TRADE-OFF ANALYSIS FOR GENERIC-POINT PARALLEL ELLIPTIC CURVE SCALAR MULTIPLICATION

FARIS FAWZAN AL-OTAIBI,
Umm Al-Qura University, **Makkah,**
Saudi Arabia

Submitted to the Faculty of the
College of Computer Science and Information System of the Umm AlQura University
in partial fulfilment of the requirements for the
Degree of MASTER OF SCIENCE

Shaaban 1440H — May 2019G

TRADE-OFF ANALYSIS FOR GENERIC-POINT PARALLEL ELLIPTIC CURVE SCALAR MULTIPLICATION

Signature of Author		....
Committee Member		Signature and Date
Prof Turki Al-Somani	(Chairman)	....
Prof Adnan Gutub	(Member)	....
Dr Fahd Aldosari	(Member)	....

ACKNOWLEDGEMENTS

All praise is due to Allah (Glorified and Exalted is He): Without His boundless favours and blessings, none of this work could exist. Then, I would like to express my special thanks and gratitude to my mentor, Prof Turki Al-Somani for all of his support and guidance in completing this thesis despite his busy schedule.

Furthermore, I would like to thank my family and friends, who have believed in me and given me their continued support in accomplishing this thesis. Finally, I would like to acknowledge Prof Adnan Gutub and Dr Fahd Aldosari, who have continuously helped me excute this thesis and improve my results.

DEDICATION

I dedicate this thesis to the precious people who have meant and continue to mean so much to me. Although some are no longer in this world, their memories remain to guide my life. First, I dedicate this thesis to my parents, who loved me unconditionally and fostered my love of science. Second, I dedicate this thesis to my beloved wife, who showed me love, patience and constant encouragement throughout my thesis. Last but not least I dedicate this thesis to to my sisters and brother, who also demonstrated their continuous support of this endeavour.

ABSTRACT

Full Name : Faris Fawzan Al-Otaibi
Thesis Title : TRADE-OFF ANALYSIS FOR
GENERIC-POINT PARALLEL ELLIPTIC CURVE SCALAR MULTIPLICATION
Major Field : Information Security
Date of Degree : May2019

Several methods have been proposed to accelerategeneric-point elliptic curve parallel scalar multiplication, including pre- com-putation-based methods and postcomputation-based methods. The methods proposed in the literature use key partitioning and process the key partitions via parallel processors. However, the best number of key partitions that would yield the best performance has yet to be investigated. Accordingly, this thesis conducts a trade-off analysis of all methods with different key sizes, numbers of processors and numbers of requests for generic- point elliptic curve parallel scalar multiplication. Furthermore, it proposes a new method and tests against the others. This new method demonstrates the best execu-tion time in most cases.

ملخص الرسالة

الاسم الكامل:فارس فوزان العتيبي

عنوان الرسالة: مفاضلة تحليلية للنقطة العامة في الضرب القياسي للمنحنى الإهليجي

التخصص:امن المعلومات

تاريخ الدرجة العلمية:شعبان 1440هـ

تم اقتراح العديد من الاساليب لتسريع الضرب القياسي للنقطة العامة في المنحنى الإهليجي. هذه الاساليب تتضمن : اسلوب العمليات السابقة والعمليات اللاحقة. الاساليب المقترحة في الابحاث السابقة تعتمد على تقسيم مفتاح التشفير لاجزاء والتعامل معاه بشكل متوازي. ورغماً عن ذلك لم نصل الى افضل عدد من الأجزاء الذي يقودنا الى اداء افضل. وفقاً لذلك، في هذا البحث اظهرنا تحليل مفصل لكل الاساليب مع احجام مفاتيح مختلفة وعدد معالجات مختلفة وعدد من الطلبات.علاوة على ذلك،تم اقتراح اسلوب جديد وتم اختباره مع بقية الاساليب الآخرى. الاسلوب الجديد المقترح أظهر نتائج افضل وقت معالجة في اغلب الحالات.

Contents

1 Introduction **1**

1.1 Motivation 1

1.2 The contributions of this thesis.. 2

1.3 Thesis Organisation.. 2

2 Background **3**

2.1 Finite-Field Arithmetic 3

2.2 Elliptic Curve Arithmetic 3

2.3 Scalar Multiplication 5

2.4 Summary.. 6

3 Review of literature **7**

3.1 Generic-Point Parallel Scalar Multiplication Method 7

3.2 Efficient and Scalable Postcomputation-Based Generic-Point Parallel Scalar Multiplication Method.. 9

3.3 Highly Efficient Generic-Point Parallel Scalar Multiplication Using Concurrent Precomputations 11

3.4 High-Performance Generic-Point Parallel Scalar Multiplication 13

3.5 Summary 16

4 Methodology **17**

4.1 Analysis 17

4.2 Implementation 19

4.3 Summary 20

5 Results **24**

5.1 Performance Analysis 24

5.1.1 Finding the best method 25

5.1.2 New method 26

5.2 Discussion of the Results 26

5.3 Summary 39

6 Conclusion **40**

List of Figures

2.1 Point addition 4

2.2 Point doubling 5

3.1 Example of the [10] method, where r = 2 and u = 4 10

3.2 Example of the [11] method, where r = 2 and u = 4 13

3.3 Example of the [12] method, where r = 2 and u = 4 14

4.1 Example of the proposed method, where r = 4 and u = 4 . . .18

5.1 Comparison of the methods in which key size = 160-bit and processors u = 2 27

5.2 Comparison of the methods in which key size = 160-bit and processors u = 4 28

5.3 Comparison of the methods in which key size = 160-bit and processors u = 8 29

5.4 Comparison of the methods in which key size = 192-bit and processors u = 2 30

5.5 Comparison of the methods in which key size = 192-bit and processors u = 4 31

5.6 Comparison of the methods in which key size = 192-bit and processors u = 8 32

5.7 Comparison of the methods in which key size = 256-bit and processors u = 2 33

5.8 Comparison of the methods in which key size = 256-bit and processors u = 4 34

5.9 Comparison of the methods in which key size = 256-bit and processors u = 8 35

5.10 Comparison of the methods in which key size = 512-bit and processors u = 2 36

5.11 Comparison of the methods in which key size = 512-bit and processors u = 4 37

5.12 Comparison of the methods in which key size = 512-bit and processors u = 8 38

List of Tables

4.1 Curve details of the 160-bit key size 20

4.2 Curve details of the 192-bit key size 21

4.3 Curve details of the 256-bit key size 22

4.4 Curve details of the 512-bit key size 23

5.1 Comparison of the scenarios in which key size = 160-bit and requests r = 8 25

5.2 Comparison of the scenarios in which key size = 160-bit and requests r = 16 25

5.3 Comparison of the scenarios in which key size = 160-bit and requests r = 32 26

5.4 Total DBLs of the new method and the previous methods [10– 12], with key size m = 160, 200, 256 and 512 and u = 8..........................26

5.5 Comparison of the methods in which key size = 160-bit and processors u = 2 27

5.6 Comparison of the methods in which key size = 160-bit and processors u = 4 28

5.7 Comparison of the methods in which key size = 160-bit and processors u = 8 29

5.8 Comparison of the methods in which key size = 192-bit and processors $u = 2$ 30

5.9 Comparison of the methods in which key size = 192-bit and processors $u = 4$ 31

5.10 Comparison of the methods in which key size = 192-bit and processors $u = 8$ 32

5.11 Comparison of the methods in which key size = 256-bit and processors $u = 2$ 33

5.12 Comparison of the methods in which key size = 256-bit and processors $u = 4$ 34

5.13 Comparison of the methods in which key size = 256-bit and processors $u = 8$ 35

5.14 Comparison of the methods in which key size = 512-bit and processors $u = 2$ 36

5.15 Comparison of the methods in which key size = 512-bit and processors $u = 4$ 37

5.16 Comparison of the methods in which key size = 512-bit and processors $u = 8$ 38

Chapter 1 Introduction

1.1 Motivation

Neal Koblitz and Victor Miller independently proposed elliptic curve cryptography (ECC) in 1985 [1, 2]. ECC is considered a serious alternative to many public key encryption algorithms. With key sizes of 128 to 256 bits, ECC offers security equal to that of RSA [3], which has key sizes of 1000 to 2000 bits [4, 5]. No significant weaknesses have yet been identified in the ECC algorithm, as it depends on the discrete logarithm problem over points on an elliptic curve [6]. The difficulty of the problem allows ECC key sizes to be reduced considerably [3]. This advantage of ECC has recently gained remarkable recognition and has been incorporated in many standards, such as IEEE, ANSI, NIST, SEC and WTLS.

Scalar multiplication is the basic operation of ECC. The scalar multiplication of a group of points on an elliptic curve is comparable to the exponentiation of a multiplicative group of integers modulo a fixed integer m. The scalar multiplication operation is denoted as kP, where k is an integer and P is a point on the elliptic curve. The kP operation represents the addition of k copies of point P. Scalar multiplication is then conducted according to a series of point doubling and point addition operations of the point P, which depend on the bit sequence representing the scalar multiplier k. Several scalar multiplication methods have been proposed [6].

Identifying efficient scalar multiplication methods for high-performance

end servers is crucial. Sequential scalar multiplication methods are too slow to meet the demands of the increasing number of customers for such servers. Scalar multiplication methods that can be parallelised are often used for high-speed implementations [7–12].

1.2 The contributions of this thesis

- We analysed all possible scenarios which may accelerate the scalar multiplication.
- We proposed a new method that shows the best result when compared with the others.
- We implemented the previous methods and the new using C++.
- We have analysed and discussed the results of the execution time for each method by comparing it with the others.

