter 1.2] or [Sil09, Chapter III.2] for a comprehensive proof.

Let $P \in E(\mathbb{F})$ and a be a positive integer, we define

$$aP := \underbrace{P + P + \cdots + P}_{a \text{ points}}. \tag{2.3}$$

We also define $0P = \infty$ for completeness. This operation is called *scalar multiplication*, or called *single-scalar multiplication* in order to distinguish it from multi-scalar multiplication.

Unless otherwise specified, *addition* refers to the point addition operation in elliptic curve groups in this thesis. Doubling is treated as addition for simplicity.

2.1.2 Explicit formulas for addition operation

Let $P = (x_1, y_1), Q = (x_2, y_2) \in E(\mathbb{F})$, we derive explicit formulas for

$$R = P + Q = (x_3, y_3).$$

These formulas are established by leveraging the geometric interpretation of elliptic curves, which implicitly uses the assumption that $\mathbb{F}$ is the real number field. The formulas can also be applied when $\mathbb{F}$ is a finite field.

If $P = \infty$, then $R = Q$, and if $Q = \infty$, then $R = P$. Thus we assume that P and Q are non-infinity points, and use PQ to represent the line passing through P and Q.

- If $x_1 \neq x_2$, the left sub-figure in Figure 2.1 provides a visual illustration for this case. The slope of the line PQ can be computed as

$$m = \frac{y_2 - y_1}{x_2 - x_1}. \tag{2.4}$$

  Since the line is not perpendicular to the x-axis, it will intersect with the curve $E(\mathbb{F})$ at a third affine point $-R = (x_3, -y_3)$. The equation for the line PQ can be expressed as

$$y = m(x - x_1) + y_1.$$

By substituting the equation for the line PQ into the equation for the curve, we obtain

$$(m(x - x_1) + y_1)^2 = x^3 + Ax + B,$$

which is equivalent to

$$x^3 - m^2x^2 + \cdots = 0.$$

According to Vieta's theorem for the roots of a cubic equation, we know that

$$x_1 + x_2 + x_3 = m^2,$$

thus

$$x_3 = m^2 - x_1 - x_2. \tag{2.5}$$

Because $-R$ lies on the line PQ, we know that

$$-y_3 = m(x_3 - x_1) + y_1,$$

which gives us

$$y_3 = m(x_1 - x_3) - y_1. \tag{2.6}$$

Equations (2.4)(2.5)(2.6) are the explicit formulas for point addition when $x_1 \neq x_2$.

- If $x_1 = x_2$, from the curve's equation we know $y_1^2 = y_2^2$, which means either $y_1 = y_2$ or $y_1 = -y_2$.

  (a) If $y_1 = -y_2$, we know $Q = -P$, so $R = P + Q = P + (-P) = \infty$.

  (b) If $y_1 = y_2$, then $R = P + Q = 2P$. In this case, the line PQ is the tangent to $E(\mathbb{F})$ and touches the curve at P. We can compute the slope of the tangent line by implicit differentiation, which results in the expression

  $$2y\mathrm{d}y = 3x^2\mathrm{d}x + A\mathrm{d}x,$$

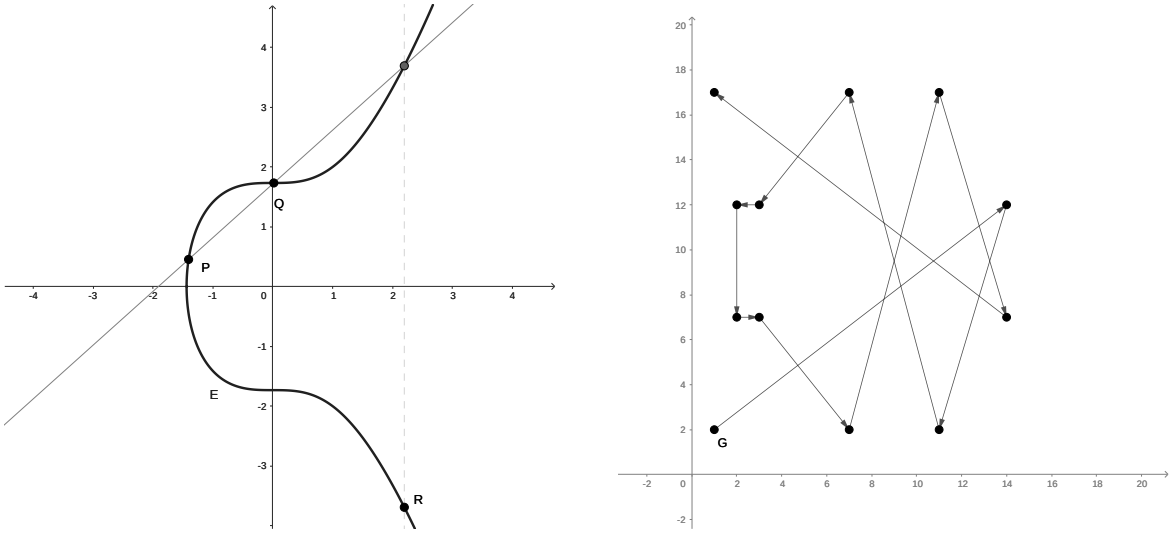
  thus

  $$m = \frac{\mathrm{d}y}{\mathrm{d}x} = \frac{3x^2 + A}{2y}. \tag{2.7}$$

  We can use this m to compute (x_3, y_3) by the same formulas presented in Equations (2.5)(2.6).

Figure 2.1 shows two curves with the same equation $E(\mathbb{F}) : y^2 = x^3 + 3$ excluding ∞. For the left curve, $\mathbb{F}$ is the real number field and $P + Q = R$. For the right curve, $\mathbb{F}$ is the finite field $\mathbb{F}_{19}$. This group happens to be a cyclic group of order 13, and $G = (1, 2)$ is a generator. The arrow points from iG to $(i + 1)G$ $(i = 1, 2, \cdots, 11)$.

Figure 2.1: Elliptic curves defined over real number field versus over finite field



2.1.3 Elliptic curve bilinear pairings

Elliptic curve bilinear pairing is a map

$$\begin{aligned} e : \mathbb{G}_1 \times \mathbb{G}_2 &\rightarrow \mathbb{G}_T, \\ (P, R) &\mapsto \mu, \end{aligned} \tag{2.8}$$

that is defined over the elliptic curve groups $\mathbb{G}_1, \mathbb{G}_2$ and the finite field $\mathbb{G}_T$. It satisfies the following properties [Men09] [EMJ17, Chapter 3],

- e is bilinear. For $P, Q \in \mathbb{G}_1$ and $R, S \in \mathbb{G}_2$, we have

  $$e(P + Q, R) = e(P, R) \cdot e(Q, R),$$

  and

  $$e(P, R + S) = e(P, R) \cdot e(P, S).$$

- e is non-degenerate. For any $P \in \mathbb{G}_1$, there exists $R \in \mathbb{G}_2$, such that $e(P, R) \neq 1$. For any $R \in \mathbb{G}_2$, there exists $P \in \mathbb{G}_1$, such that $e(P, R) \neq 1$.

A pairing e is considered appropriate for use in cryptography if e is easy to compute but difficult to invert. Inverting e means given $\mu \in \mathbb{G}_T$, find $P \in \mathbb{G}_1$ and $R \in \mathbb{G}_2$ such that $e(P, R) = \mu$.

Elliptic curve bilinear pairings can use relatively small parameters to produce elliptic curve groups $\mathbb{G}_1, \mathbb{G}_2$ and finite field $\mathbb{G}_T$ that are sufficiently large to make the corresponding discrete logarithm problems hard to solve. The bilinear property implies additive homomorphism, which means that given two ciphertexts, one can efficiently compute a ciphertext of their sum, without decrypting the ciphertexts. These features have led to the widespread use of pairings in various cryptographic primitives, such as identity-based encryptions, digital signatures, key establishment schemes, cryptographic commitments, and zkSNARKs.

2.2 Classical methods for multi-scalar multiplication

The notation $S_{n,r}$ used throughout the thesis represents the following n-scalar multiplication over fixed points,

$$S_{n,r} := a_1P_1 + a_2P_2 + \cdots + a_nP_n, \tag{2.9}$$

where each a_i $(1 \leq i \leq n)$ is a scalar such that $0 \leq a_i < r$ and each P_i $(1 \leq i \leq n)$ is a *fixed* point in an elliptic curve group.

In this section we review several classical methods for computing $S_{n,r}$, some of which are originally invented to compute single-scalar multiplication. These methods are namely the trivial method, window method, Pippenger's bucket method, BGMW method and comb method.

2.2.1 Trivial method

In the trivial method, each a_iP_i in $S_{n,r}$ is computed separately by the doubling and addition method, and then n intermediate results are added together to obtain the final result. In the worst case each scalar multiplication costs $2 \cdot (\lceil \log_2 r \rceil - 1)$ additions, the total cost for computing $S_{n,r}$ is

$$[2 \cdot (\lceil \log_2 r \rceil - 1) \cdot n + (n-1)] \approx 2n \log_2 r \tag{2.10}$$

additions. If non-adjacent form is used to represent the scalar a_i $(1 \leq i \leq n)$, then every non-zero bit has to be adjacent to two 0s, resulting in the worst case having half non-zero digits in a_i. The cost of each scalar multiplication would drop to about $(3/2)\lceil \log_2(r) \rceil$ additions. The time complexity for computing $S_{n,r}$ in the worst case is about

$$\left[\frac{3}{2}\lceil \log_2 r \rceil \cdot n + (n-1)\right] \approx \frac{3}{2} \cdot n \log_2 r \tag{2.11}$$

additions.

2.2.2 Window method

Window method is a generalization of the trivial doubling and addition method. It processes more than one bit of the scalar in each iteration.

Let $q = 2^c$, where c is a small positive integer. In order to compute $S_{n,r}$, we first precompute the following q^n points

$$\{b_1P_1 + b_2P_2 + \cdots + b_nP_n \mid 0 \le b_i < q,\ 1 \le i \le n\},$$

and then decompose each a_i into its standard q-ary representation,

$$a_i = \sum_{j=0}^{h-1} a_{ij}q^j,\ 0 \le a_{ij} < q, \tag{2.12}$$

where $h = \lceil \log_q r \rceil$. Notice that for $0 \le j \le h-1$, once

$$S_j := a_{1j}P_1 + a_{2j}P_2 + \cdots + a_{nj}P_n \tag{2.13}$$

has been precomputed, it follows that

$$\begin{aligned} S_{n,r} &= \sum_{i=1}^{n} a_iP_i \\ &= \sum_{i=1}^{n} \left(\sum_{j=0}^{h-1} a_{ij}q^j \right) P_i \\ &= \sum_{j=0}^{h-1} q^j \left(\sum_{i=1}^{n} a_{ij}P_i \right) \\ &= \sum_{j=0}^{h-1} q^j S_j. \end{aligned} \tag{2.14}$$

This can be computed by a method similar to Horner's rule,

$$S_{n,r} = \sum_{j=0}^{h-1} q^j S_j = S_0 + q(S_1 + \cdots + q(S_{h-2} + qS_{h-1}) \cdots). \tag{2.15}$$

qS_j is computed by $c = \log_2 q$ additions (doubling is treated as addition for simplicity). Equation (2.15) can thus be evaluated with at most $(h-1)(c+1)$ additions.

The aforementioned method is only suitable for small n, because the precomputation size would be exponentially large when n goes large. One variant that can be used for large n is to precompute only the following $n \cdot (q-1)$ points,

$$\{aP_i \mid 1 \le a < q,\ 1 \le i \le n\}.$$

Instead of directly retrieving S_j from the precomputation table, for $0 \le j \le h-1$, we can compute

$$S_j = a_{1j}P_1 + a_{2j}P_2 + \cdots + a_{nj}P_n$$

on the fly using at most $n-1$ additions. The cost for computing $S_{n,r}$ is thus

$$h(n-1) + (h-1)(c+1) = h(n+c) - c - 1 \approx h(n+c) \tag{2.16}$$

additions, with the help of

$$n(q-1) \tag{2.17}$$

precomputed points.

2.2.3 Pippenger's bucket method

Here we introduce Pippenger's bucket method interpreted in [BDLO12, Section 4], which is an application of Pippenger's algorithm [Pip76]. This interpretation was initially used for computing multi-scalar multiplication in the context of large batch signature verification.

When r is small enough, we have

$$\begin{aligned} S_{n,r} &= \sum_{i=1}^{n} a_i P_i \\ &= \sum_{i=1}^{n} \left(\sum_{k=1}^{r-1} k \cdot \sum_{i \ s.t.\ a_i = k} P_i \right) \\ &= \sum_{k=1}^{r-1} k \cdot \left(\sum_{1 \le i \le n,\ a_i = k} P_i \right). \end{aligned} \tag{2.18}$$

Define the intermediate subsum (it is also called *bucket* sum) S_k,

$$S_k := \sum_{1 \le i \le n,\ a_i = k} P_i,\ 1 \le k < r, \tag{2.19}$$

then we know

$$S_{n,r} = \sum_{k=1}^{r-1} kS_k. \tag{2.20}$$

$S_{n,r}$ is computed by first evaluating all S_k $(1 \leq k < r)$ using at most $n-(r-1)$ additions, because there are n points that are sorted into $r-1$ subsums, then by using Algorithm 1 with at most $2(r-2)$ additions to complete the computation. The correctness of Algorithm 1 is ensured by the following equation,

$$\sum_{k=1}^{m} kS_k = \sum_{k=1}^{m}\sum_{j=1}^{k} S_k = \sum_{j=1}^{m}\sum_{k=j}^{m} S_k.$$

To sum up, when r is small, the cost for computing $S_{n,r}$ is at most

$$n + r - 3 \tag{2.21}$$

additions.

Algorithm 1 Subsum accumulation algorithm I

Input: $S_1, S_2, \cdots, S_m$.
Output: $1S_1 + 2S_2 + \cdots + mS_m$.

```
1: tmp = 0
2: ret = 0
3: for k = m to 1 do
4:     tmp = tmp + S_k
5:     ret = ret + tmp
6: return ret
```

Example. Let us present a toy example that computes the following 13-scalar multiplication to illustrate Pippenger's bucket method,

$$\begin{aligned} S_{13,4} = {} & 2P_1 + 3P_2 + 3P_3 + 2P_4 + 1P_5 + 1P_6 + \\ & 3P_7 + 2P_8 + 2P_9 + 3P_{10} + 1P_{11} + 3P_{12} + 1P_{13}. \end{aligned}$$

First, all the points are sorted into 3 buckets according to their scalars,

$$\begin{aligned} S_{13,4} &= 1 \cdot (P_5 + P_6 + P_{11} + P_{13}) + 2 \cdot (P_1 + P_4 + P_8 + P_9) + \\ &\quad 3 \cdot (P_2 + P_3 + P_7 + P_{10} + P_{12}) \\ &= 1S_1 + 2S_2 + 3S_3. \end{aligned}$$

Here $S_1 = P_5 + P_6 + P_{11} + P_{13}$, $S_2 = P_1 + P_4 + P_8 + P_9$, and $S_3 = P_2 + P_3 + P_7 + P_{10} + P_{12}$. S_1, S_2, S_3 are evaluated using 10 additions. Then

$$1S_1 + 2S_2 + 3S_3 = S_3 + (S_3 + S_2) + (S_3 + S_2 + S_1),$$

which can be evaluated using 4 additions. The computation of $S_{13,4}$ requires 14 additions.

In order to compute $S_{n,r}$ when r is large (for example, in practice r is at least 2^{256} for achieving 128-bit security), Pippenger's bucket method proceeds similarly to window method by first breaking down each scalar into the standard q-ary representation,

$$a_i = \sum_{j=0}^{h-1} a_{ij} q^j, \ 0 \leq a_{ij} < q, \tag{2.22}$$

where $h = \lceil \log_q(r) \rceil$. The difference is that Pippenger's bucket method computes every

$$S_j := a_{1,j}P_1 + a_{2,j}P_2 + \cdots + a_{n,j}P_n \ (0 \leq j \leq h-1) \tag{2.23}$$

by the aforementioned method (treating q as the small r) using at most $n+q-3$ additions, because S_j is an n-scalar multiplication where every scalar is smaller than q. We thus have

$$\begin{aligned} S_{n,r} &= \sum_{i=1}^{n} a_i P_i \\ &= \sum_{i=1}^{n} \left(\sum_{j=0}^{h-1} a_{ij} q^j \right) P_i \\ &= \sum_{j=0}^{h-1} q^j \left(\sum_{i=1}^{n} a_{ij} P_i \right) \\ &= \sum_{j=0}^{h-1} q^j S_j, \end{aligned} \tag{2.24}$$

which again can be evaluated by a method similar to Horner's rule, as shown in Equation (2.15), with $(h-1)(c+1)$ additions. The computation of $S_{n,r}$ would take at most

$$h(n+q-3) + (h-1)(c+1) \approx h(n+q) \tag{2.25}$$

additions. Compared to Equation (2.16), at first glance it seems that Pippenger's bucket method is less efficient than window method, but this might not be true for large n. Because Pippenger's bucket method does not require precomputation, a bigger q can be selected to minimize the computational cost.

Further optimization

If the parameter r in $S_{n,r}$ is the order of the corresponding elliptic curve group[1], Pippenger's bucket method can be further optimized by halving the number of buckets. For a radix $q = 2^c$, using the observation that in an elliptic curve group $-P$ can be obtained from $P = (x, y)$ by taking the negative of its y coordinate at almost no cost, all the buckets can be restricted to the scalars that are no more than $q/2$. Algorithm 2 is used to substitute a scalar with the representation where every digit is in the interval of $[-q/2, q/2]$. The time complexity of Pippenger's bucket method would thus drop to

$$h\left(n + \frac{q}{2}\right) \tag{2.26}$$

additions.

Algorithm 2 Scalar conversion I

Input: a, q, r, such that $0 \leq a < r$ and $rP = \infty$, where P is an elliptic curve point.

Output: $\{b_j\}_{0 \leq j \leq h-1}$, such that $-q/2 \leq b_j \leq q/2$ and $aP = \left(\sum_{j=0}^{h-1} b_j q^j\right) \cdot P$.

$\mathtt{h} = \lceil \log_{\mathtt{q}} \mathtt{r} \rceil$, `condition` $= (\mathtt{a} > (\mathtt{q}^{\mathtt{h}}/2))$
if `condition` is `true` **then**
    $\mathtt{a} = \mathtt{r} - \mathtt{a}$
Express $\mathtt{a}$ as its standard q-ary form, $\mathtt{a} = \sum_{\mathtt{j}=0}^{\mathtt{h}-1} \mathtt{a_j} \mathtt{q}^{\mathtt{j}}$, where $0 \leq \mathtt{a_j} < \mathtt{q}$
for $\mathtt{j} = 0$ to $\mathtt{h} - 2$ by 1 **do**
    if $\mathtt{a_j} \leq \mathtt{q}/2$ **then**
        $\mathtt{b_j} = \mathtt{a_j}$
    else
        $\mathtt{b_j} = \mathtt{a_j} - \mathtt{q}$
        $\mathtt{a_{j+1}} = \mathtt{a_{j+1}} + 1$
$\mathtt{b_{h-1}} = \mathtt{a_{h-1}}$
if `condition` is `true` **then**
    for $\mathtt{j} = 0$ to $\mathtt{h} - 1$ by 1 **do**
        $\mathtt{b_j} = -\mathtt{b_j}$
return $\{\mathtt{b_j}\}_{0 \leq \mathtt{j} \leq \mathtt{h}-1}$

Let us conclude the introduction of Pippenger's bucket method by demonstrating the correctness of Algorithm 2.

[1]If the constraint in Equation (2.27) holds true, this condition can be removed, as discussed in the proof of the correctness of Algorithm 2.

Proof. By the definition of h, we know

$$q^{h-1} < r \leq q^h.$$

If

$$q^{h-1} < r \leq \frac{q}{2} \cdot q^{h-1}, \tag{2.27}$$

then the scalar a would always be less than $q^h/2$, and Algorithm 2 would skip Steps 2−3, 12−14. From Steps 5−10, we know $-q/2 \leq b_j \leq q/2$ for $0 \leq j \leq h-2$. The assumption in Equation (2.27) ensures $a < q^h/2$, it follows that $a_{h-1} \leq q/2 - 1$, and $b_{h-1} \leq q/2$ considering the possible carry bit from a_{h-2}. From Steps 5−11, one can easily check that $a = \sum_{j=0}^{h-1} b_j q^j$, thus

$$aP = \left(\sum_{j=0}^{h-1} b_j q^j \right) \cdot P.$$

This analysis suggests that when the parameters r, q, h for computing $S_{n,r}$ satisfy Equation (2.27), Algorithm 2 can be simplified by omitting Steps 2−3 and 12−14.

If the assumption in Equation (2.27) does not hold, then $a < q^h/2$, $a = q^h/2$, and $a > q^h/2$ are all possible.

- If $a < q^h/2$, it boils down to the case where Equation (2.27) holds.
- If $a = q^h/2$, Algorithm 2 will return $b_j = 0$ $(0 \leq j \leq h-2)$ and $b_{h-1} = q/2$.
- If $a > q^h/2$, then the `condition` variable is true. The original a is replaced by $r - a$, which would ensure $-q/2 \leq b_j \leq q/2$ for $0 \leq j \leq h-1$. After Step 11, we have

  $$r - a = \sum_{j=0}^{h-1} b_j q^j.$$

  After Step 14, we have

  $$a - r = \sum_{j=0}^{h-1} b_j q^j.$$

  Because $rP = \infty$, it follows that

  $$aP = (a - r) \cdot P = \left(\sum_{j=0}^{h-1} b_j q^j \right) \cdot P.$$

□

2.2.4 BGMW method

In the aforementioned Pippenger's bucket method, one downside is that Algorithm 1 runs h times. If a precomputation table is used, a variant of Pippenger's bucket method called BGMW method can be used to circumvent this shortcoming [BGMW92].

Let us choose a radix $q = 2^c$ and decompose a_i $(1 \leq i \leq n)$ into its q-ary form

$$a_i = \sum_{j=0}^{h-1} a_{ij} q^j, \ 0 \leq a_{ij} < q, \tag{2.28}$$

where $h = \lceil \log_q r \rceil$. It follows that

$$\begin{aligned} S_{n,r} &= \sum_{i=1}^{n} a_i P_i \\ &= \sum_{i=1}^{n} \left(\sum_{j=0}^{h-1} a_{ij} q^j \right) P_i \\ &= \sum_{i=1}^{n} \sum_{j=0}^{h-1} a_{ij} \cdot q^j P_i. \end{aligned} \tag{2.29}$$

If the following nh points

$$\{q^j P_i \mid 1 \leq i \leq n, 0 \leq j \leq h-1\}$$

are precomputed, then $S_{n,r}$ boils down to an nh-scalar multiplication where all scalars are smaller than q. It can be computed by invoking Algorithm 1 only once. By Equation (2.21), the computation for $S_{n,r}$ requires at most

$$nh + q - 3 \tag{2.30}$$

additions.

Further optimization

Silimar to Pippenger's bucket method, BGMW method can also be further optimized by restricting all the buckets to scalars that are no more than $q/2$ if r in $S_{n,r}$ is the order

of the corresponding elliptic curve group. The time complexity of BGMW method would become approximately

$$nh + \frac{q}{2} \tag{2.31}$$

additions, with the help of the following nh precomputed points

$$\{q^j P_i \mid 1 \leq i \leq n, 0 \leq j \leq h-1\}. \tag{2.32}$$

Henceforward, when mentioning Pippenger's bucket method and BGMW method, we refer to the algorithms whose time complexities are given by Equations (2.26) and (2.31), respectively.

2.2.5 Comb method

Comb method was first proposed by Lim and Lee [LL94] to compute the exponentiation for a fixed element in a given group. For computing single-scalar multiplication, this method exhibits greater flexibility compared to BGMW method because it decomposes the scalar twice. We will show that this flexibility disappears for computing multi-scalar multiplication. Here we first present the original comb method for computing single-scalar multiplication, and then propose its natural extension for computing multi-scalar multiplication.