1.3 Thesis Organisation

Chapter 2 briefly addresses finite-field arithmetic and elliptic curves in general. Chapter 3 illustrates the work related to this thesis. Chapter 4 explains the methodology of this thesis and the proposed method. Chapter 5 presents and discusses the results of all methods tested in different cases. Finally, Chapter 6 concludes the thesis and discusses future work.

Chapter 2

Background

2.1 Finite-Field Arithmetic

A finite field contains only a finite number of elements in abstract algebra and is essential for many things, including cryptography[13–15]. The field represents a set of elements F with two operations: addition " + " and multiplication "*". Finite fields can also be called Galois fields, in honor of Evariste Galois. Galois fields are denoted by $GF(q)$. A Galois field of order $q = p^m$ exists in any prime p and positive integer m.

2.2 Elliptic Curve Arithmetic

The elliptic curve E appears over the finite field $GF(p)$ and is defined by the factors $a, b \in GF(p)$ with $p > 3$. It consists of a set of points $P = (x, y)$, where $x, y \in GF(p)$, and satisfies the equation of the elliptic curve (Equation 2.1) along with an additive of the group point O, known as the infinity point[1].

$$y^2 = x^3 + ax + b \tag{2.1}$$

where $a, b \in GF(p)$ and $4a^2 + 27b^2 \neq 0 \mod p$.

Hasse's theorem [14] defines the number of points n on an elliptic curve over a finite field $GF(q)$. A group operation is a set of discretepoints on an elliptic curve that form an abelian group. The elliptic curve has two group operations: point addition (ADD) and point doubling (DBL).Elliptic curve point addition is defined as stated in the 'chord-tangent process'. It is explained below.

Let P and Q be two separate points on E defined over $GF(p)$ and $Q \neq -P$. Q should not be the additive inverse of P . The additive inverse of point $P = (x, y) \in E$, over $GF(p)$, is point $-P = (x, -y)$, which is the inverse of point P on the y-axis with respect to the x-axis on E. The sum of the two points P and Q is point R, which is presented as follows: $R = P + Q$, where R is the additive inverse of S. S is the third point on E interrupted by the straight line through points P and Q. The point addition (ADD) is shown in 2.1.

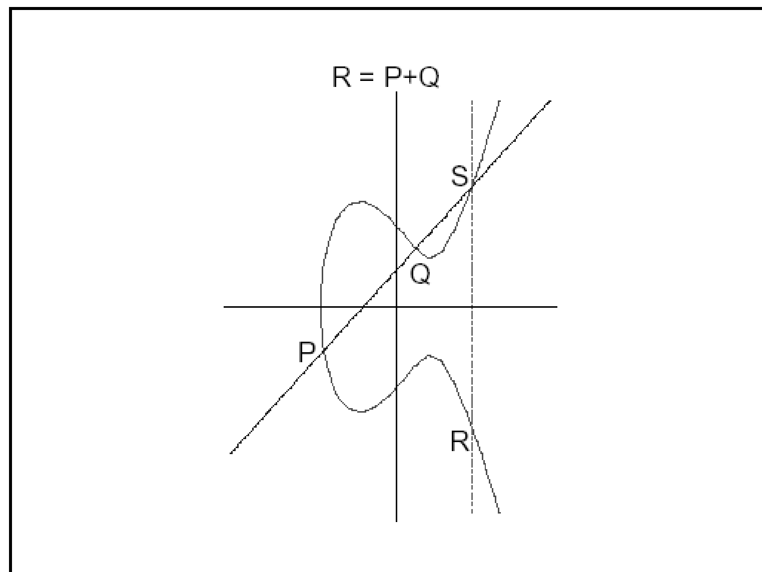

Figure 2.1: Point addition

Point doubling (DBL) involves adding P and Q when $P = Q$ and $P \neq -P$ and is represented as $R = 2P$, where R is the additive inverse of S. S is a point on E interrupted by the straight line tangent to the curve of point P and is illustrated in 2.2.

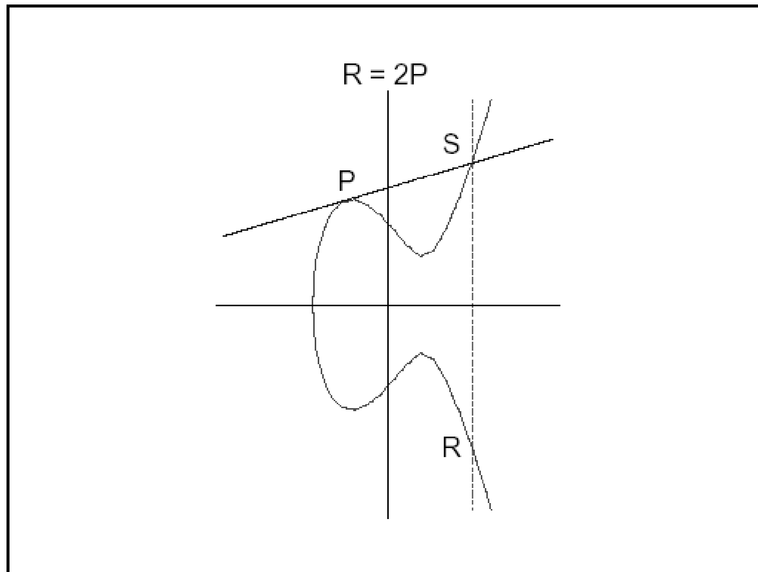

Figure 2.2: Point doubling

2.3 Scalar Multiplication

Scalar multiplication is the essential operation of ECC. The scalar multiplication of a group of points on an elliptic curve is similar to the exponentiation of a multiplicative set of integers modulo a fixed integer m. The scalar multiplication operation is denoted as kP , where k is an integer and P is a point on the elliptic curve. The kP operation represents the addition of k copiesof point P. kP can be computed using the binary method, which is thus called the double-and-add method, depending on the binary expression of the multiplier k. Using the binary method, computing kP is described below.

Let $k = (k_{(m-1)}, ..., k_0)$ represent the binary of k, where $k_{(m-1)}$ is the most significant bit of k. The multiplier k can be written asfollows:

$$k = \sum_{0 \le i < m} k_i 2^i = k_{m-1} 2^{m-1} + ... + k_1 2 + k_0 \quad (2.2)$$

k can be rewritten using the Horner expansion, as follows:

$$k = (...((k_{m-1} 2 + k_{m-2}) 2 + ... + k_1) 2 + k_0) \quad (2.3)$$

Accordingly, kP can be expressed as follows:

$$kP = 2(...2(2k_{m-1}P + k_{m-2}P) + ... + k_1 P) + k_0 P \quad (2.4)$$

The average time complexity of the binary method is expressed as follows:

$$\text{Time complexity } = \left((m)P_{DBL} + \left(\frac{m}{2}\right)P_{ADD}\right) \tag{2.5}$$

The binary method algorithm 1 is shown below.

Algorithm 1 Binary Method

Input: P, k

1: $Q \leftarrow P$
2: **for** $i \leftarrow (m-2)$ Down To 0 **do**
3: $\quad Q \leftarrow 2Q$
4: $\quad$ **if** $k_i = 1$ **then**
5: $\quad\quad Q \leftarrow Q + P$
6: $\quad$ **end if**
7: $\quad$ Output Q
8: **end for**

The binary scalar multiplication algorithm is the most straight- forward scalar multiplication algorithm. It examines the bits of the scalar multiplier k. If the checked bit $k_i = 0$, it only executes point doubling. How- ever, if the checked bit $k_i = 1$, it executes both point doubling and addition. The binary method needs m point doublings and an average of $m/2$ point additions.

2.4 Summary

Finite-field and elliptic curve arithmetic are reviewed here. The main focus of this chapter is scalar multiplication, which is an essential operation of ECC. The scalar multiplication of a group of points on an elliptic curve is similar to the exponentiation of a multiplicative set of integers modulo a fixed integer m. The scalar multiplication operation is known as kP , where k is an integer and P is a point on the elliptic curve. Furthermore, the kP operation represents the addition of k copies of point P .

Chapter 3

Review of literature

Many papers [7–12] have proposed methods to speed up generic-point elliptic curve parallel scalar multiplication. The major problem researchers face is converting the sequential steps in parallel. Early precomputation methods[7] can accelerate scalar multiplication by converting some steps in parallel, significantly improving scalar multiplication. Such methods do improve performance, but sequential steps are required before complete the process parallel. However, postcomputation methods [8–10] remove the sequential steps by moving them to the end of the computation. More recent methods are explained in detail below.

3.1 Generic-Point Parallel Scalar Multiplication Method

A method that replaces the sequential steps in precomputations to make them parallelisable has been proposed [8]. Postcomputations split the multi- plier k into u partitions, which can be processed into u processors in parallel using the binary method. Then, the postcomputations are distributed on $u-1$ processors to be performed in parallel. Finally, the key partitions and the postcomputations are assimilated to obtain kP. Algorithm 2 is performed efficiently to compute kP in parallel without precomputations.