Let us first investigate single-scalar multiplication aP, where P is a fixed point, and $0 \leq a < r$. For $q = 2^c,\ p = 2^{c_1},\ c \geq c_1$, we decompose a twice as follows,

$$\begin{aligned} a &= \sum_{j=0}^{h-1} a_j q^j \qquad (0 \leq a_j < q) \\ &= \sum_{j=0}^{h-1} \sum_{k=0}^{v-1} a_{jk} p^k q^j \qquad (0 \leq a_{jk} < p), \end{aligned}$$

where $h = \lceil \log_q r \rceil,\ v = \lceil \log_p q \rceil = \lceil c/c_1 \rceil$. If we further write a_{jk} into its binary form,

$$a_{jk} = \sum_{\ell=0}^{c_1-1} e_{jk\ell} 2^\ell,\ e_{jk\ell} \in \{0, 1\}, \tag{2.33}$$

it follows that

$$\begin{aligned} aP &= \sum_{j=0}^{h-1}\sum_{k=0}^{v-1}\sum_{\ell=0}^{c_1-1} e_{jk\ell} 2^\ell p^k q^j \cdot P \\ &= \sum_{\ell=0}^{c_1-1} 2^\ell \left(\sum_{k=0}^{v-1}\sum_{j=0}^{h-1} e_{jk\ell} p^k q^j \cdot P \right) \\ &=: \sum_{\ell=0}^{c_1-1} 2^\ell \left(\sum_{k=0}^{v-1} P_{k\ell} \right) \\ &=: \sum_{\ell=0}^{c_1-1} 2^\ell S_\ell, \end{aligned} \tag{2.34}$$

where $P_{k\ell} := \sum_{j=0}^{h-1} e_{jk\ell} p^k q^j P$, and $S_\ell := \sum_{k=0}^{v-1} P_{k\ell}$. When $k = v-1$, a_{jk} may not be full of c_1 bits. Therefore, $e_{jkl} = 0$ if $c - (v-1)c_1 \le \ell \le c_1 - 1$. It follows that

$$P_{k\ell} = \sum_{j=0}^{h-1} 0 \cdot p^k q^j P = \infty, \text{ if } k = v-1,\ c-(v-1)c_1 \le \ell \le c_1 - 1.$$

We can precompute the following $(2^h - 1)v$ points (excluding these points associated with $\sum_{j=0}^{h-1} e_j = 0$, which are infinity)

$$\left\{ \sum_{j=0}^{h-1} e_j p^k q^j P \;\middle|\; e_j \in \{0,1\},\ \sum_{j=0}^{h-1} e_j \neq 0,\ 0 \le k \le v-1 \right\}. \tag{2.35}$$

In order to compute aP in Equation (2.34), every $P_{k\ell}$ can be directly retrieved from the precomputation table. Therefore, all S_ℓ's are computed using

$$c_1(v-1) - (c_1 v - c)$$

additions because there are $(c_1 v - c)$ $P_{k\ell}$'s that are infinity. The remaining part is $\sum_{\ell=0}^{c_1-1} 2^\ell S_\ell$, which can be computed by the method presented in Equation (2.15) using

$$2(c_1 - 1)$$

additions. Thus the number of additions required to compute aP is at most

$$c_1(v-1) - (c_1 v - c) + 2(c_1 - 1) = c + c_1 - 2. \tag{2.36}$$

Compared to BGMW method, comb method provides more flexibility for the trade-off between time and memory. One can check the appendix in [LL94] for the detailed comparison between BGMW method and comb method when it comes to the computation for single-scalar multiplication.

Multi-scalar multiplication variant

We follow the same methodology to propose an extended comb method for computing the multi-scalar multiplication over fixed points

$$S_{n,r} = \sum_{i=1}^{n} a_i P_i,\ 0 \le a_i < r.$$

For $q = 2^c,\ p = 2^{c_1},\ c \ge c_1$,

$$a_i = \sum_{j=0}^{h-1} a_{ij} q^j = \sum_{j=0}^{h-1} \sum_{k=0}^{v-1} a_{ijk} p^k q^j.$$

If we further write a_{ijk} into its binary form

$$a_{ijk} = \sum_{\ell=0}^{c_1-1} e_{ijk\ell} 2^\ell,\ e_{ijk\ell} \in \{0,1\},$$

then

$$a_i P_i = \sum_{j=0}^{h-1} \sum_{k=0}^{v-1} \sum_{\ell=0}^{c_1-1} e_{ijk\ell} 2^\ell p^k q^j P_i.$$

Therefore, we have

$$\begin{aligned} S_{n,r} = \sum_{i=1}^{n} a_i P_i &= \sum_{i=1}^{n} \sum_{j=0}^{h-1} \sum_{k=0}^{v-1} \sum_{\ell=0}^{c_1-1} e_{ijk\ell} 2^\ell p^k q^j P_i \\ &= \sum_{\ell=0}^{c_1-1} 2^\ell \left(\sum_{k=0}^{v-1} \sum_{i=1}^{n} \sum_{j=0}^{h-1} e_{ijk\ell} p^k q^j P_i \right) \\ &=: \sum_{\ell=0}^{c_1-1} 2^\ell \left(\sum_{k=0}^{v-1} \sum_{i=1}^{n} P_{ik\ell} \right) \\ &=: \sum_{\ell=0}^{c_1-1} 2^\ell S_\ell, \end{aligned} \tag{2.37}$$

where $P_{ik\ell} := \sum_{j=0}^{h-1} e_{ijk\ell} p^k q^j P_i$ and $S_\ell = \sum_{k=0}^{v-1} \sum_{i=1}^{n} P_{ik\ell}$.

We can precompute the following $(2^h - 1)vn$ points (excluding these points associated with $\sum_{j=0}^{h-1} e_j = 0$, which are infinity)

$$\left\{ \sum_{j=0}^{h-1} e_j p^k q^j P_i \;\middle|\; e_j \in \{0,1\},\ \sum_{j=0}^{h-1} e_j \neq 0,\ 1 \leq i \leq n,\ 0 \leq k \leq v-1 \right\}. \tag{2.38}$$

The values of $P_{ik\ell}$ can be directly retrieved from the precomputation table, therefore all S_ℓ's can be computed by at most $c_1(nv-1)$ additions. Following a similar analysis as the single-scalar multiplication case, there are $n(c_1 v - c)$ $P_{ik\ell}$'s that are infinity. The remaining part can be computed by at most $2(c_1 - 1)$ additions. Therefore, the computation of $S_{n,r}$ requires at most

$$c_1(vn-1) - n(c_1 v - c) + 2(c_1 - 1) = cn + c_1 - 2 \approx cn \tag{2.39}$$

additions.

In the multi-scalar multiplication case, the main term in time complexity is cn. The flexibility introduced by further decomposing each scalar into p-ary representation disappears, so we can always set $c_1 = c$ and $v = 1$ to minimize the precomputation size.

2.2.6 Comparison of multi-scalar multiplication algorithms

We summarize in Table 2.1 the precomputation sizes and worst case time complexities for computing $S_{n,r}$ by the aforementioned methods, together with the proposed methods in Chapter 4. In the table, $q = 2^c$, $h = \lceil \log_q r \rceil$. Radixes q can be selected to minimize the corresponding computational complexities. P_E represents the memory size of one point, A_E refers to the addition operation in elliptic curve groups. Additionally, the following conditions should be satisfied in order to achieve the alleged time complexities.

- For Pippenger's bucket method and BGMW method, $r \leq q/2 \cdot q^{h-1}$ or r is the order of the corresponding elliptic curve group.
- For the proposed Proposition 1, $q = 2^c$ $(10 \leq c \leq 31)$ and r/q^h is small.
- For the proposed Proposition 2, $r \leq q/2 \cdot q^{h-1}$ or r is the order of the corresponding elliptic curve group.
- For the proposed Proposition 3, $r \leq q/4 \cdot q^{h-1}$.

When n is small and the precomputation memory is sufficiently large, window method and comb method might be faster than other methods.

Table 2.1: Comparison of different methods for computing $S_{n,r}$

Method	Precomputation size	Worst case complexity
Trivial method	$n \cdot P_E$	$3/2 \cdot (n \log_2 r) \cdot A_E$
Window method	$n(2^c - 1) \cdot P_E$	$h(n + c) \cdot A_E$
Pippenger's bucket method [Pip76]	$n \cdot P_E$	$h(n + q/2) \cdot A_E$
BGMW method [BGMW92]	$nh \cdot P_E$	$(nh + q/2) \cdot A_E$
Comb method [LL94]	$n(2^h - 1) \cdot P_E$	$cn \cdot A_E$
Proposition 1 in Section 4.1	$3nh \cdot P_E$	$(nh + 0.21q) \cdot A_E$
	$3n \cdot P_E$	$h(n + 0.21q) \cdot A_E$
Proposition 2 in Section 4.2	$2nh \cdot P_E$	$(nh + q/3) \cdot A_E$
	$2n \cdot P_E$	$h(n + q/3) \cdot A_E$
Proposition 3 in Section 4.3	$2nh \cdot P_E$	$(nh + 5q/16) \cdot A_E$
	$2n \cdot P_E$	$h(n + 5q/16) \cdot A_E$

2.3 Pairing-based trusted setup schemes

In this section, we will review two influential pairing-based trusted setup schemes that would benefit from efficient multi-scalar multiplication algorithms, namely the KZG polynomial commitment and the Groth16 zkSNARK scheme. KZG commitment is the cornerstone of many zkSNARK schemes, while Groth16 is an iconic scheme in zero-knowledge proof community.

KZG polynomial commitment is *universal*, in the sense that the same common reference string can be used to commit to any polynomial as long as its degree is bounded and supported by the common reference string. Thus zkSNARK schemes built upon KZG commitment inherit the universal property and can be applied to any circuit-satisfiability instance as long as the circuit size is bounded and supported (for example, PlonK). On the contrary, Groth16 is circuit-specific.

2.3.1 KZG polynomial commitment

A polynomial commitment scheme enables a prover to commit to a polynomial using a concise string that a verifier can later utilize to verify the alleged evaluations of the committed polynomial. KZG commitment is a scheme to commit to a univariate polynomial $f(X) \in \mathbb{Z}_p[X]$, with the commitment string being a single elliptic curve point [KZG10]. The term *commitment* is used because once the prover sends the commitment string (an elliptic curve point in KZG commitment) to the verifier, the prover is unable to modify the polynomial afterward.

KZG commitment utilizes the following observation. Given a polynomial $f(X) \in \mathbb{Z}_p[X]$, a point (x_0, y_0) is on f if and only if $X - x_0$ divides $f(X) - y_0$. Let us assume that we want to commit to a polynomial $f(X) \in \mathbb{Z}_p[X]$ with its degree $\deg(f) \leq \ell$. KZG commitment consists of six algorithms $\mathsf{C} = (\mathsf{Setup}, \mathsf{Commit}, \mathsf{Open}, \mathsf{VerifyPoly}, \mathsf{CreateWitness}, \mathsf{VerifyEval})$:

- $\mathtt{crs} \leftarrow \mathsf{Setup}(1^\lambda, \ell)$: Taking as input the security parameter 1^λ and the maximum degree of polynomial ℓ, it outputs a common reference string $\mathtt{crs}$ that provides λ-bit security. Let $\mathbb{G}_1$, $\mathbb{G}_2$, $\mathbb{G}_T$ be groups of prime order p over which the pairing is defined. Let $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ be the asymmetric bilinear pairing, and G, H be the generators of $\mathbb{G}_1$, $\mathbb{G}_2$ respectively. Let the uniformly and randomly picked $\alpha \in \mathbb{Z}_p^*$ be a trapdoor generated by a trusted authority. Then
$$\mathtt{crs} := \left(e,\ \{\alpha^i G\}_{0 \leq i \leq \ell},\ \{\alpha^i H\}_{0 \leq i \leq \ell}\right).$$
- $\mathtt{com} \leftarrow \mathsf{Commit}(\mathtt{crs}, f)$: Taking as input the common reference string $\mathtt{crs}$ and a polynomial $f(X) \in \mathbb{Z}_p[X]$ of degree ℓ or less, it outputs the commitment $\mathtt{com}$ to $f(X)$. If $f = \sum_{i=0}^{\deg(f)} f_i X^i$, then the commitment to $f(X)$ is given by
$$\mathtt{com} := f(\alpha)G = \sum_{i=0}^{\deg(f)} f_i \cdot \alpha^i G. \tag{2.40}$$
- $f \leftarrow \mathsf{Open}(\mathtt{crs}, \mathtt{com}, f)$: It outputs the polynomial f.
- $0/1 \leftarrow \mathsf{VerifyPoly}(\mathtt{crs}, \mathtt{com}, f)$: For a polynomial $f = \sum_{i=0}^{\deg(f)} f_i X^i$ with degree no more than ℓ, it verifies that $\mathtt{com}$ is a commitment to f. If the following equation
$$\mathtt{com} = \sum_{i=0}^{\deg(f)} f_i \cdot \alpha^i G$$
holds true, it outputs 1, otherwise it outputs 0.

- $(x_0, y_0, \pi) \leftarrow \mathsf{CreateWitness}(\mathtt{crs}, x_0, f)$: It first computes $y_0 = f(x_0)$ and

$$q(X) = \frac{f(X) - y_0}{X - x_0},$$

then it computes

$$\pi := q(\alpha)G$$

by a multi-scalar multiplication over the points in $\mathtt{crs}$. the point π will serve as an opening proof asserting that $f(x_0) = y_0$.

- $0/1 \leftarrow \mathsf{VerifyEval}(\mathtt{crs}, \mathtt{com}, x_0, y_0, \pi)$: Taking as input the common reference string $\mathtt{crs}$, commitment $\mathtt{com}$, a point $(x_0, y_0) \in \mathbb{Z}_p^2$, and the proof π, it verifies that y_0 is the evaluation at x_0 of the polynomial committed to by $\mathtt{com}$. If the following equation

$$e(\pi, \alpha H - x_0 H) \cdot e(G, H)^{y_0} = e(\mathtt{com}, H) \tag{2.41}$$

holds true, it outputs 1, otherwise it outputs 0.

It should be noted that the verification Equation (2.41) presented here is directly from [KZG10]. Others (for example, Gabizon) also employ the following verification equation,

$$e(\pi, \alpha H - x_0 H) = e(\mathtt{com} - y_0 G, H).$$

Both of them compute 2 pairings because $e(G, H)$ in Equation (2.41) can be precomputed.

One may employ multi-party computation protocols [BCG+15, BGG18] to generate the trapdoor α in Setup. The value α should be forgotten after the common reference string $\mathtt{crs}$ is computed. If the committed polynomial does not change, the commitment $\mathtt{com}$ also remains the same, in this situation $e(\mathtt{com}, H)$ can also be precomputed.

A polynomial commitment is said to be *correct*, if the honest output of Open and $\mathsf{CreateWitness}$ can be successfully verified. It is said to be *polynomial binding*, if an adversary cannot output two different polynomials f and f' that are both accepted by $\mathsf{VerifyPoly}$. It is said to be *evaluation binding*, if an adversary cannot compute two evaluation tuples (x_0, y_0, π) and (x_0, y_0', π') that are both accepted by $\mathsf{VerifyEval}$. It is said to be *hiding*, if an adversary, given k valid evaluation tuples $(x_i, f(x_i), \pi_i)$ for $1 \leq i \leq k,\ k \leq \deg(f)$, cannot correctly determine $f(x')$ for $x' \neq x_i, 1 \leq i \leq k$.

If the discrete logarithm assumption and ℓ-Strong Diffie-Hellman assumption [TS10] hold true, then the KZG commitment scheme presented above is correct, polynomial binding, evaluation binding and hiding [KZG10, Theorem 1].

One can observe that multi-scalar multiplication is the essential operation in Commit, $\mathsf{VerifyPoly}$ and $\mathsf{CreateWitness}$.

Batching opening

KZG commitment also supports batching opening, which means that a set of evaluations is verified at the same time by providing an opening proof of only one point. In the batching situation, suppose we have k points $\{(x_i, y_i)\}_{1\leq i\leq k}$, we can employ Lagrange interpolation to construct a polynomial $I(X) \in \mathbb{Z}_p[X]$ with degree smaller than k such that $I(x_i) = y_i$ for $1 \leq i \leq k$. Then all the points $\{(x_i, y_i)\}_{1\leq i\leq k}$ lie on f if and only if $\prod_{i=1}^{k}(X - x_i)$ divides $f(X) - I(X)$. The batching opening version KZG commitment has the same Setup, Commitment, Open and VerifyPoly components, with the following BatchCreateWitness and BatchVerifyEval:

- $(\{(x_i, y_i)\}_{1\leq i\leq k}, \pi) \leftarrow \mathsf{BatchCreateWitness}(\mathtt{crs}, \{x_i\}_{1\leq i\leq k}, f)$: It requires $k \leq \deg(f)$. It first computes $y_i = f(x_i)$ for $1 \leq i \leq k$ and

  $$I(X) = \sum_{i=1}^{k} y_i \cdot \prod_{j=1,\ j\neq i}^{k} \frac{X - x_j}{x_i - x_j}, \quad Z(X) = \prod_{i=1}^{k}(X - x_i),$$

  then it computes

  $$q(X) = \frac{f(X) - I(X)}{Z(X)},$$

  finally it computes

  $$\pi := q(\alpha)G$$

  by a multi-scalar multiplication over the points in $\mathtt{crs}$.

- $0/1 \leftarrow \mathsf{BatchVerifyEval}(\mathtt{crs}, \mathtt{com}, \{(x_i, y_i)\}_{1\leq i\leq k}, \pi)$: It requires $k \leq \deg(f)$. It verifies that y_i is the evaluation at x_i of the polynomial committed to by $\mathtt{com}$ for $1 \leq i \leq k$. In order to verify this, it first computes

  $$I(X) = \sum_{i=1}^{k} y_i \cdot \prod_{j=1, j\neq i}^{k} \frac{X - x_j}{x_i - x_j}, \quad Z(X) = \prod_{i=1}^{k}(X - x_i),$$

  and then computes $I(\alpha) \cdot G$ and $Z(\alpha) \cdot H$ by multi-scalar multiplications over the points in $\mathtt{crs}$. If the following equation

  $$e(\pi, Z(\alpha)H) \cdot e(I(\alpha)G, H) = e(\mathtt{com}, H)$$

  holds true, it outputs 1, otherwise it outputs 0.

One can see that multi-scalar multiplications are involved in BatchCreateWitness and BatchVerifyEval.

2.3.2 Groth16 zkSNARK scheme

Groth16 zkSNARK scheme adopts the paradigm where the prover computes a proof consisting of several elliptic curve points using multi-scalar multiplications, while the verifier verifies the proof by checking several equations that involve multi-scalar multiplications and elliptic curve pairings.

While a computational problem can typically be formulated using a high level programming language, a zkSNARK scheme usually requires it to be expressed using a set of algebraic constraints. Groth16 is no exception. In Groth16, a problem is first converted into an arithmetic circuit, which is then further converted into a R1CS (Rank-1 Constraint System, a widely-used NP-complete language), and the R1CS is then converted into a quadratic arithmetic program over which the Groth16 scheme will be applied.

Groth16 is a pairing-based proof system for the following quadratic arithmetic programs depicted as the binary relations of the form

$$R = \left(\mathbb{Z}_p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e, \ell, \{u_i(X), v_i(X), w_i(X)\}_{0\le i\le m}, t(X)\right), \tag{2.42}$$

where p is a prime and the order of $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T$, $e : \mathbb{G}_1 \times \mathbb{G}_2 \mapsto \mathbb{G}_T$ is a bilinear pairing, $\{u_i(X), v_i(X), w_i(X)\}_{0\le i\le m}$ are degree $n-1$ polynomials, and $t(X)$ is a degree n polynomial.

The relation R defines a language of public statements $a = (a_0, a_1, \ldots, a_\ell) \in \mathbb{Z}_p^{\ell+1}$ with $a_0 = 1$, and private witnesses $w = (a_{\ell+1}, \ldots, a_m) \in \mathbb{Z}_p^{m-\ell}$, (a, w) is in the language if and only if

$$\sum_{i=0}^{m} a_i u_i(X) \cdot \sum_{i=0}^{m} a_i v_i(X) = \sum_{i=0}^{m} a_i w_i(X) + h(X)t(X) \tag{2.43}$$

for some degree $n-2$ quotient polynomial $h(X)$, where n is the degree of $t(X)$.

The relation R captures an arithmetic circuit with n multiplication gates and m wires. The polynomials $\{u_i(X), v_i(X), w_i(X)\}_{0\le i\le m}$ and $t(X)$ are decided by the specific arithmetic circuit. For a given public statement a, a prover would employ Groth16 to prove that he knows a witness w such that (a, w) satisfies Equation (2.43) without revealing w.

Groth16 is the following scheme consisting of (Setup, Prove, Verify):

- $\sigma \leftarrow \mathsf{Setup}(R)$: Taking as input the description of the binary relation R presented in Equation (2.42), it first picks generators $G \in \mathbb{G}_1$, $H \in \mathbb{G}_2$, and then uniformly and randomly picks $\alpha, \beta, \gamma, \delta, x \in \mathbb{Z}_p^*$. It outputs common reference string σ computed as follows,

$$
\begin{aligned}
\sigma_1 = &\left(\alpha G, \beta G, \delta G, \left\{ x^i G \right\}_{0 \le i \le n-1}, \left\{ \frac{\beta u_i(x) + \alpha v_i(x) + w_i(x)}{\gamma} \cdot G \right\}_{0 \le i \le \ell}, \right. \\
&\left. \left\{ \frac{\beta u_i(x) + \alpha v_i(x) + w_i(x)}{\delta} \cdot G \right\}_{\ell+1 \le i \le m}, \left\{ \frac{x^i t(x)}{\delta} \cdot G \right\}_{0 \le i \le n-2} \right), \\
\sigma_2 = &\left(\beta H, \gamma H, \delta H, \left\{ x^i H \right\}_{0 \le i \le n-1} \right), \\
\sigma = &\left(\sigma_1, \sigma_2 \right).
\end{aligned}
\tag{2.44}
$$

- $\pi \leftarrow \mathsf{Prove}(R, \sigma, a, w)$: It uniformly and randomly chooses $r, s \in \mathbb{Z}_p$, and then it computes

$$
\begin{aligned}
U(X) = &\sum_{i=0}^{m} a_i u_i(X), \\
V(X) = &\sum_{i=0}^{m} a_i v_i(X), \\
W(X) = &\sum_{i=0}^{m} a_i w_i(X), \\
M(X) = &sU(X) + rV(X), \\
h(X) = &\frac{U(X)V(X) - W(X)}{t(X)}.
\end{aligned}
\tag{2.45}
$$

  Let us denote the coefficients of each polynomial using its name and the indexes, explicitly,

$$
\begin{aligned}
U(X) = &\sum_{i=0}^{n-1} U_i X^i, \ U_i \in \mathbb{Z}_p, \\
V(X) = &\sum_{i=0}^{n-1} V_i X^i, \ V_i \in \mathbb{Z}_p, \\
M(X) = &\sum_{i=0}^{n-1} M_i X^i, \ M_i \in \mathbb{Z}_p, \\
h(X) = &\sum_{i=0}^{n-2} h_i X^i, \ h_i \in \mathbb{Z}_p.
\end{aligned}
\tag{2.46}
$$

The proof $\pi = (A, B, C)$ is computed by using the following three multi-scalar multiplications,

$$\begin{aligned}
A &= \left(1, 0, r, U_0, \cdots, U_{n-1}, \underbrace{0, \cdots, 0}_{m+n \text{ 0s}}\right) \cdot \sigma_1 \\
&= (\alpha + r\delta + U(x))\, G, \\
B &= (1, 0, s, V_0, V_1, \cdots, V_{n-1}) \cdot \sigma_2 \\
&= (\beta + s\delta + V(x))\, H, \\
C &= \left(s, r, rs, M_0, \cdots, M_{n-1}, \underbrace{0, \cdots, 0}_{\ell+1 \text{ 0s}}, a_{\ell+1}, \cdots, a_m, h_0, \cdots, h_{n-2}\right) \cdot \sigma_1 \\
&= \left(\sum_{i=\ell+1}^{m} a_i \cdot \frac{\beta u_i(x) + \alpha v_i(x) + w_i(x)}{\delta} + \right. \\
&\quad \left. \frac{h(x)t(x)}{\delta} + s(\alpha + U(x)) + r(\beta + V(x)) + rs\delta\right) G.
\end{aligned} \tag{2.47}$$

- $0/1 \leftarrow \mathsf{Verify}(R, \sigma, a, \pi)$: It first parses $\pi = (A, B, C) \in \mathbb{G}_1 \times \mathbb{G}_2 \times \mathbb{G}_1$, and then accepts the proof if and only if

$$e(A, B) = e(\alpha G, \beta H) \cdot e\left(\left(\sum_{i=0}^{\ell} a_i \cdot \frac{\beta u_i(x) + \alpha v_i(x) + w_i(x)}{\gamma}\right) G, \gamma H\right) \cdot e(C, \delta H). \tag{2.48}$$

zkSNARK schemes enable a prover to prove the validity of a statement to a verifier while keeping any other information concealed. They are supposed to have three fundamental properties, i.e., *completeness*, which guarantees that given a valid statement and the corresponding witness, the prover can convince the verifier. *Soundness*, which ensures that a malicious prover cannot convince the verifier of a false statement with non-negligible probability. Lastly, *zero knowledge*, which means that the proof does not reveal anything except the truth of the statement, and specifically, it does not reveal the witness.

The Groth16 scheme given by the above $(\mathsf{Setup}, \mathsf{Prove}, \mathsf{Verify})$ algorithms is perfect complete and perfect zero-knowledge. It has statistical knowledge soundness against adversaries that only use a polynomial number of generic group operations [Gro16, Theorem 2].

In Groth16, the proof generation involves an $(n+2)$-scalar multiplication in $\mathbb{G}_1$, an $(n+2)$-scalar multiplication in $\mathbb{G}_2$, and another $(2n+m-\ell+2)$-scalar multiplication in $\mathbb{G}_1$. The proof verification mainly requires an $(\ell+1)$-scalar multiplication and 3 bilinear pairings assuming that $e(\alpha G, \beta H)$ is precomputed.

2.4 BLS12-381 curve

This section introduces BLS12-381 curve [BLS02, Bow17]. The proposed multi-scalar multiplication methods will be instantiated and tested over BLS12-381 curve.

BLS12-381 curve is a pairing-friendly elliptic curve initially designed by Sean Bowe for the cryptocurrency system Zcash [Zca, Bow17]. It is widely deployed in blockchain applications such as Zcash, Ethereum, Chia, DFINITY and Algorand.

BLS12-381 curve provides 127-bit security on the elliptic curve groups side, estimated by Pollard's rho algorithm [Pol78]. It achieves around 110-bit security on the finite field side, estimated theoretically and conservatively by the exTNFS algorithm [BGK15, KB16, BD19], while a recent non-conservative research estimates its finite field side security to be 126-bit [GMT20].

2.4.1 Parameters

BLS12-381 curve is defined by a low Hamming weight parameter u given in hexadecimal form as

$$u = \texttt{-0xd201000000010000}. \tag{2.49}$$

Its Weierstrass equation over the prime field $\mathbb{F}_p$ is

$$E(\mathbb{F}_p) : y^2 = x^3 + 4,$$

where p is the 381-bit field characteristic,

$$\begin{aligned} p &= \frac{1}{3} \cdot (u^6 - 2u^5 + 2u^3 + u + 1) \\ &= \texttt{0x1a0111ea397fe69a4b1ba7b6434bacd764774b84f38512b} \\ &\quad \texttt{f6730d2a0f6b0f6241eabfffeb153ffffb9feffffffffaaab}. \end{aligned} \tag{2.50}$$

This curve contains a subgroup with a 255-bit prime order, denoted as r,

$$\begin{aligned} r &= u^4 - u^2 + 1 \\ &= \texttt{0x73eda753299d7d483339d80809a1d80} \\ &\quad \texttt{553bda402fffe5bfeffffffff00000001}. \end{aligned} \tag{2.51}$$

2.4.2 Groups $\mathbb{G}_1$ and $\mathbb{G}_2$

In order to define bilinear pairings, two distinct groups $\mathbb{G}_1$ and $\mathbb{G}_2$ of order r are needed. However the curve $E(\mathbb{F}_p)$ has only one subgroup of order r. If we extend $\mathbb{F}_p$ to $\mathbb{F}_{p^k}$ for some integer k, $E(\mathbb{F}_{p^k})$ would eventually contain some other subgroup of order r over which we can define the pairing. The minimal integer k that makes this happen is called the *embedding degree* of the curve. BLS12-381 curve is a curve in BLS12 pairing-friendly curve family, whose embedding degree k equals 12. Embedding degree k is also the minimum integer that makes r divide $(p^k - 1)$.

Let

$$\pi_p : E(\overline{\mathbb{F}_p}) \to E(\overline{\mathbb{F}_p}),\ (x, y) \mapsto (x^p, y^p)$$

be the Frobenius map, where $\overline{\mathbb{F}_p} = \cup_{i=1}^{\infty} \mathbb{F}_{p^i}$ is the algebraic closure of $\mathbb{F}_p$, then we know

$$\mathbb{G}_1 := E(\mathbb{F}_p)[r] = \{P \in E(\mathbb{F}_p) \mid r \cdot P = \infty\} \tag{2.52}$$

is the r-torsion subgroup of $E(\mathbb{F}_p)$, and

$$\begin{aligned} \mathbb{G}_2 :=& E(\mathbb{F}_{p^{12}})[r] \cap \ker(\pi_p - [p]) \\ =& \{P \in E(\mathbb{F}_{p^{12}}) \mid r \cdot P = \infty,\ \pi_p(P) = p \cdot P\} \end{aligned} \tag{2.53}$$

is the r-torsion subgroup of $E(\mathbb{F}_{p^{12}})$ different from $\mathbb{G}_1$.

If we directly compute pairings over $\mathbb{G}_2$, some of the operations would be evaluated in $\mathbb{F}_{p^{12}}$, which is inefficient. Instead, we utilize the following degree-2 extension field $\mathbb{F}_{p^2}$,

$$\mathbb{F}_{p^2} = \mathbb{F}_p[\mu]/(\mu^2 - (-1)). \tag{2.54}$$

Let

$$E'(\mathbb{F}_{p^2}) : y^2 = x^3 + 4(\mu + 1),\ \mu \in \mathbb{F}_{p^2} \tag{2.55}$$

be the degree-6 twist of $E(\mathbb{F}_p) : y^2 = x^3 + 4$, and define

$$\begin{aligned} \mathbb{G}_2' :=& E'(\mathbb{F}_{p^2})[r] \cap \ker(\pi_p - [p]) \\ =& \{P \in E'(\mathbb{F}_{p^2}) \mid r \cdot P = \infty,\ \pi_p(P) = p \cdot P\}, \end{aligned} \tag{2.56}$$

then $\mathbb{G}_2'$ can be used to replace $\mathbb{G}_2$ when computing pairings so that all operations are computed in $\mathbb{F}_{p^2}$. If needed, the untwisted map

$$\psi : \mathbb{G}_2' \to \mathbb{G}_2,\ (x', y') \mapsto \left(x'\omega^{-2}, y'\omega^{-3}\right),\ \text{where } \omega^6 = \mu + 1 \tag{2.57}$$

can be used to pull a point in $\mathbb{G}_2'$ back to $\mathbb{G}_2$.

In this thesis, we will focus on the computation of multi-scalar multiplication in $\mathbb{G}_1$ and $\mathbb{G}_2'$. Because the computation is always carried out in $\mathbb{G}_2'$, we will refer to $\mathbb{G}_2'$ as $\mathbb{G}_2$ when conducting the experiment in Chapter 6.

2.4.3 Optimal ate pairing

Let $\mathbb{G}_1$ and $\mathbb{G}_2$ be the two groups defined in Equations (2.52)(2.53) and

$$\mathbb{G}_T := \mu_r$$

be the subgroup of $\mathbb{F}_{p^{12}}^*$ consisting of r-th roots of unity. The optimal ate pairing defined over BLS12-381 curve has a neat formula as follows [GF16] [EMJ17, Theorem 3.3],

$$\begin{aligned} e : \mathbb{G}_1 \times \mathbb{G}_2 &\to \mathbb{G}_T, \\ (P, Q) &\mapsto (f_{u,Q}(P))^{\frac{p^{12}-1}{r}}, \end{aligned} \tag{2.58}$$

where $f_{u,Q}$ is the function with respect to the divisor

$$\mathrm{Div}(f_{u,Q}) = u(Q) - ([u]Q) - (u-1)(\infty)$$

and satisfies

$$f_{i+j,Q} = f_{i,Q} \cdot f_{j,Q} \cdot \frac{\ell_{[i]Q,[j]Q}}{v_{[i+j]Q}} \tag{2.59}$$

up to a nonzero factor in $\mathbb{F}_p$. In Equation (2.59), $\ell_{[i]Q,[j]Q}$ is the line passing through $[i]Q$ and $[j]Q$, and $v_{[i+j]Q}$ is the line perpendicular to the x-axis passing through $[i+j]Q$. The line function $v_{[i+j]Q}$ evaluated at P is

$$v_{[i+j]Q}(P) = x_P - x_{[i+j]Q},$$

which will become 1 after raising to the exponent of $(p^{12}-1)/r$ [EMJ17, Lemma 3.3], so it can be ignored when computing the optimal ate pairing.

Algorithm 3 Optimal ate pairing for BLS12-381 curve

Input: $P \in G_1$, $Q \in G_2$, the curve parameter u in Equation (2.49).
Output: The optimal ate pairing $f = e(P, Q) \in \mathbb{F}_{p^{12}}$.

1: Let $|u| = \sum_{i=0}^{\ell-1} u_i 2^i, u_{\ell-1} = 1, u_i \in \{0, 1\}$
2: $T \leftarrow Q,\ f \leftarrow 1$
3: **for** $i = \ell - 2$ to 0 **do**
4: $\quad f \leftarrow f^2 \cdot \ell_{T,T}(P),\ T \leftarrow 2T$
5: $\quad$ **if** $u_i = 1$ **then**
6: $\quad\quad f \leftarrow f \cdot \ell_{T,Q}(P),\ T \leftarrow T + Q$
7: **if** $u < 0$ **then**
8: $\quad f = f^{-1}$
9: $f \leftarrow f^{(p^{12}-1)/r}$
10: **return** f

From Equations (2.58)(2.59), one can obtain Algorithm 3, which is also known as Miller's algorithm [Mil04]. Steps 2−6 are called Miller loop, Steps 7−8 are called adjustment steps, and Step 9 is called final exponentiation.

If the curve parameter u is negative, the result obtained after Miller loop is $f_{|u|,Q}(P)$. Adjustment steps are involved to correct the result since

$$f_{u,Q}(P) = (f_{|u|,Q}(P))^{-1},\ u < 0.$$

In order to utilize the exponent decomposition in Step 9, we in practice compute the cube of the optimal ate pairing rather than the original ate pairing value. This is a common practice already adopted in [AFCK+12, GF16, Sco17b].

Shared final exponentiation

Pairing-based schemes are usually required to compute several pairings in the verification algorithm. If there are m pairings to be computed and each pairing is computed separately, it would go through m Miller loops and evaluate m final exponentiations. An alternative method is going through m Miller loops, obtaining m intermediate results, multiplying them together, and then evaluating a single shared final exponentiation.

For example, computing the optimal ate pairing over BLS12-381 curve by running MIRACL core C++ library [Sco17a] on a 2.2 GHz single-thread CPU takes about 2.08 ms per pairing. During this computation, Miller loop takes about 0.83 ms, final exponentiation

takes about 1.25 ms, and the multiplication in $\mathbb{F}_{p^{12}}$ takes only 0.006 ms, which is insignificant compared to Miller loop and final exponentiation. If 3 pairings are computed (for instance, the verification algorithm of Groth16 in Section 2.3.2), the trivial method that computes every pairing separately takes about 6.24 ms, while the shared final exponentiation method takes about 3.74 ms, providing a 40% improvement.

Further improvement for pairing computation can be achieved by fully exploiting sparse multiplication and accumulating the line functions. Please refer to [EMJ17,Sco19] for more details on this topic.

Chapter 3

Techniques for Multi-Scalar Multiplication over Fixed Points

This chapter presents techniques for computing the multi-scalar multiplication over fixed points $S_{n,r}$ defined in Equation (2.9). We will propose a new subsum accumulation algorithm and two frameworks for computing $S_{n,r}$, and discuss the potential benefits of utilizing GLV endomorphism and affine coordinates.

3.1 A new subsum accumulation algorithm

When computing $S_{n,r}$ by Pippenger's bucket method, after sorting each point into the bucket with respect to its scalar and computing the intermediate subsums $\{S_i\}_{1\leq i\leq m}$, the remaining step is to invoke a subsum accumulation algorithm to compute

$$S = b_1S_1 + b_2S_2 + \cdots + b_mS_m, \tag{3.1}$$

where $1 \leq b_1 \leq b_2 \leq \cdots \leq b_m$. When set $\{b_i\}_{1\leq i\leq m}$ is not a sequence of consecutive integers, Algorithm 1 shows the limitation of handling such case with less efficiency. Bos-Coster method [DR94, Section 4] can be utilized to deal with this case, but it is a recursive algorithm and its complexity is not easy to analyze. Therefore, we propose a straightforward algorithm to tackle this case.

Let us define $b_0 = 0$, and introduce the following notation

$$d = \max_{1\leq i\leq m}\{b_i - b_{i-1}\}, \tag{3.2}$$

then S in Equation (3.1) can be computed by Algorithm 4.

Algorithm 4 Subsum accumulation algorithm II

Input: $b_1, b_2, \cdots, b_m,\ S_1, S_2, \cdots, S_m$.
Output: $S = b_1S_1 + b_2S_2 + \cdots + b_mS_m$.

1: Define a length-$(d+1)$ array $\texttt{tmp} = [\infty] \times (\texttt{d} + 1)$
2: **for** $\texttt{i} = \texttt{m}$ to 1 by -1 **do**
3: $\quad \texttt{tmp}[0] = \texttt{tmp}[0] + \texttt{S}_\texttt{i}$
4: $\quad \texttt{k} = \texttt{b}_\texttt{i} - \texttt{b}_{\texttt{i}-1}$
5: $\quad$ **if** $\texttt{k} >= 1$ **then**
6: $\quad\quad \texttt{tmp}[\texttt{k}] = \texttt{tmp}[\texttt{k}] + \texttt{tmp}[0]$
7: **return** $1 \cdot \texttt{tmp}[1] + 2 \cdot \texttt{tmp}[2] + \cdots + \texttt{d} \cdot \texttt{tmp}[\texttt{d}]$

If we denote $\delta_j = b_j - b_{j-1}$, then $b_i = \sum_{j=1}^{i} \delta_j$. The correctness of Algorithm 4 comes from the following equation,

$$\begin{aligned} \sum_{i=1}^{m} b_i S_i &= \sum_{i=1}^{m} \left(\sum_{j=1}^{i} \delta_j \right) S_i \\ &= \sum_{j=1}^{m} \delta_j \left(\sum_{i=j}^{m} S_i \right) \\ &= \sum_{k=1}^{d} k \cdot \sum_{1 \le j \le m,\ \delta_j = k} \left(\sum_{i=j}^{m} S_i \right). \end{aligned} \tag{3.3}$$

During the execution of Algorithm 4, the temporary variable $\texttt{tmp}[0]$ stores $\sum_{i=j}^{m} S_i$ when loop index i equals j, and the temporary variable $\texttt{tmp}[\texttt{k}]$ stores

$$\sum_{1 \le j \le m,\ \delta_j = k} \left(\sum_{i=j}^{m} S_i \right) \quad \text{for } 1 \le k \le d$$

after the **for** loop.

If $\{b_i\}_{1 \le i \le m}$ is strictly increasing and $\texttt{k}$ in Line 4 goes through $\{1, 2, \cdots, d\}$, then in the **for** loop (Lines 2 – 6), each iteration executes exactly 2 additions. Since all $d+1$ temporary variables in $\texttt{tmp}$ are initialized as ∞, there are $d+1$ additions with ∞, which have no computational cost. Therefore, the **for** loop executes $2m - (d+1)$ additions. Line

7 is computed by the subsum accumulation Algorithm 1 with $2(d-1)$ additions. In total, the cost of Algorithm 4 is $2m+d-3$ additions.

If $\{b_i\}_{1\leq i\leq m}$ is not strictly increasing, which means that sometimes `k` in Line 4 equals 0, the corresponding **for** iteration will only execute one addition by skipping the **if** part.

If `k` in Line 4 does not go through all integers in $\{1, 2, \cdots, d\}$, there exists a `tmp`[`k`] ($1 \leq$ `k` $\leq d$) that will skip the **for** loop and remain as ∞. In the **for** loop, the saved addition due to the initiation of `tmp`[`k`] as ∞ will no longer be saved. In the mean time, when Line 7 is executed, at least one addition will be saved because `tmp`[`k`] $= \infty$. As a result, the total cost will not increase.

In summary, the cost of Algorithm 4 in the worst case is

$$2m+d-3 \tag{3.4}$$

additions. When $d=1$, Algorithm 4 degenerates to Algorithm 1.

3.2 Frameworks for computing multi-scalar multiplication over fixed points

In this section, we will propose two frameworks for computing multi-scalar multiplication over fixed points. These frameworks extend BGMW method and Pippenger's bucket method, respectively.

3.2.1 Framework I

The first framework is inspired by BGMW method [BGMW92], which was originally designed for computing single-scalar multiplication using the notion of basic digit sets.

The goal is to compute

$$S_{n,r} = \sum_{i=1}^{n} a_i P_i,$$

where $0 \leq a_i < r$, and every P_i is fixed. Let M be a set of integers and B be a set of non-negative integers with $0 \in B$. If every a_i can be expressed as the following radix q representation (not necessarily uniquely)

$$a_i = \sum_{j=0}^{h-1} m_{ij} b_{ij} q^j,\ m_{ij} \in M, b_{ij} \in B, h = \lceil \log_q r \rceil, \tag{3.5}$$

then $S_{n,r}$ can be computed as follows,

$$\begin{aligned} S_{n,r} &= \sum_{i=0}^{n} a_i P_i \\ &= \sum_{i=1}^{n} \left(\sum_{j=0}^{h-1} m_{ij} b_{ij} q^j \right) P_i \\ &= \sum_{i=1}^{n} \sum_{j=0}^{h-1} b_{ij} \cdot m_{ij} q^j P_i. \end{aligned} \tag{3.6}$$

Denote $P_{ij} := m_{ij} q^j P_i$, then

$$\begin{aligned} S_{n,r} &= \sum_{i=1}^{n} \sum_{j=0}^{h-1} b_{ij} P_{ij} \\ &= \sum_{i=1}^{n} \sum_{j=0}^{h-1} \left(\sum_{k \in B} k \cdot \sum_{i,j \text{ s.t. } b_{ij}=k} P_{ij} \right) \\ &= \sum_{k \in B} k \cdot \left(\sum_{1 \le i \le n,\ 0 \le j \le h-1,\ b_{ij}=k} P_{ij} \right). \end{aligned} \tag{3.7}$$

Suppose these $nh|M|$ points

$$\{ m q^j P_i \mid 1 \le i \le n, 0 \le j \le h-1, m \in M \} \tag{3.8}$$

are precomputed, and define the intermediate subsum S_k as

$$S_k := \sum_{1 \le i \le n,\ 0 \le j \le h-1,\ b_{ij}=k} P_{ij}, \quad k \in B,$$

then Equation (3.7) can be evaluated by first computing all S_k ($k \in B$) using at most

$$nh - (|B| - 1)$$

additions. The reason for this is straightforward because there are nh points being sorted into $|B| - 1$ subsums. The remaining part is computed by Algorithm 4 using at most

$$2(|B| - 1) + d - 3$$

additions, where d is the maximum difference between two neighboring elements in B.

To sum up, the worst case time complexity for computing $S_{n,r}$ is

$$nh + |B| + d - 4 \tag{3.9}$$

additions, where $h = \lceil \log_q r \rceil$, with the help of

$$nh|M| \tag{3.10}$$

precomputed points.

The set M is called a *multiplier set*, because the set of precomputed points contains all points multiplied by each element from M. The set B is called a *bucket set*, since all points are sorted into subsum buckets with respect to the scalars in B. This framework is translated into Algorithm 5.

In Algorithm 5, if we denote the expected number of zero elements in the length-nh array `scalars` as f, and assume that all elements in the length-$|B|$ array `buckets` of Step 5 are non-infinity, then the average complexity for computing $S_{n,r}$ can be estimated as

$$nh + |B| + d - f \tag{3.11}$$

additions.

From Equations (3.9)(3.11) we can observe that, when n and r are fixed, we can reduce the time complexity for computing $S_{n,r}$ by choosing a larger radix q to make the length h smaller, or by finding a smaller bucket set B. These two alternatives are closely interconnected. We will discuss them in Chapter 4.

Example. Under Framework I, BGMW method presented in Section 2.2.4 has

$$M = \{1\},\ B = \{0, 1, 2, \cdots, 2^c - 1\},$$

or

$$M = \{-1, 1\},\ B = \{0, 1, 2, \cdots, 2^{c-1}\}.$$

3.2.2 Framework II

We propose the second framework, which is inspired by Pippenger's bucket method and Framework I. Framework II requires only $1/h$ of the precomputation size compared to

Algorithm 5 Multi-scalar multiplication over fixed points: Framework I

Input: Scalars $a_1, a_2, \cdots, a_n$, fixed points $P_1, P_2, \cdots, P_n$, radix q, scalar length h, multiplier set $M = \{m_0, m_1, \cdots, m_{|M|-1}\}$, bucket set $B = \{b_0, b_1, \cdots, b_{|B|-1}\}$.
Output: $S_{n,r} = \sum_{i=1}^{n} a_i P_i$.

1: Precompute a length-$nh|M|$ point array `precomputation`, such that

$$\texttt{precomputation}\,[|\texttt{M}|((\texttt{i}-1)\texttt{h}+\texttt{j})+\texttt{k}] = \texttt{m}_\texttt{k}\texttt{q}^\texttt{j}\texttt{P}_\texttt{i} \text{ for } 1 \le i \le n, 0 \le j < h, 0 \le k < |M|.$$

Precompute a hash table `mindex` to record the index of every multiplier, such that $\texttt{mindex}[\texttt{m}_\texttt{k}] = \texttt{k}$. Precompute a hash table `bindex` to record the index of every bucket, such that $\texttt{bindex}[\texttt{b}_\texttt{k}] = \texttt{k}$.

2: Convert every a_i to its standard q-ary form, then convert it to

$$\texttt{a}_\texttt{i} = \sum_{\texttt{j}=0}^{\texttt{h}-1} \texttt{m}_{\texttt{ij}}\texttt{b}_{\texttt{ij}}\texttt{q}^\texttt{j}, \texttt{m}_{\texttt{ij}} \in \texttt{M}, \texttt{b}_{\texttt{ij}} \in \texttt{B}.$$

3: Create a length-nh scalar array `scalars`, such that $\texttt{scalars}[(\texttt{i}-1)\texttt{h}+\texttt{j}] = \texttt{b}_{\texttt{ij}}$. Create a length-$nh$ array `points` recording the index of points, such that $\texttt{points}[(\texttt{i}-1)\texttt{h}+\texttt{j}] = |\texttt{M}|((\texttt{i}-1)\texttt{h}+\texttt{j}) + \texttt{mindex}[\texttt{m}_{\texttt{ij}}]$. Then n-scalar multiplication $S_{n,r}$ is equivalent to the following nh-scalar multiplication

$$\sum_{\texttt{i}=0}^{\texttt{nh}-1} \texttt{scalars}[\texttt{i}] \cdot \texttt{precomputation}\,[\texttt{points}[\texttt{i}]],$$

where all scalars in `scalars` are from the bucket set B.

4: Create a length-$|B|$ point array `buckets` to record the intermediate subsums, and initialize every point to ∞. For $0 \le i \le nh-1$, add point $\texttt{precomputation}\,[\texttt{points}[\texttt{i}]]$ to bucket $\texttt{buckets}\,[\texttt{bindex}[\texttt{scalars}[\texttt{i}]]]$.

5: Invoke Algorithm 4 to compute $\sum_{\texttt{i}=0}^{|\texttt{B}|-1} \texttt{b}_\texttt{i} \cdot \texttt{buckets}[\texttt{i}]$, then return the result.

Framework I, making it a viable alternative when the devices' memory size is relatively restricted.

Let $q = 2^c$ be a radix, M an integer set and B a non-negative integer set with $0 \in B$,

such that each scalar a_i $(0 \leq a_i < r)$ appeared in $S_{n,r}$ can be expressed as

$$a_i = \sum_{j=0}^{h-1} m_{ij} b_{ij} q^j, \ m_{ij} \in M, b_{ij} \in B, h = \lceil \log_q r \rceil. \tag{3.12}$$

Using this decomposition, the n-scalar multiplication $S_{n,r}$ can be computed as follows,

$$\begin{aligned} S_{n,r} &= \sum_{i=1}^{n} a_i P_i \\ &= \sum_{i=1}^{n} \sum_{j=0}^{h-1} b_{ij} m_{ij} q^j P_i \\ &= \sum_{j=0}^{h-1} q^j \cdot \left(\sum_{i=1}^{n} b_{ij} \cdot m_{ij} P_i \right). \end{aligned}$$

Denote $P_{ij} := m_{ij} P_i$, and suppose all the following $|M|n$ points

$$\{ mP_i \mid 1 \leq i \leq n, m \in M \} \tag{3.13}$$

are precomputed, then

$$S_j := \sum_{i=1}^{n} b_{ij} \cdot m_{ij} P_i = \sum_{i=1}^{n} b_{ij} \cdot P_{ij}$$

is an n-scalar multiplication where all scalars are from B. By Equation (3.9) in Framework I (assigning $h = 1$), each S_j can be computed with

$$n + |B| + d - 4 \tag{3.14}$$

additions. The remaining part is

$$S_{n,r} = \sum_{j=0}^{h-1} q^j S_j,$$

which can be computed using

$$(h-1)(c+1)$$

additions by a method similar to Horner's rule, as shown in Equation (2.15).