Algorithm 2 Generic-Point Parallel Scalar Multiplication

1: **Input:** P, k
2: **By filling** k **with zeros if necessary, write** k = $(k^{(u-1)}||k^{(u-2)}||...||k^{(0)})$, *where* $k^{(i)}$) **is a partition of length** $m_{(i)}$ **bits.**
3: **Initialisation:** $Q \leftarrow P, R \leftarrow O$
4: **Parallel Scalar Multiplication:**
5: **for** $i \leftarrow 0$ To u-1 **do** in parallel
6: $Q \leftarrow$ Binary method $(k^{(i)}, P_i)$
7: **if** $i > 0$ **then**
8: **for** c = 0 to $((\sum_{0 \leq j < i} m_j) - 1)$ **do**
9: $Q \leftarrow 2Q$
10: **end for**
11: **end if**
12: $R \leftarrow R + Q$
13: **end for**
14: Output R

In Algorithm 2, the multiplier k splits into u partitions as different sizes to solve the equations. The algorithm 2 provides a balanced number of point operations for different partitions. The partitioning occurs in Step 2. Parallel scalar multiplications start at Step 4. Different processors are used to process each partition independently. Step 7 is used to check whether it is partitioned $k^{(0)}$, as it does not require any postcomputations and the other partitions do require postcomputations. Finally, each partition assembles its resulting point at the accumulation point R (Step 12), which requires $u - 1$ extra point additions.

The limitation of this paper [8] is focused on a single request and the problem statement is looked for multiple requests. Thus, this paper is not suitable to analyse with the other methods. Moreover, except for the first part, each part of the request will make an additional add operation for the last point with the other point from the other processor and this additional operation will make some of part of request will hold to getting the result from the other.

3.2 Efficient and Scalable Postcomputation-Based Generic-Point Parallel Scalar Multiplication Method

An efficient mapping technique that uses a set of requests (two requests) to perform together has been proposed [10]. The mapping technique splits each request into u processors and each part of the request is the same size. Thus, the execution time of each part differs due to the extra operation that may be increased due to the number of parts. The technique uses clever mapping to perform the fastest part from the first request in combination with the slowest part from the second request. This method proposes the following algorithm 3, which can be performed efficiently for the parallel computation of kP.

Algorithm 3 Generic-Point Parallel Scalar Multiplication

1: **Input:** P, k
2: **By filling** k **with zeros if necessary, write** k = $(k^{(u-1)}||k^{(u-2)}||...||k^{(0)})$, *where* $k^{(i)}$ **is a partition of length** $v = \left\lceil \frac{m}{u} \right\rceil$ **bits.**
3: **Initialisation:** $Q \leftarrow P, R \leftarrow 0$
4: **Key Partitions Associated with Cryptoprocessors:**
5: **for** $i = 0$ to $u - 1$ **do**
6: $(k^{(i)}, Cryptoprocessor_{(i)})$
7: **end for**
8: **Parallel Scalar Multiplication:**
9: **for** $i \leftarrow 0$ To u - 1 **do** in parallel
10: $Q \leftarrow$ Binary method $(k^{(i)}, P_i)$
11: **if** $i > 0$ **then**
12: **for** $c = 0$ to iv **do**
13: $Q \leftarrow 2Q$
14: **end for**
15: **end if**
16: $R \leftarrow R + Q$
17: **end for**
18: Output R

In Algorithm 3, the multiplier k splits into u partitions of equal size to solve the equations. For a particular k and P , each key partition is

mapped to a certain cryptoprocessor in Step 4. Parallel scalar multiplications start at Step 8. It uses different processors to process each partition independently. Step 11 is used to check whether it is partition $k^{(0)}$ because it does not require any postcomputations and the other do partitions require postcomputations. Finally, each partition assembles its resulting point at the accumulation point R (Step 18), which requires $u - 1$ extra point additions.

Figure 3.1 explains how the proposed method works with the number of requests $r = 2$ and processors $u = 4$. The main operation of all of the requests is less than the extra operation. The idea of the method [10] is to try to remove the overhead of two requests by mapping the most partition work from the second request with the least partition work from the first partition, as shown in the Figure 3.1. By considering the mapping, the two requests should end at the same time.

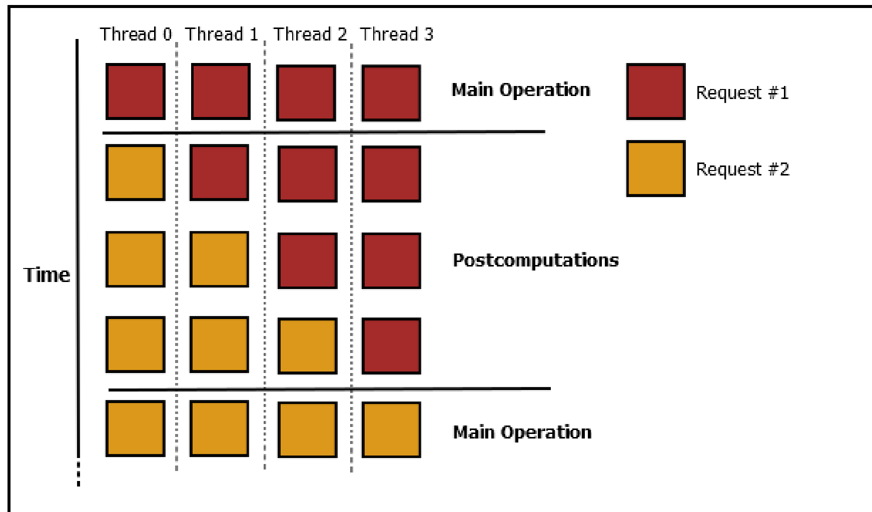

Figure 3.1: Example of the [10] method, where $r = 2$ and $u = 4$

This proposed method is mapping a set of two requests and could be compared with the other method. Besides, each part of the request will make an additional add operation for the last point with the other point from the other processor except for the first. Further, this operation will make some part of request in-hold to getting the result from the other and the whole set of two requests will run two new add operations at most. Thus, the number of extra operations is equal to the number of requests.

3.3 Highly Efficient Generic-Point Parallel Scalar Multiplication Using Concurrent Precomputations

The main point here [11] is to implement u sequential precomputations of u generic points using u processors concurrently by mapping each generic point to an individual processor. The precomputed points are then used to perform the parallel scalar multiplication of each point of the u generic points. Each multiplier k of the u multipliers is partitioned into a number of equally sized partitions (v) that can be processed in parallel by u processors. The points that result from processing these key partitions are assimilated at the end to produce kP . The number of available processors limits the number of partitions of each k. For a particular P and k, each partition is associated with a precomputed point to maintain its significance. For u partitions, $(u - 1)$ precomputed points are required for each k. Precomputed points can be computed simply by a sequence of doubling operations of the base point P.

Algorithm 4 shows the pseudocode of the proposed method in [11]. For a specific P and k, each partition $k^{(i)}$ is associated with a particular precomputed point P_i to maintain the significance of each partition (Step 20). For a particular P and k, parallel scalar multiplications start at Step 26. Each partition can be independently processed in an individual processor. The points from each execution of the binary scalar multiplication method [5] are accumulated in point R (Step 30), which requires $(u - 1)$ the addition of extra points.

Figure 3.2 shows an example of the proposed method [14] with processors u = 4 and number of requests r = 4. The extra operations are in the beginning, as shown in Figure 3.2. It is thus referred to as a precomputation method. The main operation of each request ends at the same time. The goal of this thesis is to make a computation for a set of requests r that are equal to the number of the processors u, which perform together. Each request can then be divided equally to the number of the processors u and perform together until it ends.

When the elliptic curve point is fixed the starting by the precomputation for each request will be a waste of time and resources because each processor will perform the same result. Besides, each processor, except the

Algorithm 4 Concurrent Precomputation Method

1: **Input:** P [0], P [1], ..., P [$u-1$], k[0], k[1], ..., k[$u-1$]
2: **By filling the ks with** $(uv-m)$ **zeros if required, write each** $k = (k^{(u-1)}||k^{(u-2)}||...||k^{(0)})$, *where each* $k^{(i)}$ **is a partition of length** $(v = [m/u])$ **bits**
3: **Initialisation:**
4: **for** i = 0to $u-1$ **do**
5: $Q[i] \leftarrow P[i]$
6: $R[i] \leftarrow 0$
7: **end for**
8: **Implementation of Concurrent Precomputations of u Points for Each P:**
9: **for** i = 0 to $u-1$ **do** (in parallel)
10: $P_0[i] \leftarrow Q[i]$
11: **end for**
12: **for** w = 0to $u-1$ **do** (in parallel)
13: **for** i = 0 to $u-1$ **do**
14: **for** j = 0to $u-1$ **do**
15: $Q[w] \leftarrow 2Q[w]$
16: **end for**
17: $P_i[w] \leftarrow Q[w]$
18: **end for**
19: **end for**
20: **Key Partitions Associated with Precomputed Points:**
21: **for** i = 0to $u-1$ **do**
22: **for** j = 0to $u-1$ **do**
23: $(k[i]^{(j)}, P_j[i])$
24: **end for**
25: **end for**
26: **Scalar Multiplication:**
27: **for** i = 0to $u-1$ **do**
28: **for** j = 0to $u-1$ **do** (in parallel)
29: $Q[i]$ The Binary Method $(k[i]^{(j)}, P_j[i])$
30: $R[i] \leftarrow R[i] + Q[i]$
31: **end for**
32: Output R[i]
33: **end for**

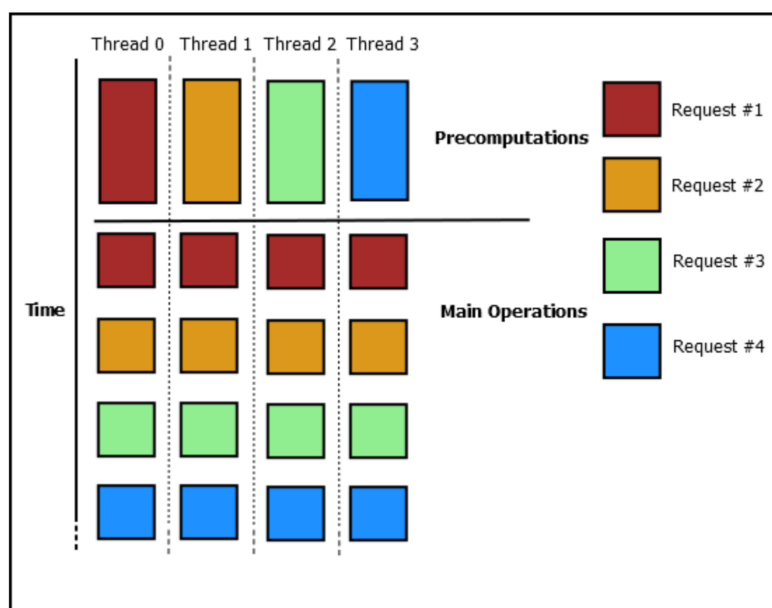

Figure 3.2: Example of the [11] method, where r = 2 and u = 4

first one, will make an additional add operation for the last point with the other point from the neighbor processor to accumulate the final result. Further, this operation will make some processor u hold to receive the result from the other and the complete time of an extra operation of each request will similar to $u - 1$ additional operations. Hence, the method will perform a great time when the number of processors u are small.

3.4 High-Performance Generic-PointParallel Scalar Multiplication

The main idea [12] here is to use u parallel processors to perform $u/2$ scalar multiplications. In the proposed method, two parallel processors are used to perform one scalar multiplication kP using the least-to-most version of the binary method (Algorithm 5) with a buffer between the two processors (as shown in Figure 3.3). The first processor only performs DBL operations, producing a point to the buffer only if the inspected bit of the multiplier k is 1. In contrast, the second processor performs an ADD operation between the resulting point of the previous addition and a point from the buffer until the buffer is empty, which means that kP is computed. It is assumed

here that average point operations are equal to (m) DBL $+(m/2)$ ADD. It is also assumed that point ADD requires twice the required time for point DBL. Accordingly, the buffer between the two processors is almost empty, indicating that both processors are fully used.

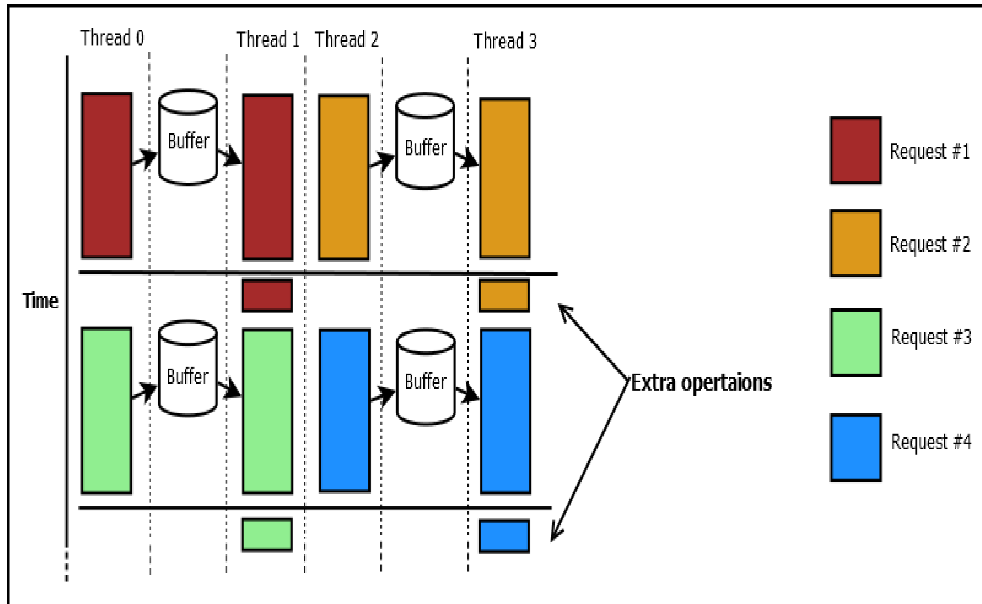

Figure 3.3: Example of the [12] method, where $r = 2$ and $u = 4$

An example of the proposed architecture is shown in Figure 3.3, with $u = 4$ processors and $r = 4$ number of requests. Each scalar multiplier requires two processors and a buffer residing between these two processors. Each processor requires a field arithmetic unit and either a point DBL unit $(Processor i)$ or a point ADD unit $(Processor_{i+1})$. The buffer size depends on m, and $\lfloor log2(m) \rfloor$ is used for the buffer size. The pseudocode of the proposed method is shown in Algorithm 5.

The resulting points that meet the condition on Step 11 add onto the buffer according to Step 12. Then, while the buffer is not empty, the add operation should work by adding the point from the buffer to the last point. The final results are obtained at Step 21.

The disadvantage of this paper [12] when the processor that responsible for add operation get the last element from the buffer the double processor will be in-hold for a time equivalent to an additional add operation. Besides, each request require two processors and buffer at least to performs

Algorithm 5 Proposed Method

1: **Input:** $P[0], \ldots, P[u/2-1], k[0], \ldots, k[u/2-1]$
2: **Initialisation:**
3: **for** i = 0 to $u/2-1$ **do**
4: $R[i] \leftarrow P[i]$
5: $Q[i] \leftarrow 0$
6: **end for**
7: **Scalar Multiplication:**
8: **for** i = 0 To $u/2-1$ **do** in parallel
9: **for** j = 0 To $m-1$ **do**
10: $R[i] \leftarrow DBL(R[i])$
11: **if** $k[i]_j = 1$ **then**
12: $buffer_i \leftarrow R[i]$
13: **end if**
14: **end for**
15: **while** $buffer_i$ is not empty **do**
16: **if** $buffer_i$ is not empty **then**
17: $Q[i] \leftarrow ADD(Point \in buffer_i, Q[i])$
18: **end if**
19: **end while**
20: **end for**
21: Output R[i]

the whole request. The time of an additional operation is not considered as cost when the number of requests is small. Thus, the cost will not be enormous if the more requests are performed concurrently due to available processors because the needed of two processors for each request. Figure3.3 shows an example of four requests performs to four processors with a great time and two extra operations at most, but when the method performs the same amount of requests on two processors the extra operation will befour. Thus the method will show a great time when the number of processors is large.

3.5 Summary

In this chapter, related work is discussed. In [8], the sequential step that should be first in the parallelisable postcomputations is replaced. Postcomputations split the key into partitions and can then process the partitions into u processors in parallel. An efficient mapping technique that uses a set of requests (two requests) to perform together has been proposed [10]. The mapping technique splits each request into u processors. Each part of the request is the same size. Thus, the execution time of each part differs due to the extra operation and may increase due to the number of parts. The technique uses clever mapping to perform the fastest part from the first request in combination with the slowest part from the second request.

Sequential precomputation of a number of generic points can be implemented using the same number of processors concurrently by mapping each generic point to an individual processor [11]. The precomputed points can then be used to accomplish the parallel scalar multiplication of each point of the generic points. Each multiplier can be partitioned into a number of equally sized partitions that can be processed in parallel by a number of processors. In one study, u parallel processors are used to perform $u/2$ scalar multiplications [12]. The proposed method uses two parallel processors to perform one scalar multiplication kP using the least-to-most version of the binary method with a buffer between the two processors. The first processor only performs DBL operations. The buffer expects a point if the inspected bit of the multiplier k is 1 in the first processor. The second processor performs an ADD operation between the resulting point of the previous addition and a point from the buffer until the buffer is empty.

Chapter 4

Methodology

Various methods have been examined to determine the best formulation to accelerate scalar multiplication. In so doing, the number of processors, re- quests and partitions of the key have been measured to obtain the best method. The next two sections of this chapter show these measurements and the method used to analyse and compare the formulations to other methods.

4.1 Analysis

The main goal of this thesis is to analyse schemes to accelerate scalar multi- plication. The number of requests r is used as a significant factor toanalyse the methods. The number of processors u and partitions of the key size v are measured with a different number of requests r. Each possible case is analysed and measured by the average time performance. During theanalysis, a new method is discovered. The average time performance is the time required for the point doubling (DBL) and addition (ADD) operations in all cases. The required computation time for point addition (DBL) is twice that required for point doubling (ADD).

The new method demonstrates the best average time compared with the others and uses only two of the factors as essential parts. These factors are the number of requests and processors. The new method is com- pared with the previous methods to determine which demonstrates thebest time performance. By working concurrently, each request must finish the

rule sequentially in one processor in the new method. Furthermore, the multiple processors allow the new method to simultaneously finish a number of requests r equal to the number of processors u. The time complexity of the new method can be measured as follows:

$$Time\ Complexity = \lceil r/u \rceil \ ((m)\ P_DBL + \ (m/2)\ P_ADD\ )$$

The newly proposed method suggests that the number of requests r should be considered as a critical factor. It uses u parallel processors to per- form u scalar multiplications, which means that each request is distributed to a unique processor to work sequentially. A number of requests equal to the number of available processors finish at the same time, and all processors become available to handle the next set of requests. Thus, each set of requests, which can each be called a stage, works sequentially in different processors and ends together.

Figure 4.1 shows an example of the proposed method when processors u = 4 and number of requests r = 4. Each processor will hold only one request till finish sequentially and concurrently with the other requests. The whole requests are ended at the same time and without need to submit any data to another pro- cessor. The pseudocode of the proposed method is shown in Algorithm 6.