To sum up, Framework II computes $S_{n,r}$ using at most

$$h(n + |B| + d - 4) + (h-1)(c+1) \tag{3.15}$$

additions. When we design bucket sets, it is ensured that d is small. Because

$$h = \lceil \log_q r \rceil = \left\lceil \frac{\log_2 r}{c} \right\rceil,$$

and $\lceil \log_2 r \rceil$ is several hundreds in practice[1], it follows that

$$(h-1)(c+1)$$

is in the range of hundreds. Thus the main term of the time complexity is

$$h(n + |B|). \tag{3.16}$$

Framework II is translated into Algorithm 6.

Example. Under Framework II, Pippenger's bucket method presented in Section 2.2.3 has

$$M = \{1\},\ B = \{0, 1, 2, \cdots, 2^c - 1\},$$

or

$$M = \{-1, 1\},\ B = \{0, 1, 2, \cdots, 2^{c-1}\}.$$

3.3 Other tricks

In this section, two tricks that can reduce the time complexity or the precomputation size are introduced. These tricks can be utilized by Pippenger bucket method, BGMW method and the proposed frameworks. We did not implement them in this thesis and further investigation is needed in order to assess their relative effectiveness and impact.

3.3.1 GLV endomorphism

Let us first discuss the potential benefits of utilizing GLV endomorphism [GLV01]. GLV endomorphism could be used to either shrink down the precomputation size under Framework I, or accelerate the computation of $S_{n,r}$ under Framework II, at the cost of computing some field multiplications on the fly.

[1]The parameter r is usually the order of the pairing-friendly curve groups over which the pairing is defined. In order to provide 128-bit security, r is at least 256-bit. In order to provide 192-bit security, r is at least 384-bit.

Algorithm 6 Multi-scalar multiplication over fixed points: Framework II

Input: Scalars $a_1, a_2, \cdots, a_n$, fixed points $P_1, P_2, \cdots, P_n$, radix q, scalar length h, multiplier set $M = \{m_0, m_1, \cdots, m_{|M|-1}\}$, bucket set $B = \{b_0, b_1, \cdots, b_{|B|-1}\}$.

Output: $S_{n,r} = \sum_{i=1}^{n} a_i P_i$.

1: Precompute a length-$n|M|$ point array `precomputation`, such that

$$\texttt{precomputation}\,[|\texttt{M}|(\texttt{i}-1)+\texttt{k}] = \texttt{m}_\texttt{k}\texttt{P}_\texttt{i} \text{ for } 1 \le i \le n, 0 \le k < |M|.$$

Precompute a hash table `mindex` to record the index of every multiplier, such that $\texttt{mindex}[\texttt{m}_\texttt{k}] = \texttt{k}$. Precompute a hash table `bindex` to record the index of every bucket, such that $\texttt{bindex}[\texttt{b}_\texttt{k}] = \texttt{k}$.

2: Convert every a_i to its standard q-ary form, then convert it to

$$\texttt{a}_\texttt{i} = \sum_{\texttt{j}=0}^{\texttt{h}-1} \texttt{m}_{\texttt{ij}}\texttt{b}_{\texttt{ij}}\texttt{q}^\texttt{j}, \texttt{m}_{\texttt{ij}} \in \texttt{M}, \texttt{b}_{\texttt{ij}} \in \texttt{B}.$$

3: Create a length-h point array `subsums` to record the intermediate subsums. For j from 0 to $h-1$, invoke Algorithm 5 with scalar length 1 to compute the following n-scalar multiplication,

$$\texttt{subsums}[\texttt{j}] = \sum_{\texttt{i}=1}^{\texttt{n}} \texttt{b}_{\texttt{ij}} \cdot \texttt{precomputation}[|\texttt{M}|(\texttt{i}-1)+\texttt{mindex}[\texttt{m}_{\texttt{ij}}]],$$

where all scalars are from the bucket set B.

4: Use the method in Equation (2.15) to compute

$$\sum_{\texttt{j}=0}^{\texttt{h}-1} \texttt{q}^\texttt{j} \cdot \texttt{subsums}[\texttt{j}],$$

then return the result.

GLV endomorphism provides a shortcut to obtain a specific scalar multiplication for every point in an elliptic curve group, at the expense of a field multiplication. Suppose GLV endomorphism is applicable in the elliptic curve group E of prime order r when computing $S_{n,r}$. For a point $P = (x, y)$ in E, let

$$\phi(P) = \lambda P = (\beta x, y)$$

be the GLV endomorphism, where $\lambda^2 + \lambda + 1 = 0 \bmod r$, $\beta^2 + \beta + 1 = 0 \bmod p$, and p is the characteristic of the field over which E is defined.

Precomputation size reduction for Framework I

Utilizing GLV endomorphism, it is possible to reduce the precomputation size of Framework I by approximately a factor of 2, at the cost of computing some field multiplications on the fly.

When computing $S_{n,r}$, we can replace every single-scalar multiplication by a 2-scalar multiplication. Suppose in the optimal situation each scalar can be decomposed to [bab86, GLV01]

$$a_i \equiv a_{i,0} + \lambda a_{i,1} \bmod r,$$

where $|a_{i,0}|$, $|a_{i,1}|$ are approximately no more than $\sqrt{r}$[2], then we have

$$\begin{aligned} S_{n,r} &= \sum_{i=1}^{n} a_i P_i \\ &= \sum_{i=1}^{n} (a_{i,0} + a_{i,1}\lambda) P_i \\ &= \sum_{i=1}^{n} (a_{i,0} P_i + a_{i,1} \cdot \lambda P_i) \\ &= \sum_{i=1}^{n} (a_{i,0} P_i + a_{i,1} \cdot \phi(P_i)). \end{aligned} \tag{3.17}$$

This is equivalent to a $2n$-scalar multiplication where all scalars are approximately no more than $\sqrt{r}$. By Equation (3.9) in Framework I, this $2n$-scalar multiplication costs at most approximately

$$2n \cdot h' + |B| \tag{3.18}$$

additions, where

$$h' \approx \left\lceil \log_q \sqrt{r} \right\rceil = \left\lceil \frac{\log_q r}{2} \right\rceil = \left\lceil \frac{h}{2} \right\rceil.$$

[2]Not all curves satisfy this condition, but BN curves and BLS12 curves do.

During the computation process, every $a_{i,1}$ in Equation (3.17) is further decomposed into the following radix q expression of length h',

$$a_{i,1} = \sum_{j=0}^{h'-1} b_{ij,1} \cdot m_{ij,1} \cdot q^j,$$

and these $m_{ij,1}q^j\phi(P_i)$ appeared in this process are computed on the fly by GLV endomorphism using nh' field multiplications. The following $|M|nh'$ points are precomputed,

$$\{mq^j P_i \mid m \in M, 1 \leq i \leq n, 0 \leq j \leq h'\}, \tag{3.19}$$

which are approximately half of the points in the precomputation set of Framework I, as shown in Equation (3.8).

Acceleration for Framework II

By Equation (3.17), $S_{n,r}$ is equivalent to a $2n$-scalar multiplication whose all scalars are approximately no more than $\sqrt{r}$. According to Equation (3.16) in Framework II, $S_{n,r}$ can be computed by

$$h'(2n + |B|) \tag{3.20}$$

additions, where

$$h' \approx \lceil \log_q(\sqrt{r}) \rceil = \left\lceil \frac{h}{2} \right\rceil. \tag{3.21}$$

During the execution of Framework II, every $a_{i,1}$ in Equation (3.17) is further decomposed into the following radix q expression of length h',

$$a_{i,1} = \sum_{j=0}^{h'-1} b_{ij,1} \cdot m_{ij,1} \cdot q^j.$$

Therefore, it also requires nh' field multiplications to compute the following points

$$\{\phi(m_{ij,1}P_i) \mid 1 \leq i \leq n, 0 \leq j \leq h' - 1\}$$

on the fly, where these $m_{ij,1}P_i$ are directly retrieved from the precomputed points.

3.3.2 Affine coordinates

It is very popular to adopt projective and Jacobian coordinates for computing multi-scalar multiplication. These coordinate systems offer advantages over affine coordinates by avoiding the need for inversions in the finite field, which are necessary for elliptic curve addition in affine coordinates. An inversion may be orders of magnitude expensive than a multiplication, sometimes estimated to be 100 times more expensive. For instance, Jacobian mixed addition requires 11 multiplications in the base field, as shown in Equation (6.7), while affine addition requires 3 multiplications and 1 inversion in the base field. Therefore, Jacobian mixed addition is usually faster than affine addition.

However, Gabizon and Williamson observed that addition in affine coordinates may use fewer field multiplications compared to addition in projective and Jacobian coordinates when implementing Pippenger's bucket method for TurboPLONK [GJW20]. If the affine points to be added are properly arranged in a way that a batch of ℓ additions can be computed concurrently, then Montgomery batch inversion technique can be utilized to compute the slopes simultaneously.

Montgomery batch inversion trick computes the inverses of ℓ elements $\lambda_1, \lambda_2, \cdots, \lambda_\ell$ in a field using the following steps,

- Set $\Lambda_1 = \lambda_1$. For i from 1 to $\ell - 1$, compute $\Lambda_{i+1} = \Lambda_i \lambda_i$. This step takes $(\ell - 1)$ field multiplications. We have
$$\Lambda_i = \prod_{j=1}^{i} \lambda_j, \ 1 \leq i \leq \ell. \tag{3.22}$$
- Compute $R_\ell = \Lambda_\ell^{-1}$ by 1 field inversion.
- For i from $\ell - 1$ to 1, compute $R_i = R_{i+1} \cdot \lambda_{i+1}$. This step requires $(\ell - 1)$ field multiplications. We have
$$R_i = \prod_{j=1}^{i} \lambda_j^{-1}, \ 1 \leq i \leq \ell. \tag{3.23}$$
- By Equations (3.22)(3.23), it is clear that $\lambda_1^{-1} = R_1$, and that for i from 2 to ℓ $\lambda_i^{-1} = R_i \Lambda_{i-1}$. This step takes $(\ell - 1)$ field multiplications.

The above method computes ℓ inverses by using $3(\ell-1)$ field multiplications and 1 field inversion. If ℓ is sufficiently large, then each inverse can be computed by approximately 3 field multiplications on average. This trick yields a batching affine addition algorithm that requires only 6 field multiplications.

Chapter 4

Constructions of Multiplier Set and Bucket Set

In this chapter, we construct five pairs of multiplier set and bucket set (M, B) that can be utilized to compute $S_{n,r}$ within the frameworks presented in Chapter 3. The essential challenge is to ensure that each scalar in $S_{n,r}$ is converted to its radix q representation where every digit is the product of an element from M and an element from B, while keeping the size of B as small as possible.

We would like to require M to be symmetric, meaning

$$M = \{i \mid i \in M'\} \cup \{-i \mid i \in M'\},$$

where the set M' only contains positive integers. In this case, the precomputation size could be halved by computing the inverse of a point on the fly when needed. For a point $P = (x, y)$ in an elliptic curve group, its inverse $-P$ is obtained by taking the negative of its y coordinate at almost no cost.

Given a pair of sets (M, B) and an arbitrary scalar a $(0 \leq a < r)$ in its standard q-ary representation

$$a = \sum_{j=0}^{h-1} a_j q^j,\ 0 \leq a_j < q, h = \lceil \log_q r \rceil, \tag{4.1}$$

the sets (M, B) are said to be *valid*, if they enable the scalar conversion from its standard q-ary representation to the following radix q representation (which may not be unique)

$$a = \sum_{j=0}^{h-1} m_j b_j q^j,\ m_j \in M, b_j \in B, h = \lceil \log_q r \rceil. \tag{4.2}$$

We will show the proposed constructions are valid, thus yielding effective algorithms for computing $S_{n,r}$ when combining with the proposed frameworks.

4.1 Construction I

For a radix $q = 2^c$ $(10 \leq c \leq 31)$, the multiplier set is picked as

$$M = \{-3, -2, -1, 1, 2, 3\}. \tag{4.3}$$

In order to determine the bucket set B, let us first define three auxiliary sets B_0, B_1 and B_2. Let $h = \lceil \log_q r \rceil$, and

$$r_{h-1} = \left\lfloor \frac{r}{q^{h-1}} \right\rfloor \tag{4.4}$$

be the leading term of r in its standard q-ary expression,

$$\begin{aligned} B_0 =& \{0\} \cup \left\{ i \mid 1 \leq i \leq \frac{q}{2},\ s.t.\ \omega_2(i) + \omega_3(i) \equiv 0 \bmod 2 \right\}, \\ B_2 =& \{0\} \cup \{ i \mid 1 \leq i \leq r_{h-1} + 1,\ s.t.\ \omega_2(i) + \omega_3(i) \equiv 0 \bmod 2 \}, \end{aligned} \tag{4.5}$$

where $\omega_2(i)$ represents the exponent of the factor 2 in i, and $\omega_3(i)$ represents the exponent of the factor 3 in i. For instance, if $i = 2^e k,\ 2 \nmid k$, then $\omega_2(i) = e$. From these definitions, B_0 (respectively, B_2) has such a property that for each t $(0 \leq t \leq q/2)$ (respectively, $0 \leq t \leq r_{h-1} + 1$), there exist an element $b \in B_0$ (respectively, $b \in B_2$) and an integer $m \in \{1, 2, 3\}$, such that

$$t = mb.$$

The set B_0 itself is a valid construction, which is mentioned in [BGMW92] in the exponentiation operation of general multiplicative groups. Since we can utilize negative elements in the multiplier set M, there are some redundant elements to be removed from B_0. Set B_1 is defined by Algorithm 7.

Property 1 holds for B_1.

Property 1. *Given $q = 2^c$ $(10 \leq c \leq 31)$, for each t $(0 \leq t \leq q)$, there exist an element $b \in B_1$ and an integer $m \in \{1, 2, 3\}$, such that*

$$t = mb \text{ or } t = q - mb.$$

The size of B_1 satisfies

$$0.208q < |B_1| < 0.211q.$$

Algorithm 7 Construction of auxiliary set B_1

Input: $B_0,\ q$.
Output: B_1.

```
1: B_1 = B_0
2: for i = q/4 to q/2 − 1 by 1 do
3:     if i is in B_0 and q − 2 · i is in B_0 then
4:         B_1.remove(q − 2 · i)
5: for i = ⌊q/6⌋ to q/4 − 1 by 1 do
6:     if i is in B_0 and q − 3 · i is in B_0 then
7:         B_1.remove(q − 3 · i)
8: return B_1
```

Property 1 is checked by computation using Algorithm 9. It is also asserted by computation that exchanging two *for* loops in Algorithm 7 would construct the same B_1.

Finally the bucket set B is constructed as

$$B = B_1 \cup B_2. \tag{4.6}$$

Example. For $r = 131101, q = 2^5 = 32$, we have $h = 4, r_{h-1} = 4$ and

$$B_0 = \{0, 1, 4, 5, 6, 7, 9, 11, 13, 16\}.$$

The redundant elements are 6 and 11, because

$$6 = q - 2 \times 13,\ \ 11 = q - 3 \times 7.$$

It follows that

$$B_1 = \{0, 1, 4, 5, 7, 9, 13, 16\}.$$

We also have

$$B_2 = \{0, 1, 4, 5\},$$

so the bucket set

$$B = B_1 \cup B_2 = \{0, 1, 4, 5, 7, 9, 13, 16\}.$$

Property 2. *For the multiplier set M and the bucket set B defined in Equations (4.3) (4.6), a scalar a $(0 \le a < r)$ can be expressed (not necessarily uniquely) as a radix q representation defined in Equation (4.2).*

Proof. By Property 1, we know that for any integer $t \in [0, q]$, it can be expressed as

$$t = mb + \alpha q,\ m \in M, b \in B, \alpha \in \{0, 1\},$$

and by the definition of B_2 we know that for any integer $t \in [0, r_{h-1}+1]$, it can be expressed as

$$t = mb,\ m \in \{1, 2, 3\},\ b \in B.$$

Back to Property 2. Algorithm 8 can be used to convert a from its standard q-ary representation defined in Equation (4.1) to its radix q representation defined in Equation (4.2).

Algorithm 8 Scalar conversion II

Input: $\{a_j\}_{0 \le j \le h-1},\ 0 \le a_j < q$ such that $a = \sum_{j=0}^{h-1} a_j q^j$.

Output: $\{(m_j, b_j)\}_{0 \le j \le h-1}, m_j \in M, b_j \in B$ such that $a = \sum_{j=0}^{h-1} m_j b_j q^j$.

1: **for** $\mathtt{j} = 0$ to $\mathtt{h} - 2$ by 1 **do**
2: Obtain $\mathtt{m_j}, \mathtt{b_j}, \alpha_\mathtt{j}$ such that $\mathtt{a_j} = \mathtt{m_j b_j} + \alpha_\mathtt{j} \mathtt{q}$
3: $\mathtt{a_{j+1}} = \alpha_\mathtt{j} + \mathtt{a_{j+1}}$
4: Obtain $\mathtt{m_{h-1}}, \mathtt{b_{h-1}}$ such that $\mathtt{a_{h-1}} = \mathtt{m_{h-1} b_{h-1}}$
5: **return** $\{(\mathtt{m_j}, \mathtt{b_j})\}_{0 \le \mathtt{j} \le \mathtt{h-1}}$

The correctness of Algorithm 8 comes from the fact that

i) $a_0 \in [0, q-1]$,

ii) $\alpha_j + a_{j+1} \in [0, q]$, for $0 \le j \le h-3$,

iii) $\alpha_{h-2} + a_{h-1} \in [0, r_{h-1}+1]$.

□

A hash table H is precomputed to store the decomposition of every $t \in [0, q]$, specifically,

$$H(t) = (m, b, \alpha) \text{ s.t. } t = mb + \alpha q, m \in M, b \in B, \alpha \in \{0, 1\}. \tag{4.7}$$

Steps 2 and 4 in Algorithm 8 are executed by retrieving the corresponding decomposition from the hash table. For the proposed (M, B), the hash table H can be implemented using a length-$(q+1)$ array `decomposition`, as shown in Algorithm 9. The `decomposition` array is also utilized to verify Property 1 by checking whether there is any entry in `decomposition` whose last element is -1.

Algorithm 9 Construction of digit decomposition hash table

Input: M, B defined in Equations (4.3) (4.6).

Output: Length-$(q+1)$ array `decomposition`, a realization of hash table H.

```
Define a length-(q+1) array decomposition and initiate every entry to be [0, 0, −1].
for m ∈ {−1, −2, −3} do
    for b ∈ B do
        if m · b + q ≥ 0 then
            decomposition[m · b + q] = [m, b, 1]
for m ∈ {1, 2, 3} do
    for b ∈ B do
        if m · b ≤ q then
            decomposition[m · b] = [m, b, 0]
return decomposition
```

When r_{h-1}/q (approximately r/q^h) is small, the value $|B_2|/|B|$ is also small. In this case, $|B| = |B_1 \cup B_2| \approx |B_1|$. By Property 1, it is expected that

$$|B| \approx |B_1| \approx 0.21q. \tag{4.8}$$

It is checked that the maximum difference between two neighboring elements in B is no more than 6. Proposition 1 is obtained by combining Construction I with Frameworks I and II.

Proposition 1. *For integer r, radix $q = 2^c$ $(10 \le c \le 31)$, length $h = \lceil \log_q r \rceil$, the multiplier set*

$$M = \{-3, -2, -1, 1, 2, 3\}$$

and the bucket set B jointly decided by Equations (4.5)(4.6) and Algorithm 7 are valid. When r/q^h is small, we have the following two methods for computing $S_{n,r}$.

- *According to Equations (3.8)(3.9) in Framework I, $S_{n,r}$ is computed by using at most approximately*

  $$nh + 0.21q$$

  additions, with the help of the following $3nh$ precomputed points

  $$\{mq^j P_i \mid 1 \le i \le n,\ 0 \le j \le h-1,\ m \in \{1, 2, 3\}\}.$$

- *According to Equations (3.13)(3.16) in Framework II, $S_{n,r}$ is computed by using at most approximately*

$$h(n + 0.21q)$$

additions, with the help of the following $3n$ precomputed points

$$\{mP_i \mid 1 \leq i \leq n,\ m \in \{1, 2, 3\}\}.$$

4.2 Construction II

For a radix $q = 2^c$, the length of a q-ary expression h is assumed to satisfy the condition[1]

$$q^{h-1} < r \leq \frac{q}{2} \cdot q^{h-1}. \tag{4.9}$$

This condition ensures that the radix q representation in Equation (4.2) also has a length of h. The multiplier set is defined as

$$M = \{-2, -1, 1, 2\}, \tag{4.10}$$

and the corresponding bucket set is constructed as

$$B = \{0\} \cup \left\{ i \;\middle|\; \omega_2(i) \equiv 0 \bmod 2,\ 1 \leq i \leq \frac{q}{2} \right\}, \tag{4.11}$$

where $\omega_2(i)$ represents the exponent of the factor 2 in i.

Property 3. *For the multiplier set M and the bucket set B defined in Equations (4.10) (4.11), a scalar a $(0 \leq a < r)$ can be expressed (not necessarily uniquely) as a radix q representation defined in Equation (4.2).*

Proof. Let us first demonstrate that every integer t $(0 \leq t \leq q)$ can be decomposed to

$$t = mb + \alpha q,\ m \in M, b \in B, \alpha \in \{0, 1\}.$$

- If $0 \leq t \leq q/2$, from the construction of B there exists an element $b \in B$ such that $t = b$ or $t = 2b$. In this case $\alpha = 0$.

[1] If r is the order of the elliptic curve group, this condition can be removed by the same technique used in Algorithm 2.

- If $q/2 < t \leq q$, then $q - t$ lies in $[0, q/2]$, which reduces to the previous case. Thus there exists an element $b \in B$ such that $q - t = b$ or $q - t = 2b$, which means $t = (-1) \cdot b + q$ or $t = (-2) \cdot b + q$. In this case $\alpha = 1$.

Back to Property 3. Given a scalar a in its standard q-ary representation defined in Equation (4.1), one can complete the scalar conversion by utilizing Algorithm 8, whose correctness is ensured by the following fact,

i) $a_0 \in [0, q-1]$,

ii) $\alpha_j + a_{j+1} \in [0, q]$ for all $0 \leq j \leq h-3$,

iii) $\alpha_{h-2} + a_{h-1} \in [0, q/2]$.

We require that $r \leq q/2 \cdot q^{h-1}$ when we select the radix q, which would ensure that

$$a_{h-1} \leq \frac{q}{2} - 1.$$

Therefore, $a_{h-1} \leq q/2$ considering the possible carry bit α_{h-2}. □

For every $t \in [0, q]$, a hash table H can be precomputed to store its decomposition, i.e., $H(t) = (m, b, \alpha)$ such that $t = mb + \alpha q,\ m \in M, b \in B, \alpha \in \{0, 1\}$. When doing the scalar conversion, values $m,\ b$ and α can be retrieved from the hash table instead of being computed on the fly.

The following lemma is used to estimate the size of B.

Lemma 1. *Let ℓ, m be positive integers such that $2^m \leq \ell < 2^{m+1}$. Let the set*

$$A = \{i \mid \omega_2(i) \equiv 0 \bmod 2,\ 1 \leq i \leq \ell\},$$

then the size of A is evaluated by

$$|A| = \sum_{i=0}^{m} (-1)^i \left\lfloor \frac{\ell}{2^i} \right\rfloor. \tag{4.12}$$

Proof. Define

$$A_i = \{j \mid j \bmod 2^i = 0, 1 \leq j \leq \ell\},\ \text{for } i = 0, 1, 2, \cdots,$$

notice that $A_{i+1} \subset A_i$, we have

$$A = \sum_{i=0}^{\infty} (-1)^i A_i.$$

Therefore Equation (4.12) holds by the observation that $|A_i| = \lfloor \ell/2^i \rfloor$. □

For $q = 2^c$, the size of B is estimated by

$$\begin{aligned} |B| &= 1 + \sum_{i=1}^{\infty} (-1)^{i+1} \left\lfloor \frac{q}{2} \cdot \frac{1}{2^{i-1}} \right\rfloor \\ &= 1 + \sum_{i=1}^{c} (-1)^{i+1} \cdot \frac{q}{2^i} \\ &= \frac{q + (-1)^{c+1}}{3} + 1 \\ &< \frac{q}{3} + 2. \end{aligned} \tag{4.13}$$

The maximum difference between two neighboring elements in B is 2. Proposition 2 is obtained by combining Construction II with Frameworks I and II.

Proposition 2. *For integer r, radix $q = 2^c$, length $h = \lceil \log_q r \rceil$, such that*

$$q^{h-1} < r \leq \frac{q}{2} \cdot q^{h-1},$$

the multiplier set

$$M = \{-2, -1, 1, 2\}$$

and the bucket set

$$B = \{0\} \cup \left\{ \; i \; \middle| \; \omega_2(i) \equiv 0 \bmod 2, \; 1 \leq i \leq \frac{q}{2} \right\}$$

are valid. This construction of (M, B) yields the following two methods for computing $S_{n,r}$.

- *According to Framework I, $S_{n,r}$ can be computed by using at most approximately*

$$nh + \frac{q}{3} \tag{4.14}$$

  additions, with the help of $2nh$ precomputed points

$$\{mq^j P_i \mid 1 \leq i \leq n, \; 0 \leq j \leq h-1, \; m \in \{1, 2\}\}.$$

- *According to Framework II, $S_{n,r}$ can be computed by using at most approximately*

$$h\left(n + \frac{q}{3}\right) \tag{4.15}$$

  additions, with the help of $2n$ precomputed points

$$\{mP_i \mid 1 \leq i \leq n, \; m \in \{1, 2\}\}.$$

4.3 Construction III

For a radix $q = 2^c$ and an integer h such that

$$q^{h-1} < r \leq \frac{q}{4} \cdot q^{h-1},$$

we denote $\lambda = q \bmod 3$, then $\lambda \in \{1, 2\}$. The multiplier set is picked as

$$M = \{-3, -1, 1, 3\}, \tag{4.16}$$

the corresponding bucket set is constructed as

$$\begin{aligned} B =& \{0\} \cup \left\{ i \;\middle|\; \omega_3(i) \equiv 0 \bmod 2,\ 1 \leq i \leq \frac{q}{12} \right\} \\ & \cup \left\{ i \;\middle|\; \frac{q}{12} \leq i \leq \frac{q}{4} \right\} \cup \left\{ 3i - \lambda \;\middle|\; i \text{ s.t. } \frac{q}{4} \leq 3i - \lambda \leq \frac{q}{2} \right\}, \end{aligned} \tag{4.17}$$

where $\omega_3(i)$ represents the exponent of the factor 3 in i.

Property 4. *For the multiplier set M and the bucket set B defined in equations (4.16) (4.17), a scalar a $(0 \leq a < r)$ can be expressed (not necessarily uniquely) as a radix q representation defined in Equation (4.2).*

Proof. Let us first demonstrate that for an integer t $(0 \leq t \leq q)$, it can be decomposed to

$$t = mb + \alpha q,\ m \in M, b \in B, \alpha \in \{0, 1\}.$$

- If $0 \leq t \leq q/4$, then $t = 1 \cdot b$ or $t = 3 \cdot b$, where $b \in B$. In this case $\alpha = 0$.
- If $3q/4 \leq t \leq q$, we know $q - t \leq q/4$, thus $q - t = mb, m \in \{1, 3\}, b \in B$. It follows that
  $$t = (-m) \cdot b + q,$$
  and $-m \in M,\ b \in B,\ \alpha = 1$.
- If $q/4 < t < 3q/4$, we further discuss the following three cases.
  - If $t \pmod 3 = 0$, then there exists an integer b $(q/12 < b \leq q/4)$, such that $t = 3b$, where $3 \in M,\ b \in B$ and $\alpha = 0$.

– If $t \pmod 3 = \lambda$, then there exists an integer b $(q/12 < b \leq q/4)$, such that $q - t = 3b$. It follows that

$$t = (-3) \cdot b + q,$$

and $-3 \in M,\ b \in B,\ \alpha = 1$.

– If $t \pmod 3 = 3 - \lambda$, then

  * If $t \leq q/2$, then $t \in B$. We have $t = 1 \cdot b$, where $1 \in M,\ b \in B$ and $\alpha = 0$.
  * If $t > q/2$, then $q - t < q/2$ and

$$q - t \equiv 2\lambda \equiv 3 - \lambda \pmod 3.$$

This boils down to the former case. We have $q - t = 1 \cdot b,\ b \in B$, thus

$$t = (-1) \cdot b + q,$$

and $-1 \in M,\ b \in B,\ \alpha = 1$.

Back to Property 4. Suppose a scalar a is given in its standard q-ary representation defined in Equation (4.1), Algorithm 8 can be used to convert a to the expression defined in Equation (4.2). The correctness is ensured by the following fact,

i) $a_0 \in [0, q-1]$,

ii) $\alpha_j + a_{j+1} \in [0, q]$ for all $0 \leq j \leq h-3$,

iii) $\alpha_{h-2} + a_{h-1} \in [0, q/4]$.

For $j = h-1$, when we select the radix q, it is required that $r \leq q/4 \cdot q^{h-1}$, which ensures $a_{h-1} \leq 1/4 \cdot q - 1$, so $a_{h-1} \leq 1/4 \cdot q$ considering the possible carry bit α_{h-2}. □

For every $t \in [0, q]$, a hash table H can be precomputed to store its decomposition, i.e., $H(t) = (m, b, \alpha)$ such that $t = mb + \alpha q,\ m \in M, b \in B, \alpha \in \{0, 1\}$. When doing the scalar conversion, values $m,\ b$ and α can be retrieved from the hash table instead of being computed on the fly.

Let us estimate the size of B. Define

$$B_1 = \left\{ i \;\middle|\; \omega_3(i) \equiv 0 \bmod 2,\ 1 \leq i \leq \frac{q}{12} \right\},$$

and suppose $3^m \leq q/12 < 3^{m+1}$, the size of B_1 is evaluated by a method similar to Lemma 1 as follows,

$$|B_1| = \sum_{i=0}^{m} (-1)^i \left\lfloor \frac{q}{12} \cdot \frac{1}{3^i} \right\rfloor. \tag{4.18}$$

If $m = 2k$, we have

$$\begin{aligned}
|B_1| &= \sum_{i=0}^{2k} (-1)^i \left\lfloor \frac{q}{12} \cdot \frac{1}{3^i} \right\rfloor \\
&= \sum_{i=0}^{k} \left\lfloor \frac{q}{12} \cdot \frac{1}{3^{2i}} \right\rfloor - \sum_{i=1}^{k} \left\lfloor \frac{q}{12} \cdot \frac{1}{3^{2i-1}} \right\rfloor \\
&\leq \sum_{i=0}^{k} \frac{q}{12} \cdot \frac{1}{3^{2i}} - \sum_{i=1}^{k} \left(\frac{q}{12} \cdot \frac{1}{3^{2i-1}} - 1 \right) \\
&= \sum_{i=0}^{m} (-1)^i \frac{q}{12} \cdot \frac{1}{3^i} + k \\
&= \frac{q}{16} + (-1)^m \frac{q}{48 \cdot 3^m} + \frac{m}{2} \\
&< \frac{q}{16} + \frac{3}{4} + \frac{m}{2}.
\end{aligned} \tag{4.19}$$

If $m = 2k + 1$,

$$\begin{aligned}
|B_1| &= \left(\sum_{i=0}^{2k} (-1)^i \left\lfloor \frac{q}{12} \cdot \frac{1}{3^i} \right\rfloor \right) - \left\lfloor \frac{q}{12} \cdot \frac{1}{3^{2k+1}} \right\rfloor \\
&< \left(\frac{q}{16} + \frac{3}{4} + \frac{m-1}{2} \right) - 0 \\
&= \frac{q}{16} + \frac{1}{2} + \frac{m}{2}.
\end{aligned} \tag{4.20}$$

Notice the fact that a real interval $[\alpha, \beta]$ covers at most $\beta - \alpha + 1$ integers, then

$$\begin{aligned}
|B| &\leq 1 + \left(\frac{q}{16} + \frac{3}{4} + \frac{m}{2} \right) + \left(\frac{q}{4} - \frac{q}{12} + 1 \right) \\
&\quad + \left(\frac{1}{3} \left(\frac{q}{2} + \lambda \right) - \frac{1}{3} \left(\frac{q}{4} + \lambda \right) + 1 \right) \\
&= \frac{5q}{16} + \frac{15}{4} + \frac{m}{2}.
\end{aligned} \tag{4.21}$$

If c is small, then $15/4 + m/2$ is a small integer that is negligible. The maximum difference between two neighboring elements in B is 3. Proposition 3 is obtained by combining Construction III with Frameworks I and II.

Proposition 3. *For integer r, radix $q = 2^c$, length $h = \lceil \log_q r \rceil$, such that*

$$q^{h-1} < r \leq \frac{q}{4} \cdot q^{h-1},$$

the multiplier set

$$M = \{-3, -1, 1, 3\}$$

and the bucket set

$$\begin{aligned} B = &\{0\} \cup \left\{ i \;\middle|\; \omega_3(i) \equiv 0 \bmod 2,\ 1 \leq i \leq \frac{q}{12} \right\} \\ &\cup \left\{ i \;\middle|\; \frac{q}{12} \leq i \leq \frac{q}{4} \right\} \cup \left\{ 3i - \lambda \;\middle|\; i \text{ s.t. } \frac{q}{4} \leq 3i - \lambda \leq \frac{q}{2} \right\} \end{aligned}$$

are valid. This construction of (M, B) yields the following two methods for computing $S_{n,r}$.

- *According to Framework I, $S_{n,r}$ can be computed by using at most approximately*

$$nh + \frac{5q}{16} \tag{4.22}$$

  additions, with the help of the following $2nh$ precomputed points

$$\{mq^j P_i \mid 1 \leq i \leq n,\ 0 \leq j \leq h-1,\ m \in \{1, 3\}\}.$$

- *According to Framework II, $S_{n,r}$ can be computed by using at most approximately*

$$h\left(n + \frac{5q}{16}\right) \tag{4.23}$$

  additions, with the help of the following $2n$ precomputed points

$$\{mP_i \mid 1 \leq i \leq n,\ m \in \{1, 3\}\}.$$

4.4 Other constructions

Recall that a multiplier set M is selected to be

$$M = \{-m_\ell, -m_{\ell-1}, \cdots, -m_1, m_1, m_2, \cdots, m_\ell\},$$

where each m_i $(1 \leq i \leq \ell)$ is a positive integer. The symmetry of M allows us to halve the precomputation size. Given a radix q, and a multiplier set M of size 2ℓ, the ultimate goal is to construct a bucket set B such that

$$|B| \approx \frac{q}{2\ell}.$$

We have proposed three constructions by far, and now we are ready to study two more constructions that further shrink down the size of the bucket sets, approaching the theoretical limit. In Construction IV, the radix $q = 2^c - 1$, while in Construction V, the radix q can be selected from pseudo-Mersenne primes, i.e., $q = 2^c - e$, where e is a small integer. Construction V is theoretically optimal in terms of the size of the bucket set.

When the radix q is not a power of 2, the radix conversion algorithms presented in [CH06, AKT20] can be utilized. These algorithms use only integer additions, integer subtractions and bitwise operations to convert a scalar from its binary expression to the standard q-ary expression. However, such radix conversion is much slower than that when $q = 2^c$. The advantage in time complexity brought by the smaller bucket set constructions is largely offset by the radix conversion. Thus if a more efficient radix conversion algorithm does not emerge, these bucket set constructions are primarily of theoretical significance.

4.4.1 Construction IV

For a radix $q = 2^c - 1$, and an integer h such that

$$q^{h-1} < r \leq \frac{q}{3} \cdot q^{h-1},$$

which implies $h = \lceil \log_q r \rceil$, the multiplier set is picked as

$$M = \{-2, -1, 1, 2\}, \tag{4.24}$$

and the corresponding bucket set is designed as

$$B = \{0\} \cup \left\{ i \;\middle|\; 1 \leq i \leq \frac{q}{6} \text{ s.t. } \omega_2(i) \equiv 0 \bmod 2 \right\} \cup \left\{ i \;\middle|\; \frac{q}{6} \leq i \leq \frac{q}{3} \right\}. \tag{4.25}$$

Here, $\omega_2(i)$ represents the exponent of the factor 2 in i.

Property 5. *For the multiplier set M and the bucket set B defined in Equations (4.24) (4.25), a scalar a $(0 \leq a < r)$ can be expressed (not necessarily uniquely) as a radix q representation defined in Equation (4.2).*

Proof. Let us first show that for an arbitrary integer t $(0 \leq t \leq q)$, it can be decomposed to

$$t = mb + \alpha q,\ m \in M, b \in B, \alpha \in \{0, 1\}.$$

- If $t \leq q/3$,
  - If $t \leq q/6$, we can further discuss the following two cases. If $\omega_2(t)$ is even, then $t \in B$. If $\omega_2(t)$ is odd, then $t/2 \in B$.
  - If $q/6 < t \leq q/3$, then $t \in B$.

  Anyway, there exists an element $b \in B$, such that $t = 1b$ or $t = 2b$. In these cases, $\alpha = 0$.
- If $t \geq 2q/3$, we have $q - t \leq q/3$, which falls into the above situation. There exists an element $b \in B$, such that $q - t = mb$, where $m = 1$ or 2. It follows that

$$t = (-m)b + q,\ \text{where}\ -m \in M, b \in B, \alpha = 1.$$

- If $q/3 < t < 2q/3$,
  - If $t \pmod 2 = 0$, then there exists an integer b $(q/6 < b < q/3)$ such that $t = 2b$, where $2 \in M, b \in B, \alpha = 0$.
  - If $t \pmod 2 = 1$, recall that q is odd, there exists an integer b $(q/6 < b < q/3)$ such that $q - t = 2b$. It follows that

$$t = (-2) \cdot b + q,\ \text{where}\ -2 \in M, b \in B, \alpha = 1.$$

Back to Property 5. Algorithm 8 can be used to convert a from its standard q-ary expression defined in Equation (4.1) to the representation defined in Equation (4.2). The correctness is ensured by the following fact,

i) $a_0 \in [0, q-1]$,

ii) $\alpha_j + a_{j+1} \in [0, q]$ for all $0 \leq j \leq h-3$,

iii) $\alpha_{h-2} + a_{h-1} \in [0, q/3]$.

For $j = h-1$, we require that $r \leq q/3 \cdot q^{h-1}$ when we select the radix q, which would ensure that

$$a_{h-1} \leq \frac{q}{3} - 1.$$

This leads to $a_{h-1} \leq q/3$ considering the possible carry bit α_{h-2}. We thus have $a_{h-1} = mb$ for some $b \in B$ and $m \in M$.

□

Let us estimate the size of B. Define

$$B_1 = \left\{ i \;\middle|\; 1 \le i \le \frac{q}{6} \text{ s.t. } \omega_2(i) \equiv 0 \bmod 2 \right\},$$

and suppose m is the positive integer such that $2^m \le q/6 < 2^{m+1}$, let us first estimate the size of B_1 by Lemma 1. We use the fact that for a real number α,

$$\alpha - 1 < \lfloor \alpha \rfloor \le \alpha.$$

If $m = 2k$, we have

$$\begin{aligned}
|B_1| &= \sum_{i=0}^{2k} (-1)^i \left\lfloor \frac{q}{6} \cdot \frac{1}{2^i} \right\rfloor \\
&= \sum_{i=0}^{k} \left\lfloor \frac{q}{6} \cdot \frac{1}{2^{2i}} \right\rfloor - \sum_{i=1}^{k} \left\lfloor \frac{q}{6} \cdot \frac{1}{2^{2i-1}} \right\rfloor \\
&\le \sum_{i=0}^{k} \frac{q}{6} \cdot \frac{1}{2^{2i}} - \sum_{i=1}^{k} \left(\frac{q}{6} \cdot \frac{1}{2^{2i-1}} - 1 \right) \\
&= \frac{q}{6} \cdot \sum_{i=0}^{2k} (-1)^i \frac{1}{2^i} + k \\
&= \frac{q}{6} \cdot \frac{2}{3} + \frac{q}{6} \cdot (-1)^m \frac{1}{2^m} + k \\
&< \frac{q}{9} + 2 + \frac{m}{2}.
\end{aligned}$$

If $m = 2k + 1$, then

$$\begin{aligned}
|B_1| &= \sum_{i=0}^{2k+1} (-1)^i \left\lfloor \frac{q}{6} \cdot \frac{1}{2^i} \right\rfloor \\
&\le \sum_{i=0}^{2k} (-1)^i \left\lfloor \frac{q}{6} \cdot \frac{1}{2^i} \right\rfloor \\
&< \frac{q}{9} + 2 + \frac{m}{2}.
\end{aligned}$$

The size of B is thus bounded by

$$\begin{aligned} |B| &< 1 + \left(\frac{q}{9} + 2 + \frac{m}{2}\right) + \left(\frac{q}{3} - \frac{q}{6} + 1\right) \\ &= \frac{5q}{18} + \frac{m}{2} + 4, \text{ where } m = \left\lfloor \log_2 \left(\frac{q}{6}\right) \right\rfloor \\ &= \frac{5q}{18} + \frac{\lfloor \log_2 q - \log_2 6 \rfloor}{2} + 4 \\ &< \frac{5q}{18} + \frac{c}{2} + 3. \end{aligned} \tag{4.26}$$

The maximum difference between two neighboring elements in B is 2. This construction of (M, B) can also yield two methods for computing $S_{n,r}$ when combining with the proposed frameworks. These methods are mainly of theoretical interest because the corresponding radix conversions are time consuming.

4.4.2 Construction V

Let q be a prime such that 2 is a primitive element in the finite field $\mathbb{F}_q$, and ℓ a small positive integer, h a small integer such that $2^\ell < q$ and $q^{h-1} < r \leq 2^{\ell-1} q^{h-1}$. The multiplier set is picked as

$$M = \{-2^i \mid 0 \leq i \leq \ell - 1\} \cup \{2^i \mid 0 \leq i \leq \ell - 1\}, \tag{4.27}$$

and the corresponding bucket set is constructed as

$$B = \left\{i \mid 0 \leq i \leq 2^\ell\right\} \cup \left\{ 2^{i\ell} \bmod q \;\middle|\; 0 \leq i \leq \left\lfloor \frac{q-1}{2\ell} \right\rfloor \right\}. \tag{4.28}$$

We first show that the following property holds for (M, B) defined above.

Property 6. *An integer t $(-2^{\ell-1} \leq t \leq q + 2^\ell - 1)$ can be expressed (not necessarily uniquely) as*

$$t = mb + \alpha q, \text{ where } m \in M, b \in B, -2^{\ell-1} + 1 \leq \alpha \leq 2^{\ell-1}. \tag{4.29}$$

Proof. The proof is divided into the following cases,

- If $t \in [-2^{\ell-1}, 2^\ell]$, $t = 1 \cdot |t|$ or $t = -1 \cdot |t|$. We have $m = \pm 1 \in M, b = |t| \in B, \alpha = 0$.

- If $t \in [2^{\ell-1}+1, q-1]$, notice that $2^{(q-1)/2} = -1 \bmod q$ because 2 is a primitive element in the finite field $\mathbb{F}_q$, then

$$\begin{aligned}\{t \mid 1 \le t \le q-1\} &= \{2^i \bmod q \mid 0 \le i \le q-2\} \\ &= \left\{2^i \bmod q \mid 0 \le i \le (q-1)/2\right\} \\ &\quad \cup \left\{-2^i \bmod q \mid 1 \le i \le (q-3)/2\right\}.\end{aligned}$$

  - If $t \in \{2^i \bmod q \mid 0 \le i \le (q-1)/2\}$, then

$$\begin{aligned}t &= 2^{i\ell+j} \bmod q \quad (0 \le i \le \lfloor (q-1)/(2\ell) \rfloor, 0 \le j \le \ell-1) \\ &= 2^j \cdot (2^{i\ell} \bmod q) \bmod q \\ &= mb \bmod q \\ &= mb + \alpha q,\end{aligned}$$

    where $-2^{\ell-1} + 1 \le \alpha \le 0$.
  - If $t \in \{-2^i \bmod q \mid 1 \le i \le (q-3)/2\}$, then

$$q - t \in \{2^i \bmod q \mid 0 \le i \le (q-1)/2\}.$$

    We have

$$q - t = mb + \alpha q, \text{ where } -2^{\ell-1} + 1 \le \alpha \le 0.$$

    One can obtain that

$$t = q - (mb + \alpha q) = (-m)b + (1-\alpha)q = m'b + \alpha' q,$$

    where $m' = -m \in M, b \in B$ and $\alpha' = 1 - \alpha \in [1, 2^{\ell-1}]$.

- If $t \in [q, q + 2^\ell - 1]$, then $t = mb + \alpha q$, where $m = 1, b = t - q \in B, \alpha = 1$.

□

Property 7. *For the multiplier set M and the bucket set B defined in Equations (4.27) (4.28), an arbitrary scalar a $(0 \le a < r)$ can be expressed (not necessarily uniquely) as a radix q representation defined in Equation (4.2).*

Proof. One can convert scalar a from its standard q-ary representation defined in Equation (4.1) to the radix q representation defined in Equation (4.2) by Algorithm 8. With Property 6, one can check that

i) $a_0 \in [0, q-1]$,

ii) $\alpha_j + a_{j+1} \in [1 - 2^{\ell-1}, q + 2^{\ell-1} - 1]$ for all $0 \leq j \leq h-3$,

iii) $\alpha_{h-2} + a_{h-1} \in [1 - 2^{\ell-1}, 2^{\ell} - 1]$.

These fact ensures the correctness of the scalar conversion. □

Similar to the previous constructions, for every $t \in [-2^{\ell-1}, q + 2^{\ell} - 1]$, a hash table H can be precomputed to store its decomposition, i.e., $H(t) = (m, b, \alpha)$ such that

$$t = mb + \alpha q,\ m \in M, b \in B, 1 - 2^{\ell-1} \leq \alpha \leq 2^{\ell-1}.$$

For a small positive integer ℓ, the size of B is estimated by

$$|B| \leq 2 + 2^{\ell} + \left\lfloor \frac{q-1}{2\ell} \right\rfloor \approx \frac{q}{2\ell}. \tag{4.30}$$

Let us demonstrate that this bucket set construction is asymptotically smallest. For the pair (M, B) to be considered valid, it is necessary for them to enable the conversion of the following q scalars

$$\{a \mid 0 \leq a < q\}$$

into the representations defined in Equation (4.2). Let us assume that

$$a = \sum_{j=0}^{h-1} m_j b_j q^j,\ m_j \in M, b_j \in B, h = \lceil \log_q r \rceil,$$

it follows that

$$a = m_0 b_0 \bmod q.$$

This implies

$$\{a \mid 0 \leq a < q\} \subseteq \{mb \bmod q \mid m \in M, b \in B\},$$

which requires

$$|M| \cdot |B| \geq q.$$

Therefore, if $|M| = 2\ell$, we must have

$$|B| \geq \frac{q}{2\ell}.$$

4.5 Comparison of different multiplier set and bucket set constructions

Among the five proposed constructions, the first three constructions provide different trade-offs between the precomputation size and the time complexity for computing $S_{n,r}$ when combining with Frameworks I and II in Chapter 3. The fourth and fifth constructions are of theoretical significance.

The estimation for the bucket set size in Construction I holds true when $q = 2^c$ ($10 \leq c \leq 31$) and r/q^h is small. The estimation for the bucket set size in Construction V holds true when ℓ is small.

Table 4.1: Different constructions of multiplier set and bucket set for computing $S_{n,r}$

Construction	Radix q	Multiplier set M	$\lvert B\rvert$
Pippenger and BGMW	$q = 2^c$	$\{-1, 1\}$	$\approx q/2$
Construction I	$q = 2^c$	$\{-3, -2, -1, 1, 2, 3\}$	$\approx 0.21q$
Construction II	$q = 2^c$	$\{-2, -1, 1, 2\}$	$\approx q/3$
Construction III	$q = 2^c$	$\{-3, -1, 1, 3\}$	$\approx 5q/16$
Construction IV	$q = 2^c - 1$	$\{-2, -1, 1, 2\}$	$\approx 5q/18$
Construction V	q is a prime s.t. 2 is primitive in $\mathbb{F}_q$	$\{-2^i \mid 0 \leq i \leq \ell - 1\} \cup \{2^i \mid 0 \leq i \leq \ell - 1\}$	$\approx q/(2\ell)$

Chapter 5

Instantiation

In this chapter, we will focus on the instantiation of Construction I over BLS12-381 curve [Bow17]. By combining Construction I with Frameworks I and II respectively, we can derive two methods for computing $S_{n,r}$. We will present the theoretical analysis for these two methods, and compare them against Pippenger's bucket method and BGMW method. We will also explain why we have chosen worst case time complexity as the representative for comparison.