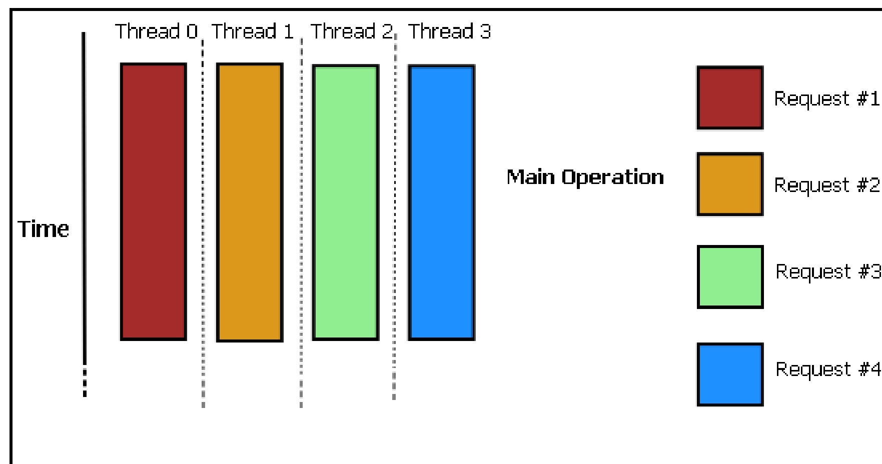

Figure 4.1: Example of the proposed method, where r = 4 and u = 4

Algorithm 6 Proposed Method

1: **Input:** $P[0], P[1], ..., P[u-1], k[0], k[1], ..., k[u-1]$.
2: **Initialisation:**
3: **for** $i = 0$ to $u-1$ **do**
4: $Q[i] \leftarrow P[i]$
5: $R[i] \leftarrow 0$
6: **end for**
7: **Scalar Multiplication:**
8: **for** $i = 0$ to $u-1$ **do** (in parallel)
9: **for** $j = 0$ to $m-1$ **do**
10: **if** $k[i]_j = 1$ **then**
11: $R[i] \leftarrow R[i] + Q[i]$
12: **end if**
13: $Q[i] \leftarrow DBL(Q[i])$
14: **end for**
15: Output R[i]
16: **end for**

Algorithm 6 takes u requests, translating to u points and u keys. Each request i has a base point $P[i]$ and key $k[i]$. The algorithm is initialised in Step 2. The requests are distributed to work in parallel in Step 8. Each re- quest is worked separately. Finally, the results of each request i are obtained in Step 15.

4.2 Implementation

The main goal of this thesis is to conduct a trade-off analysis of the previous methods of accelerating the scalar multiplication of generic-point parallel elliptic curves. To achieve this, most of these methods are implemented using C++. They are then tested with a sequential method to validate the results. The technique of the methods depends on how they separate the requests on multiple processors. Moreover, the new method discovered during the analysis process helps speed up scalar multiplication.

Key Size	160-bit
Curve ID	brainpoolP160r1
P	E95E4A5F737059DC60DFC7AD95B3D8139515620F
A	340E7BE2A280EB74E2BE61BADA745D97E8F7C300
B	1E589A8595423412134FAA2DBDEC95C8D8675E58
xP_0	BED5AF16EA3F6A4F62938C4631EB5AF7BDBCDBC3
yP_0	1667CB477A1A8EC338F94741669C976316DA6321
q	E95E4A5F737059DC60DF5991D45029409E60FC09

Table 4.1: Curve details of the 160-bit key size

4.3 Summary

A new method is proposed based on a trade-off analysis of the previous methods. To ensure that the results of each method are implemented using C++, both the new and previous methods are validated, and their results are tested with the approved sequential method. The methods are tested with different curves and key sizes. Tables 4.1, 4.2, 4.3 and 4.4 show the curves [16] and their data, which are used with the methods. In the next chapter, the results of each method in different cases (in terms of processors u and number of requests r) are presented and discussed.

Key Size	192-bit
Curve ID	brainpoolP192r1
P	C302F41D932A36CDA7A3463093D18DB78FCE476D E1A86297
A	6A91174076B1E0E19C39C031FE8685C1CAE040E5 C69A28EF
B	469A28EF7C28CCA3DC721D044F4496BCCA7EF414 6FBF25C9
xP_0	C0A0647EAAB6A48753B033C56CB0F0900A2F5C48 53375FD6
yP_0	14B690866ABD5BB88B5F4828C1490002E6773FA2 FA299B8F
q	C302F41D932A36CDA7A3462F9E9E916B5BE8F102 9AC4ACC1

Table 4.2: Curve details of the 192-bit key size

Key Size	256-bit
Curve ID	brainpoolP256r1
P	A9FB57DBA1EEA9BC3E660A909D838D726E3BF623D 52620282013481D1F6E5377
A	7D5A0975FC2C3057EEF67530417AFFE7FB8055C12 6DC5C6CE94A4B44F330B5D9
B	26DC5C6CE94A4B44F330B5D9BBD77CBF958416295 CF7E1CE6BCCDC18FF8C07B6
xP_0	8BD2AEB9CB7E57CB2C4B482FFC81B7AFB9DE27E1 E3BD23C23A4453BD9ACE3262
yP_0	547EF835C3DAC4FD97F8461A14611DC9C2774513 2DED8E545C1D54C72F046997
q	A9FB57DBA1EEA9BC3E660A909D838D718C397AA3B 561A6F7901E0E82974856A7

Table 4.3: Curve details of the 256-bit key size

Key Size	512-bit
Curve ID	brainpoolP512r1
P	AADD9DB8DBE9C48B3FD4E6AE33C9FC07CB308DB3B 3C9D20ED6639CCA703308717D4D9B009BC66842AECDA 12AE6A380E62881FF2F2D82C68528AA6056583A48F3
A	7830A3318B603B89E2327145AC234CC594CBDD8D3DF9 1610A83441CAEA9863BC2DED5D5AA8253AA10A2EF1 C98B9AC8B57F1117A72BF2C7B9E7C1AC4D77FC94CA
B	3DF91610A83441CAEA9863BC2DED5D5AA8253AA10A2 EF1C98B9AC8B57F1117A72BF2C7B9E7C1AC4D77FC94 CADC083E67984050B75EBAE5DD2809BD638016F723
xP_0	81AEE4BDD82ED9645A21322E9C4C6A9385ED9F70B5D 916C1B43B62EEF4D0098EFF3B1F78E2D0D48D50D168 7B93B97D5F7C6D5047406A5E688B352209BCB9F822
yP_0	7DDE385D566332ECC0EABFA9CF7822FDF209F70024A 57B1AA000C55B881F8111B2DCDE494A5F485E5BCA4B D88A2763AED1CA2B2FA8F0540678CD1E0F3AD80892
q	AADD9DB8DBE9C48B3FD4E6AE33C9FC07CB308DB3 B3C9D20ED6639CCA70330870553E5C414CA9261941866 1197FAC10471DB1D381085DDADDB58796829CA90069

Table 4.4: Curve details of the 512-bit key size

Chapter 5

Results

In the previous chapter, the calculation of the time performance of each method is explained. In this chapter, the performance analysis of all different cases that may help accelerate scalar multiplication is shown. Furthermore, the performance of the new method is compared with that of the other methods [10–12] and is discussed in the first section. In the second section, the suggested method and the other methods [10–12], implemented using C++,
are analysed on Lenovo Y50-70 with an Intel ® Core ™ an i7-4720HQ CPU @ 2.60GHz processor, 16-GB RAM and a 64-bit operating system.

The resulting execution times of scalar multiplication for each method are compared using the sequential binary method. The results depend on the different key sizes (i.e., 160, 190, 256 and 512) and number of processors (i.e., 2, 4 and 8), which are used to determine the best execution time. Each key size is analysed according to various request situations (i.e., 4, 8, 16, 23 and 64). The results are recorded for all analysed situations. Every situation is discussed in the following subsections, with results tables and charts included for reference.

5.1 Performance Analysis

The new method is observed during the analysis of the possible methods. The analysis and all related results are discussed in the first section. The comparisons of the new and previous methods [10–12] are shown in the second section.

u	v = 1	v = 2	v = 4	v = 8
1	**2,560**	3,200	4,480	7,040
2	**1,280**	1,600	2,240	3,520
4	**640**	800	1,120	1,760
8	**320**	400	560	880

Table 5.1: Comparison of the scenarios in which key size = 160-bit and requests r = 8

u	v = 1	v = 2	v = 4	v = 8
1	**5,120**	6,400	8,960	14,080
2	**2,560**	3,200	4,480	7,040
4	**1,280**	1,600	2,240	3,520
8	**640**	800	1,120	1,760

Table 5.2: Comparison of the scenarios in which key size = 160-bit and requests r = 16

5.1.1 Finding the best method

Tables 5.1, 5.2 and 5.3 show the different scenarios in which the time performance of each method is tested, where every table has a unique number of requests r and a 160-bit key size. The number of processors u, requests r and partitions of key size v are the significant factors. Table 5.1 shows the results of different cases when the number of requests r = 8. In this case, the best scenario is when processors u = 8 and partitions of key size v = 1. Tables **5.2** and 5.3 show the same results. Finally, the analyses of the result led to the separation of the key size may made an overhead on the system. In the case of parallel processors, scalar multiplication should separate the many requests r into equal-size processors u, where u = r and works sequentially.

u	v = 1	v = 2	v = 4	v = 8
1	**10,240**	12,800	17,920	28,160
2	**5,120**	6,400	8,960	14,080
4	**2,560**	3,200	4,480	7,040
8	**1,280**	1,600	2,240	3,520

Table 5.3: Comparison of the scenarios in which key size = 160-bit and requests r = 32

M	r = 8				r = 16				r = 32			
	[10]	[11]	[12]	New	[10]	[11]	[12]	New	[10]	[11]	[12]	New
128	760	480	256	256	1,520	960	512	512	3,040	1,920	1,024	1,02
160	920	572	320	320	1,840	1,144	640	640	3,680	2,288	1,280	1,28
200	1,120	687	400	400	2,240	1,374	800	800	4,480	2,748	1,600	1,60
256	1,400	848	512	512	2,800	1,696	1,024	1,024	5,600	3,392	2,048	2,04

Table 5.4: Total DBLs of the new method and the previous methods [10–12], with key size m = 160, 200, 256 and 512 and u =8

5.1.2 New method

According to [9], the eight processors can achieve the best performance when using the postcomputation method of [10]. Therefore, the methods in [10– 12] are compared with the new method when the processors u = 8 and for several key sizes m. The required computation time for point addition is twice that required for point doubling. The comparison results of the methods are shown in Table 5.4 as Total DBLs for requests $r \geq 8$ and up to 32. Finally, the results of the new method are the same as those in [12] with a different algorithm and no need for a buffer.