The detailed parameters of BLS12-381 curve are presented in Section 2.4. When analyzing the time complexity for computing $S_{n,r}$, the most important parameter is the 255-bit group order r,

$$r = \texttt{0x73eda753299d7d483339d80809a1d80}\\ \texttt{553bda402fffe5bfeffffffff00000001}.$$

5.1 Bucket sets over BLS12-381 curve

Table 5.1 lists the bucket sets obtained by instantiating Construction I in Section 4.1 over BLS12-381 curve for radix $q = 2^c$ $(10 \leq c \leq 31)$. Here h is the length of a scalar in its standard q-ary expression, and $r_{h-1} = \lfloor r/q^{h-1} \rfloor$ is the leading term of r in its standard q-ary expression. Additionally, d is the maximum difference between two neighboring elements in B. From the table, we have $d \leq 6$ and

$$|B| \approx \begin{cases} 0.21q, & q = 2^c \ (10 \leq c \leq 31, c \neq 15, 16, 17), \\ 0.28q, & q = 2^{16}. \end{cases} \tag{5.1}$$

These radix values marked in gray are discarded. Radixes $2^{15}, 2^{17}$ are abandoned because $|B|/q$ is too large. For Framework I, the time complexity for computing $S_{n,r}$ is approximately $nh + |B|$, and for Framework II, the time complexity is approximately $h(n + |B|)$. When h is the same, a smaller $|B|$ results in lower time complexity. Radix 2^{21} is abandoned because while having the same h with radix 2^{20}, it has larger $|B|$. The similar reason applies to radixes 2^c for $c \in \{23, 25, 27, 28, 30, 31\}$.

Table 5.1: Bucket sets obtained by Construction I over BLS12-381 curve

Radix q	h	r_{h-1}	$\lvert B\rvert$	d	$\lvert B\rvert/q$
2^{10}	26	28	218	6	0.213
2^{11}	24	3	427	6	0.208
2^{12}	22	7	857	6	0.209
2^{13}	20	231	1725	6	0.211
2^{14}	19	7	3417	6	0.209
2^{15}	17	29677	17312	4	0.528
2^{16}	16	29677	18343	6	0.280
2^{17}	15	118710	69249	4	0.528
2^{18}	15	7	54618	6	0.208
2^{19}	14	231	109244	6	0.208
2^{20}	13	29677	220931	6	0.211
2^{21}	13	7	436906	6	0.208
2^{22}	12	7419	874437	6	0.208
2^{23}	12	3	1747625	6	0.208
2^{24}	11	29677	3497731	6	0.208
2^{25}	11	28	6990507	6	0.208
2^{26}	10	1899369	14139299	6	0.211
2^{27}	10	3709	27962333	6	0.208
2^{28}	10	7	55924059	6	0.208
2^{29}	9	7597479	112481229	6	0.210
2^{30}	9	29677	223698691	6	0.208
2^{31}	9	115	447392434	6	0.208

5.2 Method I

By combining Framework I in Section 3.2.1 with the first construction of multiplier set and bucket set pair presented in Section 4.1, we can derive the first instantiated method, which is summarized as the following Proposition 4.

Proposition 4. *Given the number of points n and the group order r over BLS12-381 curve, and suppose $q = 2^c$ $(10 \leq c \leq 31)$, $h = \lceil \log_q r \rceil$, then the multiplier set and bucket set defined in (4.3) (4.6) yield a method for computing $S_{n,r}$ using at most approximately*

$$\begin{cases} (nh + 0.21q) \text{ additions}, & q = 2^c \ (10 \leq c \leq 31, c \neq 15, 16, 17), \\ (nh + 0.28q) \text{ additions}, & q = 2^{16}, \end{cases} \tag{5.2}$$

with the help of the following $3nh$ precomputed points

$$\left\{ mq^j P_i \mid 1 \leq i \leq n,\ 0 \leq j \leq h-1,\ m \in \{1, 2, 3\} \right\}.$$

For a point $P = (x, y)$ on the elliptic curve E with short Weierstrass form, its inverse $-P = (x, -y)$ can be obtained for almost no cost. Therefore, the points associated with the negative elements in M are excluded from the precomputation table. Correspondingly, in Step 3 of Algorithm 5, a length-nh boolean array is added to record the sign of these multipliers. In Step 4, if a multiplier is negative, the negative of the corresponding point should be added to the intermediate subsum.

5.2.1 Theoretical analysis

A radix q is considered optimal if it minimizes the number of additions required to compute $S_{n,r}$ in the worst case. The optimal q and its corresponding scalar length h for different methods are summarized in Table 5.2. The precomputation size presented in this table is in terms of the points in $\mathbb{G}_1$ with affine coordinates. The precomputation size in $\mathbb{G}_2$ would double its counterpart in $\mathbb{G}_1$. Pippenger's bucket method and BGMW method are these two further optimized methods introduced in Section 2.2.3 and Section 2.2.4, respectively.

Table 5.2: Radix q, length h and precomputation size for computing $S_{n,r}$

	Pippenger			BGMW			Method I		
n	q	h	Precomput.	q	h	Precomput.	q	h	Precomput.
2^{10}	2^{8}	32	96.0 KB	2^{12}	22	2.06 MB	2^{13}	20	5.62 MB
2^{11}	2^{10}	26	192 KB	2^{13}	20	3.75 MB	2^{14}	19	10.6 MB
2^{12}	2^{10}	26	384 KB	2^{13}	20	7.50 MB	2^{14}	19	21.3 MB
2^{13}	2^{11}	24	768 KB	2^{15}	17	12.8 MB	2^{16}	16	36.0 MB
2^{14}	2^{12}	22	1.50 MB	2^{15}	17	25.5 MB	2^{16}	16	72.0 MB
2^{15}	2^{13}	20	3.00 MB	2^{16}	16	48.0 MB	2^{16}	16	144 MB
2^{16}	2^{13}	20	6.00 MB	2^{17}	15	90.0 MB	2^{19}	14	252 MB
2^{17}	2^{16}	16	12.0 MB	2^{17}	15	180 MB	2^{20}	13	468 MB
2^{18}	2^{16}	16	24.0 MB	2^{19}	14	336 MB	2^{20}	13	936 MB
2^{19}	2^{16}	16	48.0 MB	2^{20}	13	624 MB	2^{20}	13	1.83 GB
2^{20}	2^{16}	16	96.0 MB	2^{20}	13	1.22 GB	2^{22}	12	3.38 GB
2^{21}	2^{19}	14	192 MB	2^{22}	12	2.25 GB	2^{22}	12	6.75 GB

The number of additions taken to compute $S_{n,r}$ in the worst case and their comparison are summarized in Table 5.3, where

- Improv1 = (Pippenger - Method I)/Pippenger,
- Improv2 = (BGMW - Method I)/BGMW.

Table 5.3 shows that theoretically when computing $S_{n,r}$ over BLS12-381 curve for $n = 2^e$ $(10 \leq e \leq 21)$, Method I saves 21.05%–39.77% additions compared to Pippenger's bucket method, and it saves 2.08%–9.65% additions compared to BGMW method.

It should be noted that the proposed bucket sets listed in Section 5.1 are sufficient for computing $S_{n,r}$ over BLS12-381 for $n = 2^e$ $(22 \leq e \leq 29)$. Method I still shows 2.76%–5.81% theoretical improvement against BGMW method in these cases. However, a major drawback of Method I in these cases is that the precomputation size would become too large.

Table 5.3: Comparison of number of additions for computing $S_{n,r}$ in the worst case

n	Pippenger	BGMW	Method I	Improv1	Improv2
2^{10}	3.69×10^4	2.46×10^4	2.22×10^4	39.77%	9.65%
2^{11}	6.66×10^4	4.51×10^4	4.23×10^4	36.40%	6.05%
2^{12}	1.20×10^5	8.60×10^4	8.12×10^4	32.19%	5.55%
2^{13}	2.21×10^5	1.56×10^5	1.49×10^5	32.45%	4.00%
2^{14}	4.06×10^5	2.95×10^5	2.80×10^5	30.83%	4.89%
2^{15}	7.37×10^5	5.57×10^5	5.43×10^5	26.40%	2.59%
2^{16}	1.39×10^6	1.05×10^6	1.03×10^6	26.27%	2.08%
2^{17}	2.62×10^6	2.03×10^6	1.92×10^6	26.57%	5.25%
2^{18}	4.72×10^6	3.93×10^6	3.63×10^6	23.10%	7.71%
2^{19}	8.91×10^6	7.34×10^6	7.04×10^6	21.05%	4.13%
2^{20}	1.73×10^7	1.42×10^7	1.35×10^7	22.22%	4.93%
2^{21}	3.30×10^7	2.73×10^7	2.60×10^7	21.16%	4.48%

5.3 Method II

By combining Framework II in Section 3.2.2 with the first construction of multiplier set and bucket set pair in Section 4.1, we can obtain the second instantiated method, which is summarized as the following Proposition 5.

Proposition 5. *Given the number of points n and the group order r over BLS12-381 curve, and suppose $q = 2^c$ $(10 \leq c \leq 31)$, $h = \lceil \log_q r \rceil$, then the multiplier set and bucket set defined in (4.3) (4.6) yield a method for computing $S_{n,r}$ using at most approximately*

$$\begin{cases} h(n + 0.21q) \text{ additions}, & q = 2^c \ (10 \leq c \leq 31, c \neq 15, 16, 17), \\ h(n + 0.28q) \text{ additions}, & q = 2^{16}, \end{cases} \tag{5.3}$$

with the help of the following $3n$ precomputed points

$$\{mP_i \mid 1 \leq i \leq n, \ m \in \{1, 2, 3\}\}.$$

These points associated with the negative elements in M are excluded from the precomputation table, because they can be computed on the fly when needed.

5.3.1 Theoretical analysis

Method II utilizes less precomputation compared to BGMW method and Method I. As a result, it is expected to be slower than these two methods.

Table 5.4 summarizes the optimal q, the scalar length h, the precomputation size and the number of additions required for computing $S_{n,r}$ by Pippenger's bucket method and Method II, respectively. The precomputation size presented in this table is in terms of points in $\mathbb{G}_1$ with affine coordinates. The precomputation size in $\mathbb{G}_2$ would be twice that of $\mathbb{G}_1$. Pippenger's bucket method refers to the further optimized method introduced in Section 2.2.3.

Figure 5.1 illustrates the comparison of the number of additions used to compute $S_{n,r}$ by Pippenger's bucket method, BGMW method, Method I and Method II.

Table 5.4: Radix q, length h, precomputation and number of additions for computing $S_{n,r}$

	Pippenger				Method II				Improv.
n	q	h	Precomput.	Additions	q	h	Precomput.	Additions	
2^{10}	2^8	32	96.0 KB	3.69×10^4	2^{10}	26	288 KB	3.23×10^4	12.26%
2^{11}	2^{10}	26	192 KB	6.66×10^4	2^{10}	26	576 KB	5.90×10^4	11.41%
2^{12}	2^{10}	26	384 KB	1.20×10^5	2^{11}	24	1.13 MB	1.09×10^5	9.35%
2^{13}	2^{11}	24	768 KB	2.21×10^5	2^{13}	20	2.25 MB	1.98×10^5	10.31%
2^{14}	2^{12}	22	1.50 MB	4.06×10^5	2^{13}	20	4.50 MB	3.62×10^5	10.67%
2^{15}	2^{13}	20	3.00 MB	7.37×10^5	2^{14}	19	9.00 MB	6.88×10^5	6.74%
2^{16}	2^{13}	20	6.00 MB	1.39×10^6	2^{14}	19	18.0 MB	1.31×10^6	5.92%
2^{17}	2^{16}	16	12.0 MB	2.62×10^6	2^{16}	16	36.0 MB	2.39×10^6	8.80%
2^{18}	2^{16}	16	24.0 MB	4.72×10^6	2^{16}	16	72.0 MB	4.49×10^6	4.89%
2^{19}	2^{16}	16	48.0 MB	8.91×10^6	2^{16}	16	144 MB	8.68×10^6	2.59%
2^{20}	2^{16}	16	96.0 MB	1.73×10^7	2^{19}	14	288 MB	1.62×10^7	6.31%
2^{21}	2^{19}	14	192 MB	3.30×10^7	2^{20}	13	576 MB	3.01×10^7	8.76%

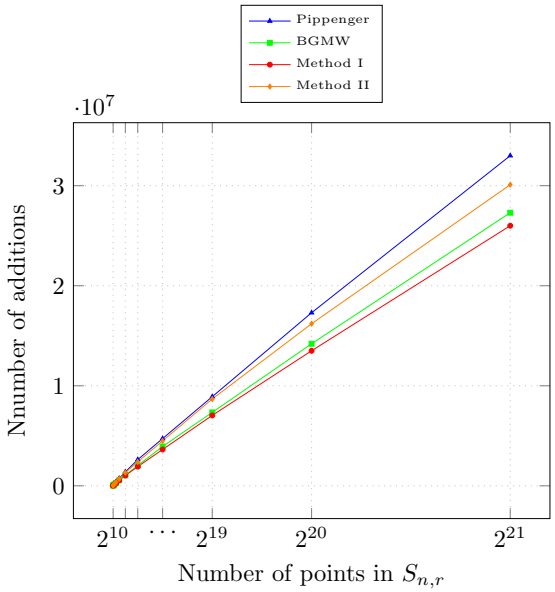


Figure 5.1: Theoretical comparison of the number of additions for computing $S_{n,r}$

5.4 Time complexity: worst case versus average case

In this section, we are going to demonstrate that the difference between the worst case time complexity and the average case time complexity is tiny. Therefore, the worst case time complexity is used in this thesis as the representative. The result relies on the parameter r and the pair of multiplier set and bucket set (M, B), that is why we do the analysis after the instantiation.

Let us start by analyzing the average case time complexity for the proposed Method I. This analysis involves estimating the expected number of zero elements, denoted as f, in the length-nh array `scalars` of Algorithm 5, as shown in Equation (3.11).

Let us denote integer r in its standard q-ary form as

$$r = r_{h-1}||r_{h-2}||\cdots||r_0 = \sum_{j=0}^{h-1} r_j q^j,\ 0 \le r_j < q,$$

where $||$ represents digit concatenation. Remember that r is fixed, so each r_j $(0 \le j \le h-1)$ is also a constant integer and $r_{h-1} \neq 0$. When uniformly and randomly picking a scalar

a $(0 \le a < r)$ and then converting it to the standard q-ary form

$$a = \sum_{j=0}^{h-1} a_j q^j,\ 0 \le a_j < q,$$

the probabilities $\Pr[a_j = 0]$ and $\Pr[a_j = (q-1)]$ are estimated by Lemma 2.

Lemma 2. *For a moderate large q, we have the following approximations,*

$$\Pr[a_j = 0] \approx \frac{1}{q},\ \Pr[a_j = (q-1)] \approx \frac{1}{q},\ 0 \le j \le h-2,$$

and

$$\Pr[a_{h-1} = 0] \approx \frac{1}{r_{h-1}+1}.$$

Proof. For $0 \le j \le h-2$,

- If $r_j \ne 0$, then

$$\begin{aligned}
\Pr[a_j = 0] &= \Pr[a_j = 0 \text{ and } 0 \le a < r] \\
&= \Pr[a_j = 0 \text{ and } 0 \le a_{h-1}||a_{h-2}||\cdots||a_{j+1} \le r_{h-1}||r_{h-2}||\cdots||r_{j+1} \\
&\qquad \text{and arbitrarily pick } 0 \le a_k \le q-1 \text{ for } 0 \le k \le j-1] \\
&= \frac{((r_{h-1}||r_{h-2}||\cdots||r_{j+1})+1)\cdot q^j}{r} \\
&= \frac{((r_{h-1}||r_{h-2}||\cdots||r_{j+1})+1)\cdot q^j}{(r_{h-1}||r_{h-2}||\cdots||r_{j+1})\cdot q^{j+1} + (r_j||r_{j-1}||\cdots||r_0)} \\
&= \frac{1 + 1/(r_{h-1}||r_{h-2}||\cdots||r_{j+1})}{q + (r_j||r_{j-1}||\cdots||r_0)/\left(q^j \cdot (r_{h-1}||r_{h-2}||\cdots||r_{j+1})\right)}.
\end{aligned}$$

  If $r_j = 0$, then

$$\begin{aligned}
\Pr[a_j = 0] &= \Pr[a_j = 0 \text{ and } a < r] \\
&= \Pr[a_j = 0 \text{ and } 0 \le a_{h-1}||a_{h-2}||\cdots||a_{j+1} < r_{h-1}||r_{h-2}||\cdots||r_{j+1} \\
&\qquad \text{and arbitrarily pick } 0 \le a_k \le q-1 \text{ for } 0 \le k \le j-1] + \\
&\quad \Pr[a_j = 0 \text{ and } a_{h-1}||a_{h-2}||\cdots||a_{j+1} = r_{h-1}||r_{h-2}||\cdots||r_{j+1} \\
&\qquad \text{and } 0 \le a_{j-1}||a_{j-2}||\cdots||a_0 < r_{j-1}||r_{j-2}||\cdots||r_0] \\
&= \frac{(r_{h-1}||r_{h-2}||\cdots||r_{j+1})\cdot q^j + (r_{j-1}||r_{j-2}||\cdots||r_0)}{r} \\
&= \frac{(r_{h-1}||r_{h-2}||\cdots||r_{j+1})\cdot q^j + (r_{j-1}||r_{j-2}||\cdots||r_0)}{(r_{h-1}||r_{h-2}||\cdots||r_{j+1})\cdot q^{j+1} + (r_j||r_{j-1}||\cdots||r_0)}.
\end{aligned}$$

- If $r_j \neq q-1$, then

$$\begin{aligned}
\Pr[a_j = (q-1)] &= \Pr[a_j = (q-1) \text{ and } 0 \leq a < r] \\
&= \Pr[a_j = (q-1) \text{ and } 0 \leq a_{h-1}||a_{h-2}|| \cdots ||a_{j+1} < r_{h-1}||r_{h-2}|| \cdots ||r_{j+1} \\
&\qquad \text{and arbitrarily pick } 0 \leq a_k \leq q-1 \text{ for } 0 \leq k \leq j-1] \\
&= \frac{(r_{h-1}||r_{h-2}|| \cdots ||r_{j+1}) \cdot q^j}{r} \\
&= \frac{(r_{h-1}||r_{h-2}|| \cdots ||r_{j+1}) \cdot q^j}{(r_{h-1}||r_{h-2}|| \cdots ||r_{j+1}) \cdot q^{j+1} + (r_j||r_{j-1}|| \cdots ||r_0)} \\
&= \frac{1}{q + (r_j||r_{j-1}|| \cdots ||r_0)/(q^j \cdot (r_{h-1}||r_{h-2}|| \cdots ||r_{j+1}))}.
\end{aligned}$$

  If $r_j = q-1$, then

$$\begin{aligned}
\Pr[a_j = (q-1)] &= \Pr[a_j = (q-1) \text{ and } 0 \leq a < r] \\
&= \Pr[a_j = (q-1) \text{ and } 0 \leq a_{h-1}||a_{h-2}|| \cdots ||a_{j+1} < r_{h-1}||r_{h-2}|| \cdots ||r_{j+1} \\
&\qquad \text{and arbitrarily pick } 0 \leq a_k \leq q-1 \text{ for } 0 \leq k \leq j-1] + \\
&\quad \Pr[a_{h-1}||a_{h-2}|| \cdots ||a_j = r_{h-1}||r_{h-2}|| \cdots ||r_j \\
&\qquad \text{and } 0 \leq a_{j-1}||a_{j-2}|| \cdots ||a_0 < r_{j-1}||r_{j-2}|| \cdots ||r_0] \\
&= \frac{(r_{h-1}||r_{h-2}|| \cdots ||r_{j+1}) \cdot q^j + (r_{j-1}||r_{j-2}|| \cdots ||r_0)}{r} \\
&= \frac{(r_{h-1}||r_{h-2}|| \cdots ||r_{j+1}) \cdot q^j + (r_{j-1}||r_{j-2}|| \cdots ||r_0)}{(r_{h-1}||r_{h-2}|| \cdots ||r_{j+1}) \cdot q^{j+1} + (r_j||r_{j-1}|| \cdots ||r_0)}.
\end{aligned}$$

  Lastly,

$$\Pr[a_{h-1} = 0] = \frac{q^{h-1}}{r} = \frac{q^{h-1}}{\sum_{j=0}^{h-1} r_j q^j} \approx \frac{1}{r_{h-1}+1}.$$

  According to those formulas, the result in the lemma follows immediately. □

Example. Let us assume $r = 4909$, $q = 10$. When uniformly and randomly picking a scalar a ($0 \leq a < r$) and then converting it into $a = a_3||a_2||a_1||a_0 = a_3 \cdot 1000 + a_2 \cdot 100 + a_1 \cdot 10 + a_0$,

we have

$$\begin{aligned}
\Pr[a_2 = 0] &= \frac{5 \cdot 100}{4909} = \frac{500}{4909}, \\
\Pr[a_1 = 0] &= \frac{49 \cdot 10 + 9}{4909} = \frac{499}{4909}, \\
\Pr[a_1 = 9] &= \frac{49 \cdot 10}{4909} = \frac{490}{4909}, \\
\Pr[a_2 = 9] &= \frac{4 \cdot 100 + 9}{4909} = \frac{409}{4909} \\
\Pr[a_3 = 0] &= \frac{1000}{4909}.
\end{aligned}$$

When converting the scalar a by Algorithm 8 from its standard q-ary form to the radix q representation

$$a = \sum_{j=0}^{h-1} m_j b_j q^j, \ m_j \in M, b_j \in B, \tag{5.4}$$

we observe that $b_j = 0$ $(1 \leq j \leq h-2)$ if and only if

(1) $a_j = 0$ and the carry bit from the previous digit $\alpha_{j-1} = 0$, or

(2) $a_j = q - 1$ and the carry bit $\alpha_{j-1} = 1$.

Let us assume the probability of carry bit being 0 is λ, which is equal to the probability of $\alpha = 0$ in the array `decomposition` decided by Algorithm 9. Consequently,

$$\begin{aligned}
\Pr[b_0 = 0] &= \frac{1}{q}, \\
\Pr[b_j = 0] &= \lambda \frac{1}{q} + (1 - \lambda)\frac{1}{q} = \frac{1}{q}, \text{ for } 1 \leq j \leq h-2, \\
\Pr[b_{h-1} = 0] &= \lambda \cdot \frac{1}{r_{h-1} + 1}.
\end{aligned} \tag{5.5}$$

By Equation (5.5), if a random scalar is converted to the representation in Equation (5.4), the expected number of j's such that $b_j = 0$ is

$$\frac{h-1}{q} + \frac{\lambda}{r_{h-1} + 1}.$$

Therefore, the expected number of zeros in the array `scalars` of Algorithm 5 is given by

$$f = \frac{n(h-1)}{q} + \frac{\lambda \cdot n}{r_{h-1} + 1}. \tag{5.6}$$

Let us define

$$I = \frac{\text{worst case time complexity} - \text{average case time complexity}}{\text{worst case time complexity}}$$

to measure the difference between the worst case time complexity and the average case time complexity. Since Method I utilizes q comparable to n (see Table 5.2), the first term $n(h-1)/q$ in f is a small number that can be ignored. It follows that

$$I = \frac{f}{nh + |B| + 2} \approx \frac{\lambda \cdot n/(r_{h-1}+1)}{nh + |B| + 2} < \frac{\lambda \cdot n/(r_{h-1}+1)}{nh} = \frac{\lambda}{(r_{h-1}+1)h}. \tag{5.7}$$

For $q = 2^c$ $(10 \le c \le 22, c \ne 15, 17)$ used in Table 5.2, the triad (q, h, r_{h-1}) can be found in Table 5.1, and it is checked that $\lambda < 0.7$ for these radix values. It follows that

$$I < 1\%,$$

indicating a small difference.

Similar analysis also applies to the proposed Method II, Pippenger's bucket method and BGMW method, where the result in Equation (5.7) still holds. Pippenger's bucket method and BGMW method have $\lambda \approx 0.5$. For $q = 2^c$ $(8 \le c \le 22)$, their I values are even smaller compared to Methods I and II.

Chapter 6

Software Implementation

This chapter presents the software implementation, which mainly includes an overview of the fundamental arithmetic algorithms, the implementation and analysis for Method I, and the implementation and analysis for Method II. The implementation is done over BLS12-381 curve and is based on `blst` library, which is a BLS12-381 signature library written in C and assembly [bls].

6.1 Fundamental arithmetic in implementation

As introduced in Section 2.4, the finite field characteristic of BLS12-381 curve is a 381-bit prime p, and the group order is a 255-bit prime r.

The necessary ingredients for implementing multi-scalar multiplications include the field $\mathbb{F}_p$ and the field $\mathbb{F}_{p^2}$ over which the related elliptic curve groups are defined, the finite field $\mathbb{F}_r$ from which the scalars are randomly selected, and the addition arithmetic in elliptic curve groups.

6.1.1 Base field $\mathbb{F}_p$

The arithmetic operations in $\mathbb{F}_p$, including addition, subtraction, multiplication, and inversion, can be implemented using various ways such as the schoolbook method, Karatsuba method [Sco15], Montgomery reduction [Mon85] or Barrett reduction [Bar86]. Another

roadmap to implement $\mathbb{F}_p$ involves the residue number system [Duq11] and lazy reduction [Sco07, CDF+11]. In this thesis, Montgomery reduction is utilized to implement the field $\mathbb{F}_p$.

For our implementation, we choose the radix $R = 2^{384}$, which is larger than p and coprime to it. The mod R operation can be efficiently done by bit shifting operation. Algorithm 10 presents the Montgomery reduction algorithm, denoted as REDC(a), which efficiently computes $aR^{-1} \bmod p$ for $0 \leq a < Rp$.

Algorithm 10 REDC(a)

Input: R, p and a $(0 \leq a < Rp)$.
Output: $aR^{-1} \bmod p$.

1: Precompute R^{-1}, p' $(0 < R^{-1} < p, 0 < p' < R)$, such that $RR^{-1} - pp' = 1$
2: $m = (a \bmod R)p' \bmod R$, so $0 \leq m < R$
3: $t = (a + mp)/R$
4: **if** $t \geq p$ **then**
5: **return** $t - p$
6: **else**
7: **return** t

The correctness of Algorithm 10 comes from the following fact.

(1) The variable t is an integer, because

$$\begin{aligned} a + mp &\equiv a + ap'p \\ &\equiv a + a(RR^{-1} - 1) \\ &\equiv 0 \pmod{R}. \end{aligned}$$

(2) $t \equiv aR^{-1} \bmod p$, because

$$tR = a + mp \equiv a \pmod{p}.$$

(3) $0 \leq t < 2p$, because

$$0 \leq tR = a + mp < Rp + Rp = 2Rp.$$

For $a \in \mathbb{F}_p$, its Montgomery representation is denoted as $\widetilde{a} = aR \bmod p$. Given two elements a and b in their Montgomery representations, $\widetilde{a}$ and $\widetilde{b}$, the Montgomery representation of their product is $\widetilde{ab} = abR \bmod p$, which can be computed by REDC($\widetilde{a} \cdot \widetilde{b}$) without the mod p operation.

By representing all elements in $\mathbb{F}_p$ as their Montgomery representations and utilizing the REDC algorithm, we can build an arithmetic system for $\mathbb{F}_p$ that avoids the mod p operation. Explicitly,

- Converting an element a to its Montgomery representation $\widetilde{a}$ is done by $\text{REDC}(a \cdot (R^2 \bmod p))$, where $R^2 \bmod p$ is precomputed.
- Addition and subtraction in Montgomery representations stay unchanged. Multiplication in Montgomery representations is performed by $\text{REDC}(\widetilde{a} \cdot \widetilde{b})$.
- The Montgomery representation of the inverse of a is given by $a^{-1}R \bmod p$, which is computed by $\text{REDC}((\widetilde{a})^{-1} \cdot (R^3 \bmod p))$, where $(\widetilde{a})^{-1}$ is computed by the extended Euclidean algorithm and $R^3 \bmod p$ is precomputed.

Today's processors typically use 32-bit or 64-bit words. However, a single word is not sufficient for implementing the field $\mathbb{F}_p$, where p is a 381-bit integer. Therefore, multi-precision arithmetic is needed. For multi-precision arithmetic, please refer to [MVOV18, Chapter14].

6.1.2 Extension field $\mathbb{F}_{p^2}$

For BLS12-381 curve, the following field extension is used,

$$\mathbb{F}_{p^2} = \mathbb{F}_p[\mu]/(\mu^2 - (-1)),$$

where p is defined in Equation (2.50). Addition and subtraction in $\mathbb{F}_{p^2}$ are straightforward, while multiplication, squaring and inversion are worth noting.

Let $a = a_0 + a_1\mu,\ b = b_0 + b_1\mu \in \mathbb{F}_{p^2}$, where $a_i, b_i\ (i = 0, 1) \in \mathbb{F}_p$. The multiplication formula using the schoolbook method is

$$\begin{aligned} ab &= (a_0 + a_1\mu)(b_0 + b_1\mu) \\ &= (a_0b_0 - a_1b_1) + (a_0b_1 + a_1b_0)\mu, \end{aligned}$$

which requires 4 multiplications and 2 additions in $\mathbb{F}_p$. Karatsuba method can be utilized to save one multiplication by computing

$$a_0b_1 + a_1b_0 = (a_0 + a_1)(b_0 + b_1) - a_0b_0 - a_1b_1.$$

Using this method it costs 3 multiplications and 5 additions in $\mathbb{F}_p$. The choice between the two methods depends on the cost of multiplication and addition in $\mathbb{F}_p$. If the cost of one multiplication is higher than that of 3 additions in $\mathbb{F}_p$, then Karatsuba method is preferable.

Because $\mu^2 = -1 \in \mathbb{F}_p$, the squaring is

$$\begin{aligned} a^2 &= (a_0 + a_1\mu)^2 \\ &= (a_0 + a_1)(a_0 - a_1) + 2a_0a_1\mu. \end{aligned}$$

The inversion is

$$\begin{aligned} a^{-1} &= \frac{1}{a_0 + a_1\mu} \\ &= \frac{a_0 - a_1\mu}{(a_0 + a_1\mu)(a_0 - a_1\mu)} \\ &= \frac{a_0 - a_1\mu}{a_0^2 + a_1^2} \\ &= \frac{a_0}{a_0^2 + a_1^2} + \left(-\frac{a_1}{a_0^2 + a_1^2}\right)\mu. \end{aligned}$$

6.1.3 Addition formulas in elliptic curve groups

When computing multi-scalar multiplication $S_{n,r}$, all the precomputed points are stored in affine coordinates in order to save memory. Addition in affine coordinates requires a field inversion operation, which is significantly more costly than a field multiplication. To avoid this inversion, popular coordinate systems such as projective coordinates and Jacobian coordinates can be used. In our implementation, Jacobian coordinates are utilized.

The elliptic curve group in Jacobian coordinates can be represented as

$$E(\mathbb{F}) = \{(X, Y, Z) \in \mathbb{F}^3 \mid Y^2 = X^3 + AXZ^4 + BZ^6\}, \tag{6.1}$$

where $A, B \in \mathbb{F}$ are constant. The point at infinity is $\infty = (1, 1, 0)$. The inverse of a point $P = (X_1, Y_1, Z_1)$ is given by $-P = (X_1, -Y_1, Z_1)$. For the point $P = (X_1, Y_1, Z_1)$ in Jacobian coordinates, it remains the same point after being transformed to $(\lambda^2 X_1, \lambda^3 Y_1, \lambda Z_1)$ for $\lambda \neq 0$. If $Z_1 \neq 0$, the point P in its Jacobian coordinates can be represented in affine coordinates as $(X_1/Z_1^2, Y_1/Z_1^3)$.

Frameworks I and II presented in Section 3.2 can be roughly divided into two phases. Phase one is sorting all points into subsums with respect to their scalars, and phase two

involves accumulating the subsums together to obtain the final result. In phase one, addition is computed between a point in affine coordinates and another point in Jacobian coordinates, which is known as *mixed addition*. In phase two, addition is evaluated between two points in Jacobian coordinates.

Addition in Jacobian coordinates

Let $P = (X_1, Y_1, Z_1)$, $Q = (X_2, Y_2, Z_2)$ be two points given in Jacobian coordinates, and $R = P + Q$. The Jacobian coordinates of $R = (X_3, Y_3, Z_3)$ are computed by

$$\begin{aligned} C_1 &= X_1 Z_2^2, \\ C_2 &= X_2 Z_1^2, \\ D_1 &= Y_1 Z_2^3, \\ D_2 &= Y_2 Z_1^3, \\ E &= C_2 - C_1, \\ F &= D_2 - D_1, \end{aligned} \tag{6.2}$$

and

$$\begin{cases} X_3 = -E^3 - 2C_1 E^2 + F^2 \\ Y_3 = -D_1 E^3 + F(C_1 E^2 - X_3) \\ Z_3 = E Z_1 Z_2, \end{cases} \tag{6.3}$$

which cost 12 multiplications and 4 squares (i.e., Z_1^2, Z_2^2, E^2, F^2) in $\mathbb{F}$.

Let $P = (X_1, Y_1, Z_1)$ be a point given in Jacobian coordinates, and $R = P + P$. The Jacobian coordinates of $R = (X_3, Y_3, Z_3)$ are given by

$$\begin{aligned} C &= 4X_1 Y_1^2, \\ D &= 3X_1^2 + AZ_1^4, \end{aligned} \tag{6.4}$$

and

$$\begin{cases} X_3 = -2C + D^2 \\ Y_3 = -8Y_1^4 + D(C - X_3) \\ Z_3 = 2Y_1 Z_1, \end{cases} \tag{6.5}$$

which cost 4 multiplications and 6 squares (i.e., $Y_1^2, (Y_1^2)^2, X_1^2, Z_1^2, (Z_1^2)^2, D^2$) in $\mathbb{F}$.

Mixed addition

Let $P = (X_1, Y_1, Z_1)$ be a point given in Jacobian coordinates, $Q = (x_2, y_2)$ be a point given in affine coordinates, and $R = P + Q$, then the Jacobian coordinates of $R = (X_3, Y_3, Z_3)$ are given by

$$\begin{aligned} C &= x_2 Z_1^2, \\ D &= y_2 Z_1^3, \\ E &= C - X_1, \\ F &= D - Y_1, \end{aligned} \tag{6.6}$$

and

$$\begin{cases} X_3 = -E^3 - 2X_1E^2 + F^2 \\ Y_3 = -Y_1E^3 + F(X_1E^2 - X_3) \\ Z_3 = EZ_1, \end{cases} \tag{6.7}$$

which require 8 multiplications and 3 squares in $\mathbb{F}$.

Let $P = (X_1, Y_1, Z_1)$ be a point of the curve given in Jacobian coordinates, and $R = P + P$, then the Jacobian coordinates of $R = (X_3, Y_3, Z_3)$ are given by

$$\begin{aligned} C &= 4X_1Y_1^2, \\ D &= 3X_1^2 + A, \end{aligned} \tag{6.8}$$

and

$$\begin{cases} X_3 = -2C + D^2 \\ Y_3 = -8Y_1^4 + D(C - X_3) \\ Z_3 = 2Y_1, \end{cases} \tag{6.9}$$

which cost 3 multiplications and 4 squares (i.e., $Y_1^2, (Y_1^2)^2, X_1^2, D^2$) in $\mathbb{F}$. For BLS12-381 curve, the constant $A = 0$, so it only requires 4 multiplications and 4 squares in $\mathbb{F}$.

Utilize intermediate result

Further savings can be achieved by storing the coordinates of a point as (X, Y, Z^2, Z^3), which corresponds to Jacobian coordinates (X, Y, Z). For example, this technique saves 2 squares in $\mathbb{F}$ when doing addition by Equations (6.2)(6.3). Let $P = (X_1, Y_1, Z_1^2, Z_1^3)$, $Q = (X_2, Y_2, Z_2^2, Z_2^3)$ and $R = (X_3, Y_3, Z_3^2, Z_3^3)$. The modified formulas compute $Z_3^2 = E^2Z_1^2Z_2^2$ and $Z_3^3 = E^3Z_1^3Z_2^3$ instead of computing $Z_3 = EZ_1Z_2$ in the last step. In this case, the addition takes 12 multiplications and 2 squares (i.e., E^2, F^2) in $\mathbb{F}$.

This technique also saves the cost for determining whether the operation is an addition or doubling by checking if $P = Q$. We have that $P = Q$ if and only if E and F in Equation (6.2) are 0.

6.1.4 Scalar field $\mathbb{F}_r$

In multi-scalar multiplication $S_{n,r}$, the field from which the scalars are randomly picked is referred to as the scalar field $\mathbb{F}_r$. The field $\mathbb{F}_r$ is the arena in which the scalar decomposition is done. In our implementation, the scalar decomposition does not involve the mod r operation and field inversion. The field $\mathbb{F}_r$ behaves more like the integer ring. Therefore, the implementation of $\mathbb{F}_r$ adopts the schoolbook multi-precision arithmetic, in contrast to the implementation of $\mathbb{F}_p$ which adopts Montgomery representation.

The scalars in $S_{n,r}$ are expected to be uniformly and randomly chosen from $\mathbb{F}_r$ when performing tests for computing $S_{n,r}$. However, there is a small obstacle. A cryptographically secure pseudorandom bit generator (PRBG) usually generates an ℓ-bit sequence, which is equivalent to uniformly and randomly pick an integer from $[0, 2^\ell - 1]$. This obstacle is overcome by the trick shown in Algorithm 11. SHA256 is used as the PRBG in our implementation.

Algorithm 11 Uniformly and randomly choose an element from $\mathbb{F}_r$

Input: Integer r, a pseudorandom bit generator PRBG.
Output: An element in $\mathbb{F}_r$.

1: Compute $\ell = \lceil \log_2 r \rceil$ and initiate $t = r$
2: **while** $t \geq r$ **do**
3: Utilize the PRBG to generate an ℓ-bit integer t
4: **return** t

6.2 Test for Method I

The performance concern about Method I is mainly focused on two aspects. Firstly, it involves converting all scalars in $S_{n,r}$ to the radix q representation where every digit is the product of an element from the multiplier set and another element from the bucket set. Secondly, it requires retrieving data from the large precomputation tables. In order to assess the cost of scalar conversions and the impact of memory locality issue caused by the large precomputation size, we conducted the experiments. Our implementation is based

on `blst`, a BLS12-381 signature library written in C and assembly [bls]. The `blst` library includes the point addition/doubling arithmetic and the implementation of Pippenger's bucket method in $\mathbb{G}_1$ and $\mathbb{G}_2$ over BLS12-381. We implemented BGMW method and Method I following Algorithm 5, we invoked the Pippenger's bucket method built in the `blst` library.

6.2.1 Implementation analysis

Regarding scalar conversion, which corresponds to Step 2 of Algorithm 5, a scalar in $\mathbb{F}_r$ is given as a length-4 `uint64_t` array. Both Pippenger's bucket method and BGMW method need to first convert the scalar to its standard q-ary form, then convert to the expression where the absolute value of every digit is no more than $q/2$ using Algorithm 2. Method I first converts the scalar to its standard q-ary form, then converts to the expression where every digit is the product of an element from the multiplier set and an element from the bucket set using Algorithm 8 with the help of the decomposition hash table. This hash table is realized as a length-$(q+1)$ array `decomposition`. Therefore, the performance concern for scalar conversion boils down to retrieving data from the array `decomposition`.

Regarding Step 4 in Algorithm 5, where all points are sorted into different buckets, the addition is performed between a point fetched from the array `precomputation` and another point fetched from the array `buckets`. We treat the n fixed points in $S_{n,r}$ as the length-n `precomputation` array for Pippenger's bucket method. We have the following observations.

- Pippenger's bucket method performs the computation in Equation (2.18) h times, so in total it fetches data nh times from its length-n `precomputation` array, and it fetches data nh times from its length-$(0.5q+1)$ `buckets` array.
- BGMW method fetches data nh times from its length-nh `precomputation` array, and it fetches data nh times from its length-$(0.5q+1)$ `buckets` array.
- Method I fetches data nh times from its length-$3nh$ `precomputation` array, and it fetches data nh times from its `buckets` array, whose length is approximately $0.21q$ (when $q \neq 2^c$, $c \in \{15, 16, 17\}$).

BGMW method and Method I show some advantages here regarding the number of fetch operations, since their h values are usually smaller than that of Pippenger's bucket method. Their disadvantages originate from the fact that their fetch operations are executed in larger

`precomputation` and `buckets` arrays. Step 4 of Algorithm 5 is a simple loop, so we utilize prefetching to mitigate the latency of memory access to these large arrays.

It should be noted that in terms of fetching data from the `buckets` array, Method I has an advantage against BGMW method when the same radix q is used, as Method I uses a smaller `buckets` array in this case. Even if its radix q ($q \neq 2^{16}$) is twice as big, Method I still maintains such advantage.

6.2.2 Experimental result

The experiment was conducted on an Apple 14-inch MacBook Pro equipped with a 3.2 GHz M1 Pro chip and 16 GB of memory, running as a single thread. The M1 Pro chip offers advantages such as large cache size and high memory bandwidth. Most importantly, its cache line is 128 bytes, which is sufficient to accommodate a BLS12-381 $\mathbb{G}_1$ point with a size of 96 bytes. These characteristics are expected to provide us some benefit when fetching data from large arrays.

The experimental results are presented in Tables 6.1, 6.2 and 6.3. Both BGMW method and Method I use the optimal radixes presented in Table 5.2, while the Pippenger's bucket method built in `blst` utilizes radixes slightly different from the radixes suggested in Table 5.2, explicitly,

$$q = \begin{cases} 2^{e-2} & \text{for } n = 2^e \ (10 \leq e \leq 12), \\ 2^{e-3} & \text{for } n = 2^e \ (13 \leq e \leq 21). \end{cases}$$

We keep `blst`'s implementation intact, because on one hand our focus is on the comparison between BGMW method and Method I, on the other hand `blst`'s implementation can serve as a performance benchmark.

Table 6.1: Time for computing $S_{n,r}$ by different methods[1]

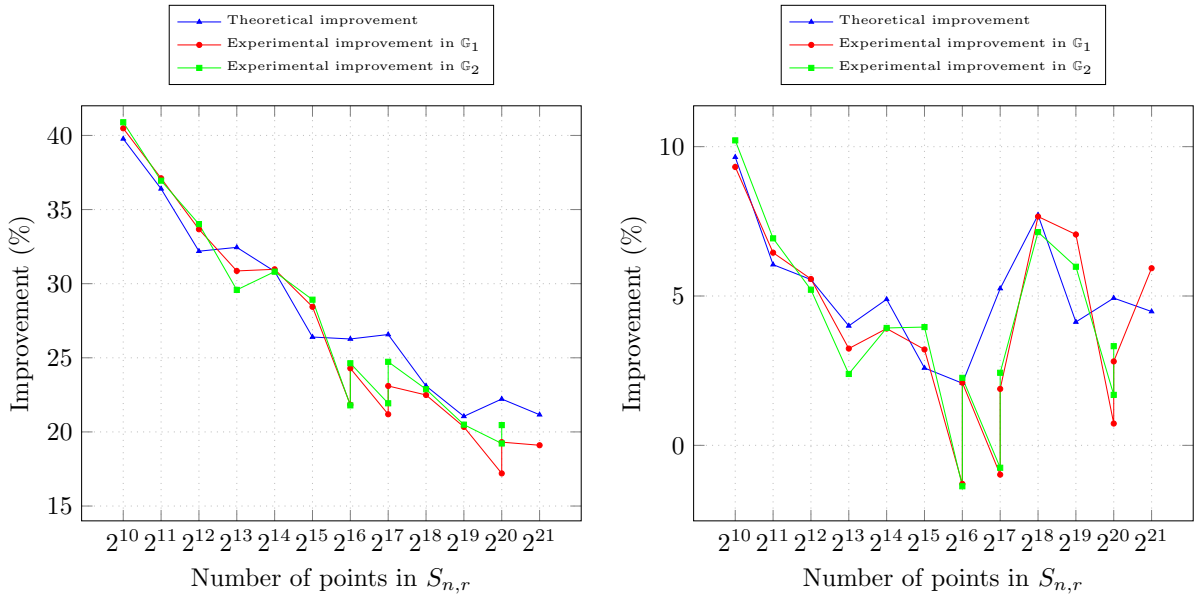
	$\mathbb{G}_1$			$\mathbb{G}_2$		
n	Pipp.	BGMW	Method I	Pipp.	BGMW	Method I
2^{10}	15.1 ms	9.89 ms	8.96 ms	37.2 ms	24.5 ms	22.0 ms
2^{11}	27.2 ms	18.3 ms	17.1 ms	66.6 ms	45.1 ms	42.0 ms
2^{12}	48.7 ms	34.2 ms	32.3 ms	120 ms	83.2 ms	78.9 ms
2^{13}	89.7 ms	64.1 ms	62.0 ms	218 ms	157 ms	154 ms
2^{14}	165 ms	118 ms	114 ms	401 ms	289 ms	278 ms
2^{15}	302 ms	223 ms	216 ms	737 ms	546 ms	524 ms
2^{16}	551 ms	425 ms	431 ms	1.35 s	1.04 s	1.05 s
$*2^{16}$	551 ms	426 ms	417 ms	1.34 s	1.04 s	1.01 s
2^{17}	1.04 s	814 ms	822 ms	2.55 s	1.98 s	1.99 s
$*2^{17}$	1.04 s	816 ms	801 ms	2.55 s	1.96 s	1.92 s
2^{18}	1.93 s	1.62 s	1.50 s	4.72 s	3.92 s	3.64 s
2^{19}	3.56 s	3.05 s	2.84 s	8.69 s	7.35 s	6.91 s
2^{20}	6.89 s	5.75 s	5.71 s	16.8 s	13.8 s	13.6 s
$*2^{20}$	6.90 s	5.73 s	5.57 s	16.8 s	13.9 s	13.4 s
2^{21}	13.4 s	11.5 s	10.8 s	–	–	–

[1]We did not do test in $\mathbb{G}_2$ for $n = 2^{21}$ due to the memory size restriction of the test device.

Table 6.2: Method I versus Pippenger's bucket method and BGMW method

	$\mathbb{G}_1$		$\mathbb{G}_2$	
n	Improv1	Improv2	Improv1	Improv2
2^{10}	40.48%	9.32%	40.89%	10.21%
2^{11}	37.11%	6.45%	36.95%	6.93%
2^{12}	33.67%	5.57%	34.02%	5.21%
2^{13}	30.86%	3.24%	29.58%	2.39%
2^{14}	30.97%	3.91%	30.81%	3.93%
2^{15}	28.44%	3.21%	28.92%	3.96%
2^{16}	21.83%	-1.29%	21.79%	-1.37%
$*2^{16}$	24.29%	2.10%	24.63%	2.26%
2^{17}	21.19%	-0.98%	21.93%	-0.75%
$*2^{17}$	23.10%	1.89%	24.74%	2.43%
2^{18}	22.49%	7.66%	22.88%	7.14%
2^{19}	20.34%	7.06%	20.49%	5.98%
2^{20}	17.20%	0.73%	19.22%	1.69%
$*2^{20}$	19.31%	2.81%	20.46%	3.32%
2^{21}	19.10%	5.93%	–	–

In Table 6.2, Improv1 refers to the comparison between Pippenger's bucket method and Method I, while Improv2 is the comparison between BGMW method and Method I. The results in Table 6.2, combined with the theoretical improvement shown in Table 5.3, are visualized in Figure 6.1.



(a) Pippenger versus Method I

(b) BGMW versus Method I

Figure 6.1: Improvement against Pippenger's bucket method and BGMW method

Table 6.3 shows the percentages of time spent by BGMW method and Method I to conduct the scalar conversions in Step 2 of Algorithm 5 for all n scalars in $S_{n,r}$. They are the percentages of time taken by the scalar conversions out of the entire $S_{n,r}$ computation.

Both BGMW method and Method I show significant improvements compared to Pippenger's bucket method, which demonstrates the feasibility of speeding up the computation of $S_{n,r}$ using large precomputation tables.