5.2 Discussion of the Results

The 160-bit key size is tested with different numbers of processors. First, the methods are tested for processors u = 2. The results of execution time are shown in Table 5.5. Excellent results compared with the rest of the methods are provided by [11] when processors u = 2. The new method demonstrates better results when requests $r < 16$. In this case, the [12] and sequential

r	Sequential	[10]	[11]	[12]	New
4	0.125	0.073	0.064	0.093	0.069
8	0.244	0.127	0.099	0.196	0.081
16	0.495	0.262	0.186	0.303	0.233
32	0.983	0.536	0.369	0.833	0.497
64	1.972	1.321	0.731	1.56	1.02

Table 5.5: Comparison of the methods in which key size = 160-bit and processors $u = 2$

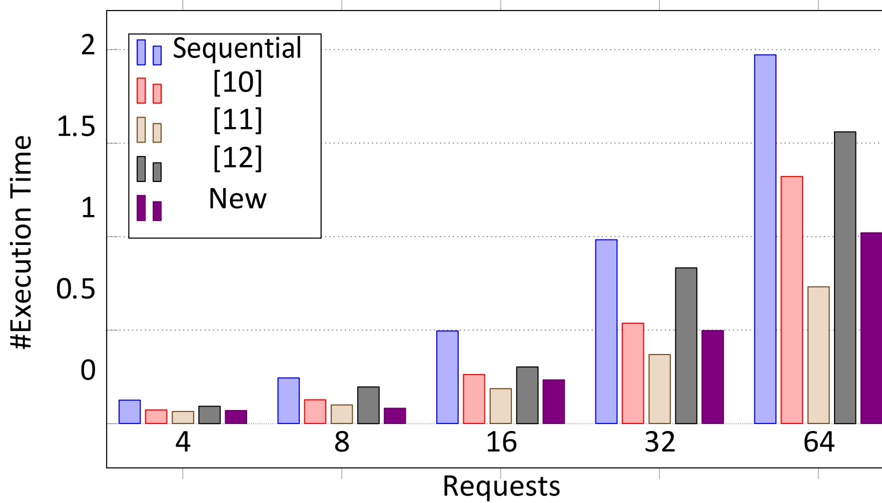

Figure 5.1: Comparison of the methods in which key size = 160-bit and processors $u = 2$

methods demonstrate the worst results. Results that are quite similar to the normal case of scalar multiplication are provided by [12] when processors $u = 2$. This makes it a poor choice for small values of u. Figure 5.1 shows the results in Table 5.5 in diagram form.

Table 5.6 shows the results of the 160-bit key size when processors $u = 4$. The new method demonstrates the best execution time. Further- more, when the requests r increase, the difference in the execution time is remarkable. This difference is shown in Figure 5.2. [11] and [12] are quite similar, as shown in Figure 5.2. The rest consume considerable time. [11] yields good results with the processors $u = 4$ and $u = 2$.

r	Sequential	[10]	[11]	[12]	New
4	0.129	0.089	0.063	0.068	0.05
8	0.254	0.18	0.12	0.12	0.063
16	0.498	0.337	0.247	0.27	0.164
32	0.999	0.653	0.472	0.499	0.396
64	1.99	1.317	0.918	0.984	0.76

Table 5.6: Comparison of the methods in which key size = 160-bit and processors u = 4

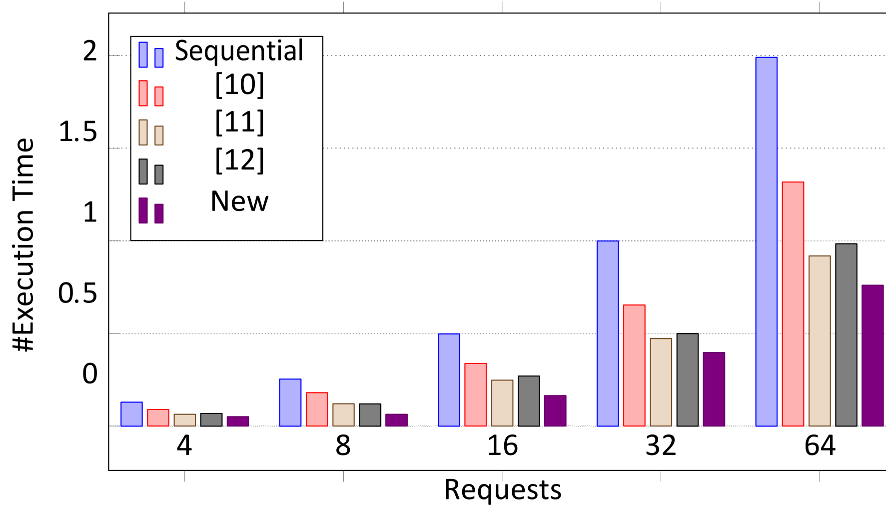

Figure 5.2: Comparison of the methods in which key size = 160-bit and processors u = 4

r	Sequential	[10]	[11]	[12]	New
4	0.129	0.131	0.064	0.046	0.053
8	0.25	0.226	0.107	0.097	0.076
16	0.493	0.432	0.213	0.183	0.141
32	1.009	0.911	0.402	0.354	0.284
64	1.978	2.005	0.906	0.671	0.55

Table 5.7: Comparison of the methods in which key size = 160-bit and processors u = 8

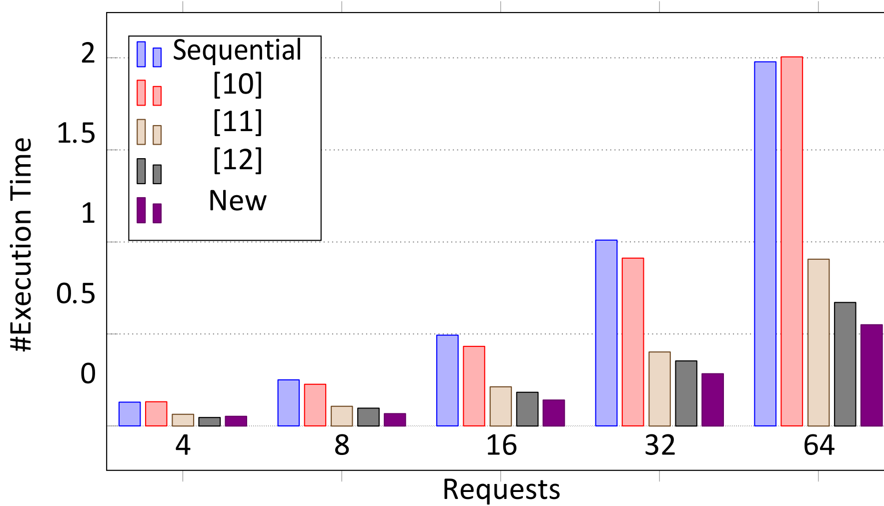

Figure 5.3: Comparison of the methods in which key size = 160-bit and processors u = 8

The results of the 160-bit key size when processors u = 8 are shown in Table 5.7. The new method demonstrates the best execution time when the requests $r > 4$. The [11], [12] and new methods are so comparable when $r < 64$. The rest consume considerable time and are too similar, as shown in Figure 5.3. [11] consumes the same amount of time when processors u = 8, u = 4 u = 2, but it is not the best choice when u = 8. The new method demonstrates the best execution time and [10] demonstrates the worst when $u > 2$. Figures 5.1, 5.2 and 5.3 show that the [12] method improves when the processors u increase.

r	Sequential	[10]	[11]	[12]	New
4	0.186	0.141	0.12	0.158	0.094
8	0.363	0.257	0.211	0.284	0.134
16	0.736	0.511	0.408	0.626	0.311
32	1.457	1.02	0.817	1.19	0.7
64	2.922	2.053	1.646	2.366	1.531

Table 5.8: Comparison of the methods in which key size = 192-bit and processors u = 2

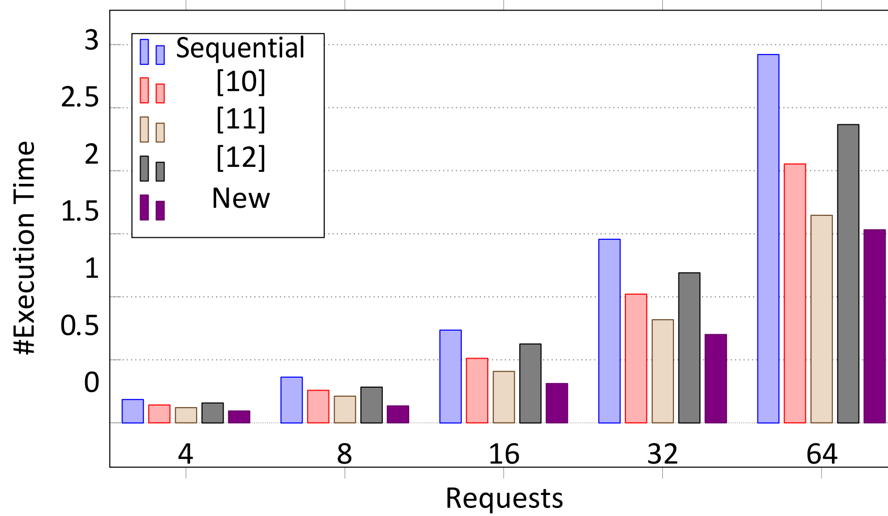

Figure 5.4: Comparison of the methods in which key size = 192-bit and processors u = 2

Here, the results of the methods with the 192-bit key size are shown. Each number of processors is tested with the mentioned key size. The results for execution time are shown in Table 5.8. The new method demonstrates great results when the processors u = 2, particularly in comparison with the rest of the methods. [11] shows excellent results for the 160-bit key size when the processors u = 2. However, with the 192-bit key size, it yields the second-best results. The [12] and sequential methods are the worst methods in this case. [12] yields results quite similar to the general case of scalar multiplication when the processors u = 2, making it an inappropriate choice for small values of u. Figure 5.4 shows the results in Table 5.8 in diagram form.

r	Sequential	[10]	[11]	[12]	New
4	0.182	0.147	0.102	0.09	0.069
8	0.362	0.243	0.189	0.166	0.091
16	0.734	0.509	0.377	0.401	0.247
32	1.45	0.964	0.713	0.721	0.527
64	2.915	1.953	1.375	1.54	1.14

Table 5.9: Comparison of the methods in which key size = 192-bit and processors $u = 4$

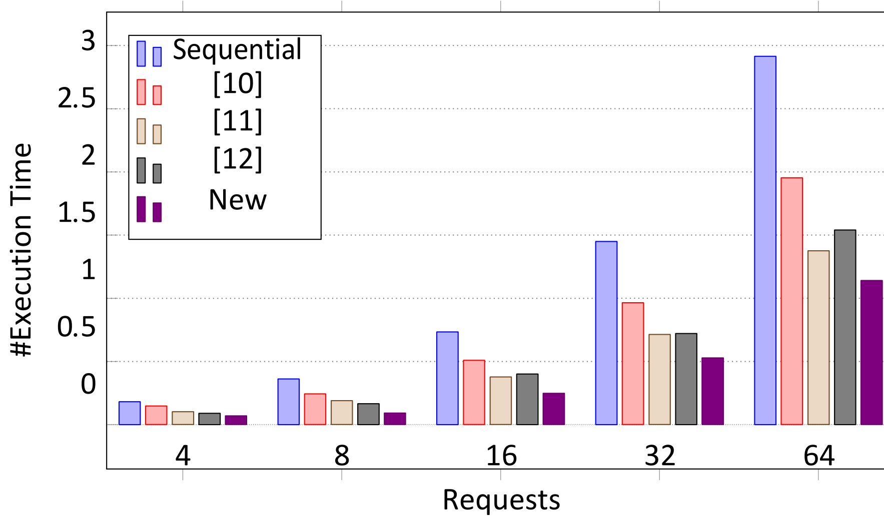

Figure 5.5: Comparison of the methods in which key size = 192-bit and processors $u = 4$

The results of the 192-bit key size when the number of processors $u = 4$ are shown in Table 5.9. The new method demonstrates the best execution time compared with the other methods. The difference in the execution time is notable when the requests r increase. [11] and [12] are quite similar when $r < 64$, as shown in Figure 5.5. Furthermore, [12] consumes considerable time. [11] yields good results when the processors $u = 4$ and $u = 2$ regardless of whether the key size is 192-bit or 160-bit. The remaining methods yield the worst results for this case and the previous cases, as shown in Figure 5.5.

r	Sequential	[10]	[11]	[12]	New
4	0.204	0.172	0.096	0.091	0.092
8	0.367	0.315	0.158	0.12	0.092
16	0.725	0.619	0.304	0.321	0.192
32	1.451	1.277	0.568	0.602	0.38
64	2.903	2.546	1.148	1.144	0.747

Table 5.10: Comparison of the methods in which key size = 192-bit and processors u = 8

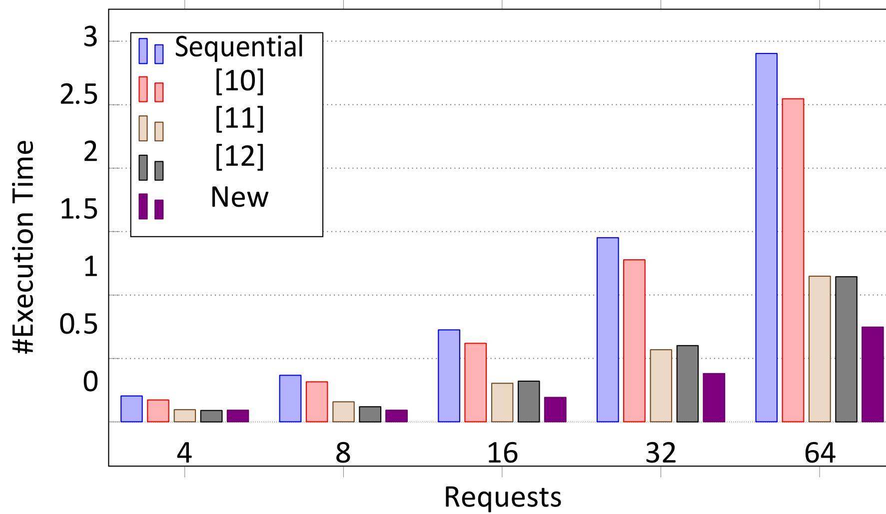

Figure 5.6: Comparison of the methods in which key size = 192-bit and processors u = 8

Table 5.10 shows the results of all of the methods tested with pro- cessors u = 8. According to Table 5.10, the [11, 12] and new methods are somewhat similar when the number of requests r = 4. The new method offers the best execution time when the requests $r > 4$. [11] and [12] are very similar. The rest consume considerable time and are too similar, as shown in Figure 5.6. The new method demonstrates the best execution time, and the [10] method demonstrates the worst when $u > 2$. Tables 5.8, 5.9 and 5.10 show that the [12] method is enhanced when the processors u increase.

Here, the results of each method using a 256-bit key size are presented. Every situation for the key is discussed in consideration of the number

r	Sequential	[10]	[11]	[12]	New
4	0.345	0.246	0.19	0.25	0.176
8	0.687	0.447	0.339	0.496	0.263
16	1.372	0.913	0.654	1.035	0.658
32	2.75	1.849	1.305	2.185	1.273
64	5.48	3.599	2.557	4.147	3.086

Table 5.11: Comparison of the methods in which key size = 256-bit and processors u = 2

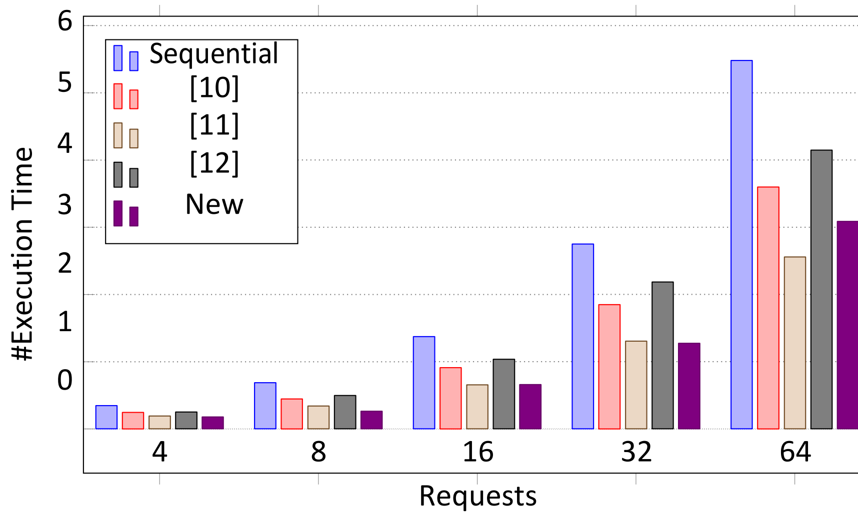

Figure 5.7: Comparison of the methods in which key size = 256-bit and processors u = 2

of processors. The results are shown in tables and charts. The execution time of each method is shown in Table 5.11. The new method and [11] are too similar when the number of requests $r < 64$. However, when $r = 64$, [11] is the best. The [12] and sequential methods are the worst choices in this case. Generally, the new method is the best according to Table 5.11.