If we focus on the comparison between BGMW method and Method I, we have the following observations when computing $S_{n,r}$ in $\mathbb{G}_1$ for $n = 2^e$ $(10 \leq e \leq 21)$, and in $\mathbb{G}_2$ for $n = 2^e$ $(10 \leq e \leq 20)$,

- In terms of the scalar conversion time, in $\mathbb{G}_1$, BGMW method takes 1.00%–1.35% out of its entire $S_{n,r}$ computation time, while Method I takes 1.08%–2.20%. Because in $\mathbb{G}_2$ the point addition arithmetic takes relatively more time compared to that in $\mathbb{G}_1$, the percentages are smaller. In $\mathbb{G}_2$, BGMW method takes 0.41%–0.56% out of its whole $S_{n,r}$ computation time, while Method I takes 0.45%–0.88%.
- Compared to the theoretical analysis, Method I does not perform well experimentally for $n = 2^{16}, 2^{17}, 2^{20}$. For $n = 2^{16}$, the optimal radix of Method I is $q = 2^{19}$, which is 4

Table 6.3: Comparison of scalar conversion time by BGMW method and Method I

	$\mathbb{G}_1$		$\mathbb{G}_2$	
n	BGMW	Method I	BGMW	Method I
2^{10}	1.00%	1.16%	0.41%	0.49%
2^{11}	1.05%	1.30%	0.42%	0.53%
2^{12}	1.16%	1.43%	0.48%	0.58%
2^{13}	1.06%	1.08%	0.43%	0.45%
2^{14}	1.15%	1.16%	0.47%	0.48%
2^{15}	1.14%	1.22%	0.47%	0.50%
2^{16}	1.13%	1.23%	0.46%	0.50%
$*2^{16}$	1.19%	1.54%	0.46%	0.62%
2^{17}	1.18%	1.55%	0.49%	0.64%
$*2^{17}$	1.18%	1.36%	0.49%	0.64%
2^{18}	1.31%	1.63%	0.54%	0.67%
2^{19}	1.26%	1.65%	0.53%	0.70%
2^{20}	1.34%	2.11%	0.56%	0.88%
$*2^{20}$	1.35%	1.74%	0.56%	0.71%
2^{21}	1.22%	2.20%	–	–

times larger than that of BGMW method. For $n = 2^{17}$, the optimal radix of Method I is $q = 2^{20}$, which is 8 times larger than that of BGMW method. For $n = 2^{20}$, the optimal radix of Method I is $q = 2^{22}$, which is 4 times larger than that of BGMW method. Since the radix values are even larger than n and the temp variables take way more space than the cache size, it has a negative impact on fetching data from the `buckets` array, as the analysis in Section 6.2.1 indicates. When we try smaller radix values for Method I, specifically $q = 2^{18}$ for $n = 2^{16}$, $q = 2^{19}$ for $n = 2^{17}$, and $q = 2^{20}$ for $n = 2^{20}$, Method I performs better, as shown by the results marked with asterisks, although these radixes are not theoretically optimal.

- Method I outperforms BGMW method for $n = 2^e$ ($10 \leq e \leq 21, e \neq 16, 17$). In these cases, Method I demonstrates 0.73%–10.21% improvement against BGMW method, as shown in Table 6.2.

6.3 Test for Method II

Compared to Pippenger's bucket method, the overhead in Method II mainly arises from scalar conversions and the memory access latency caused by fetching data from the large precomputation tables. This is similar to Method I, thus a detailed analysis for Method II is omitted.

The experiment was conducted on an Apple 14-inch MacBook Pro equipped with a 3.2 GHz M1 Pro chip and 16 GB of memory, running as a single thread. The experimental results are presented in Table 6.4. The improvement column in the table refers to the comparison between Pippenger's bucket method and Method II. The comparison is visualized in Figure 6.2.

Table 6.4: Time for computing $S_{n,r}$ by Pippenger's bucket method and Method II

	$\mathbb{G}_1$			$\mathbb{G}_2$		
n	Pippenger	Method II	Improvement	Pippenger	Method II	Improvement
2^{10}	15.1 ms	13.8 ms	8.45%	37.2 ms	34.0 ms	8.47%
2^{11}	27.1 ms	24.2 ms	10.61%	66.6 ms	59.1 ms	11.27%
2^{12}	48.7 ms	44.3 ms	9.03%	119 ms	108 ms	9.54%
2^{13}	89.2 ms	83.9 ms	5.95%	219 ms	208 ms	5.17%
2^{14}	164 ms	149 ms	9.24%	402 ms	363 ms	9.59%
2^{15}	301 ms	281 ms	6.85%	740 ms	685 ms	7.42%
2^{16}	551 ms	524 ms	4.87%	1.34 s	1.28 s	5.07%
2^{17}	1.04 s	998 ms	4.14%	2.55 s	2.46 s	3.54%
2^{18}	1.93 s	1.83 s	5.29%	4.73 s	4.45 s	5.94%
2^{19}	3.57 s	3.48 s	2.48%	8.67 s	8.42 s	2.83%
2^{20}	6.90 s	6.71 s	2.73%	16.8 s	16.3 s	2.92%
2^{21}	13.3 s	12.5 s	5.76%	32.4 s	30.6 s	5.64%

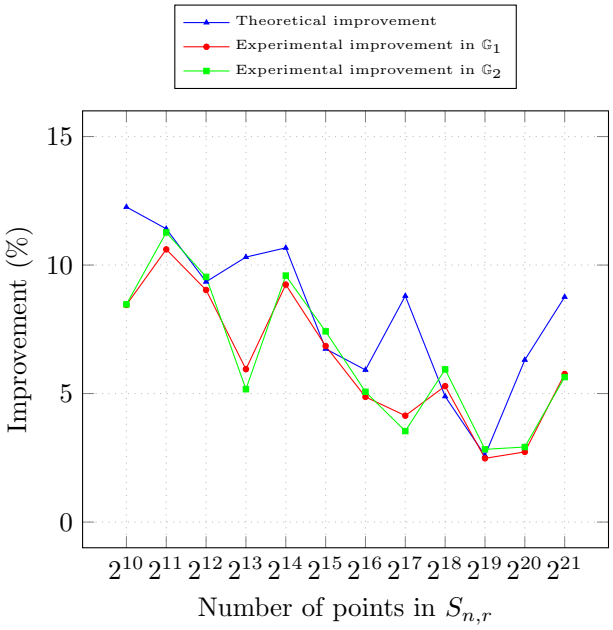


Figure 6.2: Pippenger's bucket method versus Method II

Chapter 7

Conclusion and Future Work

7.1 Conclusion

This thesis mainly covers the following contents.

- In Chapters 1 and 2, the motivation and background knowledge are introduced. The classical multi-scalar multiplication algorithms including the trivial method, window method, Pippenger's bucket method, BGMW method and comb method are reviewed. Two influential pairing-based trusted setup schemes, KZG commitment and Groth16 zkSNARK scheme, are introduced.
- In Chapters 3 and 4, theoretical techniques for computing the multi-scalar multiplication over fixed points $S_{n,r}$ are presented. Two frameworks for computing $S_{n,r}$ and an accumulation algorithm associated with these frameworks are proposed. The potential benefits of utilizing GLV endomorphism and affine coordinates are briefly discussed. Five pairs of bucket set and multiplier set constructions that can be used with the proposed frameworks are proposed.
- In Chapters 5 and 6, instantiation and experiments are conducted. Two concrete methods for computing $S_{n,r}$ in the BLS12-381 groups are proposed and analyzed. The effectiveness of these two methods are demonstrated by experiments.

The objective of these contents is to establish efficient methods for computing multi-scalar multiplication over fixed points, which is an essential and time-consuming operation

in pairing-based trusted setup zkSNARK schemes. For instance, the proof of Groth16 presented in Section 2.3.2 is generated by 3 multi-scalar multiplications. Therefore, any speed improvement achieved in the computation of multi-scalar multiplication would directly translate to the improvement of Groth16's proof generation.

The contributions can be structured to the following three aspects, which better align with the actual research process.

- If the devices have sufficient large memory, they can adopt the methods that are yielded by combining the proposed Framework I in Section 3.2.1 and Constructions I, II, III in Sections 4.1, 4.2, 4.3, respectively. These methods utilize large precomputation tables to achieve better time efficiency. Specifically, by instantiating Framework I and Construction I in BLS12-381 groups whose order r is the 255-bit prime in Equation (2.51), one can obtain Method I in Section 5.2. When computing $S_{n,r}$ in the BLS12-381 groups,
  - Theoretically, Method I saves 21.05%–39.77% additions compared to Pippenger's bucket method for $n = 2^e$ $(10 \leq e \leq 21)$, and it saves 2.08%–9.65% additions compared to BGMW method for $n = 2^e$ $(10 \leq e \leq 21)$. The detailed theoretical comparison is presented in Section 5.2.1.
  - Experimentally, Method I saves 17.20%–40.89% of the computational time compared to Pippenger's bucket method for $n = 2^e$ $(10 \leq e \leq 21)$, and it saves 0.73%–10.21% of the computational time compared to BGMW method for $n = 2^e$ $(10 \leq e \leq 21, e \neq 16, 17)$. The detailed experimental comparison is presented in Section 6.2.
- If the devices have relatively constrained memory, they can adopt the methods that are yielded by combining the proposed Framework II in Section 3.2.2 and Constructions I, II, III, respectively. These methods require only a fraction of precomputation size compared to Framework I, while still offering speed improvement compared to Pippenger's bucket method. Specifically, by instantiating Framework II and Construction I in the BLS12-381 groups, one can obtain Method II in Section 5.3.

  When computing $S_{n,r}$ for $n = 2^e$ $(10 \leq e \leq 21)$ in the BLS12-381 groups and compared to Pippenger's bucket method, Method II saves 2.59%–12.26% additions theoretically, and it saves 2.48%–11.27% of the computational time experimentally. The detailed theoretical comparison is presented in Section 5.3.1, while the detailed experimental comparison is presented in Section 6.3.

- The theoretical limits of the proposed frameworks are explored by Constructions IV and V in Section 4.4. Because these two constructions rely on radixes that are not the powers of 2, their radix conversions are costly. These two constructions are primarily of theoretical significance.

7.2 Future work

For the goal of computing n-scalar multiplication over fixed points with large n, particularly for supporting pairing-based zkSNARK applications, the following two directions, one theoretical and one practical, are worth putting effort into.

7.2.1 Small bucket set constructions

The fourth and fifth multiplier set and bucket set constructions in Section 4.4 explore the possible theoretical limits. As the radixes in these constructions are not the powers of 2, the radix conversions are much slower than that when the radix values $q = 2^c$. The complexity advantage for computing multi-scalar multiplication brought by the smaller bucket set constructions would be largely offset by the radix conversions.

Given $q = 2^c$ and ℓ a small integer, an interesting problem is to construct a pair of multiplier set and bucket set (M, B), such that

$$M = \{-m_\ell, -m_{\ell-1}, \cdots, -m_1, m_1, m_2, \cdots, m_\ell\},$$

where each m_i $(1 \leq i \leq \ell)$ is a positive integer, and at the same time making the size of B as small as possible, particularly, trying to achieve

$$|B| \approx \frac{q}{2\ell}.$$

The difficulty boils down to the fact that this bucket set size is asymptotically smallest, as analyzed in Section 4.4.2.

7.2.2 Better software implementations

In Step 3 of Algorithm 5 and Step 3 of Algorithm 6, it is required to fetch the appropriate points from the huge precomputation tables. Additionally, scalar conversions by Algorithm

8 also rely on fetching data from the precomputed digit decomposition hash tables. These two processes heavily affect memory access latency, therefore it is worth exploring better hardware and better implementations to reduce the data fetching time.

It is also worth considering to have multi-thread implementations for the proposed frameworks, especially for these steps before the last accumulation step. Both Algorithm 5 and Algorithm 6 essentially involve adding nh points into their respective buckets. We can try to parallelize this process. The challenge lies in enabling each thread to simultaneously and quickly access the huge precomputation tables in memory.

Finally, it is worth investing effort to integrate the proposed methods into the actual blockchain applications. This would allow us to effectively assess the overall time efficiency improvement.

References

[AFCK+12] Diego F Aranha, Laura Fuentes-Castaneda, Edward Knapp, Alfred Menezes, and Francisco Rodríguez-Henríquez. Implementing pairings at the 192-bit security level. In *International Conference on Pairing-Based Cryptography*, pages 177–195. Springer, 2012.

[AKT20] Thomas Dybdahl Ahle, Jakob Tejs Bæk Knudsen, and Mikkel Thorup. The power of hashing with Mersenne primes. *arXiv preprint arXiv:2008.08654*, 2020.

[Azt] Aztec: A zkRollup on Ethereum, the most secure settlement layer in crypto. https://aztec.network/.

[bab86] On Lovász' lattice reduction and the nearest lattice point problem. *Combinatorica*, 6:1–13, 1986.

[Bar86] Paul Barrett. Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor. In *Conference on the Theory and Application of Cryptographic Techniques*, pages 311–323. Springer, 1986.

[BC89] Jurjen Bos and Matthijs Coster. Addition chain heuristics. In *Conference on the Theory and Application of Cryptology*, pages 400–407. Springer, 1989.

[BCG+14] Eli Ben-Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. Zerocash: Decentralized anonymous payments from bitcoin. In *2014 IEEE symposium on security and privacy*, pages 459–474. IEEE, 2014.

[BCG+15] Eli Ben-Sasson, Alessandro Chiesa, Matthew Green, Eran Tromer, and Madars Virza. Secure sampling of public parameters for succinct zero knowl-

edge proofs. In *2015 IEEE Symposium on Security and Privacy*, pages 287–304. IEEE, 2015.

[BCTV13] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. Succinct non-interactive zero knowledge for a von Neumann architecture. Cryptology ePrint Archive, Paper 2013/879, 2013. https://eprint.iacr.org/2013/879.

[BD19] Razvan Barbulescu and Sylvain Duquesne. Updating key size estimations for pairings. *Journal of Cryptology*, 32(4):1298–1336, 2019.

[BDLO12] Daniel J Bernstein, Jeroen Doumen, Tanja Lange, and Jan-Jaap Oosterwijk. Faster batch forgery identification. In *International Conference on Cryptology in India*, pages 454–473. Springer, 2012.

[Ber06] Daniel J Bernstein. Differential addition chains. https://cr.yp.to/ecdh/diffchain-20060219.pdf, 2006.

[BGG18] Sean Bowe, Ariel Gabizon, and Matthew D Green. A multi-party protocol for constructing the public parameters of the Pinocchio zk-SNARK. In *International Conference on Financial Cryptography and Data Security*, pages 64–77. Springer, 2018.

[BGK15] Razvan Barbulescu, Pierrick Gaudry, and Thorsten Kleinjung. The tower number field sieve. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 31–55. Springer, 2015.

[BGMW92] Ernest F Brickell, Daniel M Gordon, Kevin S McCurley, and David B Wilson. Fast exponentiation with precomputation. In *Workshop on the Theory and Application of of Cryptographic Techniques*, pages 200–207. Springer, 1992.

[bls] blst: A BLS12-381 signature library focused on performance and security written in C and assembly. https://github.com/supranational/blst.

[BLS02] Paulo SLM Barreto, Ben Lynn, and Michael Scott. Constructing elliptic curves with prescribed embedding degrees. In *International Conference on Security in Communication Networks*, pages 257–267. Springer, 2002.

[BN05] Paulo SLM Barreto and Michael Naehrig. Pairing-friendly elliptic curves of prime order. In *International Workshop on Selected Areas in Cryptography*, pages 319–331. Springer, 2005.

[Bow17] Sean Bowe. BLS12-381: New zk-SNARK elliptic curve construction, 2017.

[Bro15] Daniel R Brown. Multi-dimensional Montgomery ladders for elliptic curves, 2015. US Patent 8,958,551.

[CDF+11] Ray CC Cheung, Sylvain Duquesne, Junfeng Fan, Nicolas Guillermin, Ingrid Verbauwhede, and Gavin Xiaoxu Yao. FPGA implementation of pairings using residue number system and lazy reduction. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 421–441. Springer, 2011.

[CGGN17] Matteo Campanelli, Rosario Gennaro, Steven Goldfeder, and Luca Nizzardo. Zero-knowledge contingent payments revisited: Attacks and payments for services. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 229–243, 2017.

[CH06] Jaewook Chung and M Anwar Hasan. Low-weight polynomial form integers for efficient modular multiplication. *IEEE Transactions on Computers*, 56(1):44–57, 2006.

[CHM+20] Alessandro Chiesa, Yuncong Hu, Mary Maller, Pratyush Mishra, Noah Vesely, and Nicholas Ward. Marlin: Preprocessing zkSNARKs with universal and updatable SRS. In *Advances in Cryptology – EUROCRYPT 2020, Proceedings, Part I 39*, pages 738–768. Springer, 2020.

[DFGK14] George Danezis, Cédric Fournet, Jens Groth, and Markulf Kohlweiss. Square span programs with applications to succinct NIZK arguments. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 532–550. Springer, 2014.

[DKS09] Christophe Doche, David R Kohel, and Francesco Sica. Double-base number system for multi-scalar multiplications. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 502–517. Springer, 2009.

[DR94] Peter De Rooij. Efficient exponentiation using precomputation and vector addition chains. In *Workshop on the Theory and Application of of Cryptographic Techniques*, pages 389–399. Springer, 1994.

[Duq11] Sylvain Duquesne. RNS arithmetic in $\mathbb{F}_{p^k}$ and application to fast pairing computation. *Journal of Mathematical Cryptology*, 5(1):51–88, 2011.

[EMJ17] Nadia El Mrabet and Marc Joye. *Guide to pairing-based cryptography*. CRC Press, 2017.

[GF16] Loubna Ghammam and Emmanuel Fouotsa. On the computation of the optimal ate pairing at the 192-bit security level. *IACR Cryptol. ePrint Arch.*, 2016:130, 2016.

[GGPR13] Rosario Gennaro, Craig Gentry, Bryan Parno, and Mariana Raykova. Quadratic span programs and succinct NIZKs without PCPs. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 626–645. Springer, 2013.

[GJW20] Ariel Gabizon and Zachary J. Williamson. Proposal: The Turbo-PLONK program syntax for specifying SNARK programs. 2020.

[GKM+18] Jens Groth, Markulf Kohlweiss, Mary Maller, Sarah Meiklejohn, and Ian Miers. Updatable and universal common reference strings with applications to zk-SNARKs. In *Advances in Cryptology – CRYPTO 2018, Proceedings, Part III*, pages 698–728. Springer, 2018.

[GLS11] Steven D Galbraith, Xibin Lin, and Michael Scott. Endomorphisms for faster elliptic curve cryptography on a large class of curves. *Journal of cryptology*, 24(3):446–469, 2011.

[GLV01] Robert P Gallant, Robert J Lambert, and Scott A Vanstone. Faster point multiplication on elliptic curves with efficient endomorphisms. In *Annual International Cryptology Conference*, pages 190–200. Springer, 2001.

[GMT20] Aurore Guillevic, Simon Masson, and Emmanuel Thomé. Cocks–Pinch curves of embedding degrees five to eight and optimal ate pairing computation. *Designs, Codes and Cryptography*, 88(6):1047–1081, 2020.

[gna] gnark zk-SNARK library. https://github.com/ConsenSys/gnark.

[Gro10] Jens Groth. Short pairing-based non-interactive zero-knowledge arguments. In *Asiacrypt*, volume 6477, pages 321–340. Springer, 2010.

[Gro16] Jens Groth. On the size of pairing-based non-interactive arguments. In *Annual international conference on the theory and applications of cryptographic techniques*, pages 305–326. Springer, 2016.

[GWC19] Ariel Gabizon, Zachary J Williamson, and Oana Ciobotaru. PlonK: Permutations over Lagrange-bases for oecumenical noninteractive arguments of knowledge. *IACR Cryptol. ePrint Arch.*, 2019:953, 2019.

[KB16] Taechan Kim and Razvan Barbulescu. Extended tower number field sieve: A new complexity for the medium prime case. In *Annual International Cryptology Conference*, pages 543–571. Springer, 2016.

[Knu97] Donald E Knuth. *The Art of Programming, vol. 2 (3rd ed.), Seminumerical algorithms*. Addison Wesley Longman, 1997.

[KZG10] Aniket Kate, Gregory M Zaverucha, and Ian Goldberg. Constant-size commitments to polynomials and their applications. In *International conference on the theory and application of cryptology and information security*, pages 177–194. Springer, 2010.

[LFG23] Guiwen Luo, Shihui Fu, and Guang Gong. Speeding up multi-scalar multiplication over fixed points towards efficient zkSNARKs. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 358–380, 2023.

[LG23] Guiwen Luo and Guang Gong. Fast computation of multi-scalar multiplication for pairing-based zkSNARK applications. In *2023 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*, pages 1–5, 2023.

[LL94] Chae Hoon Lim and Pil Joong Lee. More flexible exponentiation with precomputation. In *Advances in Cryptology – CRYPTO 94*, pages 95–107. Springer, 1994.

[Mat82] David W Matula. Basic digit sets for radix representation. *Journal of the ACM (JACM)*, 29(4):1131–1143, 1982.

[MBKM19] Mary Maller, Sean Bowe, Markulf Kohlweiss, and Sarah Meiklejohn. Sonic: Zero-knowledge SNARKs from linear-size universal and updatable structured reference strings. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 2111–2128, 2019.

[Men09] Alfred Menezes. An introduction to pairing-based cryptography. *Recent trends in cryptography*, 477:47–65, 2009.

[Mil04] Victor S Miller. The Weil pairing, and its efficient calculation. *Journal of cryptology*, 17(4):235–261, 2004.

[Mon85] Peter L Montgomery. Modular multiplication without trial division. *Mathematics of computation*, 44(170):519–521, 1985.

[Mon87] Peter L Montgomery. Speeding the Pollard and elliptic curve methods of factorization. *Mathematics of computation*, 48(177):243–264, 1987.

[Mon92] Peter L Montgomery. Evaluating recurrences of form $x_{m+n} = f(x_m, x_n, x_{m-n})$ via Lucas chains, 1983. https://cr.yp.to/bib/1992/montgomery-lucas.pdf, 1992.

[MVOV18] Alfred J Menezes, Paul C Van Oorschot, and Scott A Vanstone. *Handbook of applied cryptography*. CRC press, 2018.

[PHGR13] Bryan Parno, Jon Howell, Craig Gentry, and Mariana Raykova. Pinocchio: Nearly practical verifiable computation. In *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE, May 2013. Best Paper Award.

[Pip76] Nicholas Pippenger. On the evaluation of powers and related problems. In *17th Annual Symposium on Foundations of Computer Science (sfcs 1976)*, pages 258–263. IEEE Computer Society, 1976.

[Pol78] John M Pollard. Monte Carlo methods for index computation. *Mathematics of computation*, 32(143):918–924, 1978.

[Rao15] Srinivasa Rao Subramanya Rao. A note on Schoenmakers algorithm for multi exponentiation. In *2015 12th International Joint Conference on e-Business and Telecommunications (ICETE)*, volume 4, pages 384–391. IEEE, 2015.

[Sco07] Michael Scott. Implementing cryptographic pairings. *Lecture Notes in Computer Science*, 4575:177, 2007.

[Sco15] Michael Scott. Missing a trick: Karatsuba revisited. *IACR Cryptol. ePrint Arch.*, 2015:1247, 2015.

[Sco17a] Michael Scott. The MIRACL core cryptographic library, 2017.

[Sco17b] Michael Scott. MIRACL cryptographic SDK, 2017.

[Sco19] Michael Scott. Pairing implementation revisited. *IACR Cryptol. ePrint Arch.*, 2019:77, 2019.

[Sil09] Joseph H Silverman. *The arithmetic of elliptic curves*, volume 106. Springer, 2009.

[SIM12] Vorapong Suppakitpaisarn, Hiroshi Imai, and Edahiro Masato. Fastest multi-scalar multiplication based on optimal double-base chains. In *World Congress on Internet Security (WorldCIS-2012)*, pages 93–98. IEEE, 2012.

[ST92] Joseph H Silverman and John Torrence Tate. *Rational points on elliptic curves*, volume 9. Springer, 1992.

[Str64] Ernst G Straus. Addition chains of vectors (problem 5125). *American Mathematical Monthly*, 70(806-808):16, 1964.

[TS10] Naoki Tanaka and Taiichi Saito. On the q-strong Diffie-Hellman problem. *IACR Cryptol. ePrint Arch.*, 2010:215, 2010.

[YWLT13] Wei Yu, Kunpeng Wang, Bao Li, and Song Tian. Joint triple-base number system for multi-scalar multiplication. In *International Conference on Information Security Practice and Experience*, pages 160–173. Springer, 2013.

[Zca] Zcash: Privacy-protecting digital currency. https://z.cash/.