Table 5.12 shows the results of the 256-bit key size when processors u = 4. The new method provides the best execution time relative to the other methods. Furthermore, when the requests r increase, the difference in the execution time is significant, as shown in Figure 5.8. [11] yields good results when the number of requests $r > 8$, according to Table 5.12, in comparison

r	Sequential	[10]	[11]	[12]	New
4	0.341	0.222	0.193	0.168	0.12
8	0.676	0.47	0.368	0.291	0.161
16	1.357	0.859	0.644	0.687	0.467
32	2.712	1.703	1.271	1.355	0.966
64	5.415	3.385	2.557	2.685	1.974

Table 5.12: Comparison of the methods in which key size = 256-bit and processors u = 4

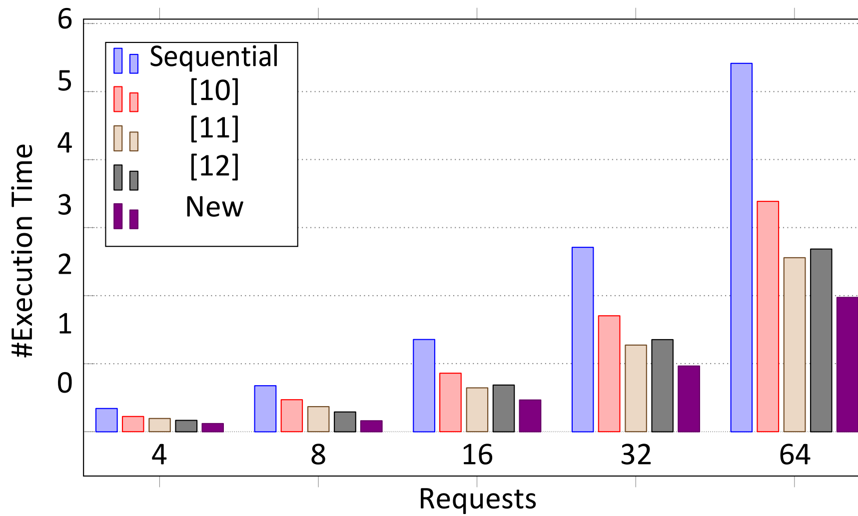

Figure 5.8: Comparison of the methods in which key size = 256-bit and processors u = 4

with [12], which yields respectable results when $r \leq 8$. Generally, the new method yields the best results when the processors u = 4 and u = 2.

Based on Table 5.13, the new method demonstrates the best execution time when processors u = 8. [11] and [12] are so comparable when $r < 64$. The remaining methods consume considerable time and are too similar, as shown in Figure 5.9. The execution time of [11] improves when the processors u = 8. Contrarily, when u = 2 and u = 4, the results are similar. The new method shows the best execution time in all cases. Furthermore, the [12] method improves when the processors u increase, according to Tables 5.11, 5.12 and 5.13.

r	Sequential	[10]	[11]	[12]	New
4	0.346	0.288	0.163	0.152	0.117
8	0.688	0.584	0.273	0.235	0.153
16	1.375	1.086	0.508	0.491	0.306
32	2.753	2.198	1.02	0.981	0.655
64	5.498	4.304	2.12	1.898	1.277

Table 5.13: Comparison of the methods in which key size = 256-bit and processors u = 8

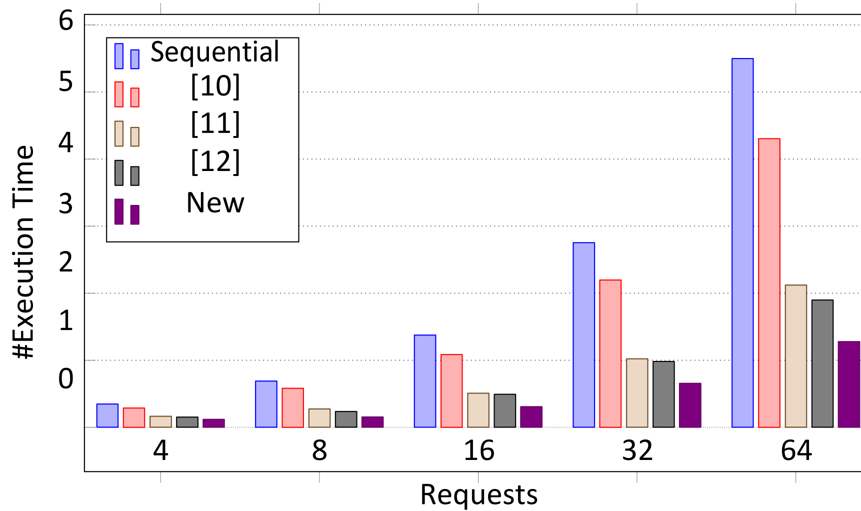

Figure 5.9: Comparison of the methods in which key size = 256-bit and processors u = 8

r	Sequential	[10]	[11]	[12]	New
4	1.591	1.109	0.969	1.245	0.846
8	3.187	2.267	1.673	2.354	1.008
16	6.371	4.331	3.235	5.067	3.032
32	12.794	8.688	6.309	9.903	6.064
64	25.481	17.332	12.401	19.603	12.378

Table 5.14: Comparison of the methods in which key size = 512-bit and processors u = 2

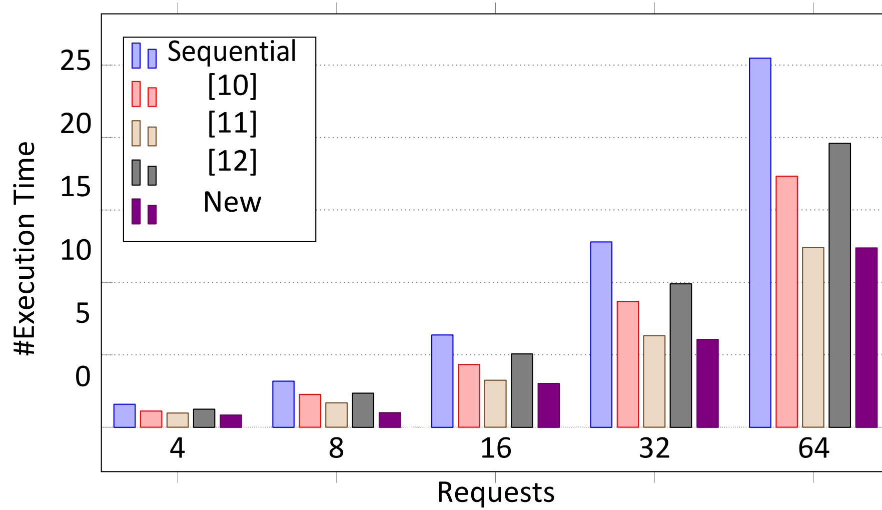

Figure 5.10: Comparison of the methods in which key size = 512-bit and processors u = 2

Here, the 512-bit key size is discussed. Each number of processors is tested with the declared key size. The 512-bit key size is tested on processors u = 2. Table 5.14 shows the results of the execution time. The new method demonstrates the best results according to Table 5.14, whereas the [11] yields good results for the 512-bit key size when the processors u = 2. Moreover, the [12] and sequential methods demonstrate the worst results in this case. [12] yields extremely poor results when the processors u = 2, which makes it an unwise choice when the values of u are small. [10] consistentlydemonstrates good results when the processors u = 2 for different key sizes, as shown in Figures 5.1, 5.4, 5.7 and 5.10.

r	Sequential	[10]	[11]	[12]	New
4	1.608	1.053	0.709	0.774	0.522
8	3.218	1.888	1.456	1.303	0.742
16	6.431	3.619	2.797	3.044	2.026
32	12.857	7.159	5.437	6.26	4.455
64	25.724	15.573	10.624	12.329	7.761

Table 5.15: Comparison of the methods in which key size = 512-bit and processors u = 4

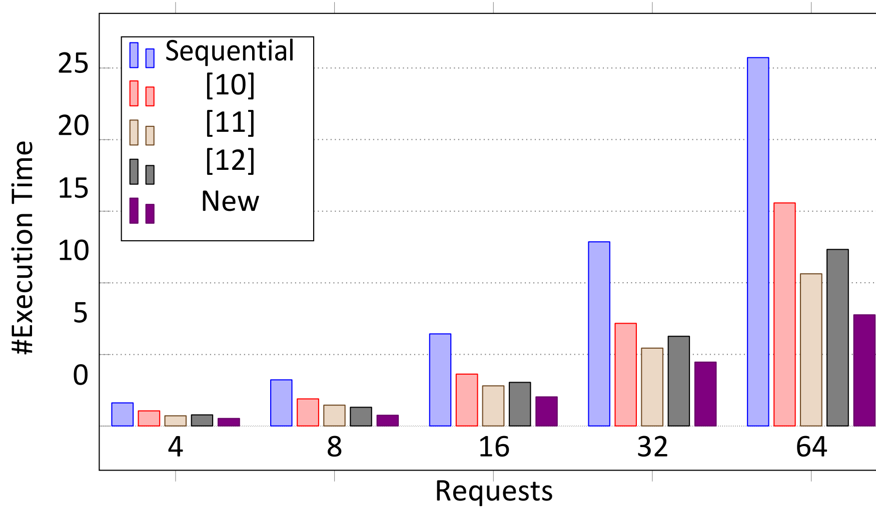

Figure 5.11: Comparison of the methods in which key size = 512-bit and processors u = 4

The results for the 512-bit key size when number of processors u = 4 are shown in Table 5.15. The new method demonstrates the best execution time in comparison with the other methods. [11] and [12] are quite similar when $r < 16$, as shown in Figure 5.11. Furthermore, [11] consumes less time than [12]. [11] yields good results when the processors u = 4 and u = 2 for different key sizes. The remaining methods yield the worst results for this case, as shown in Figure 5.11.

According to Table 5.16, which presents the results when the processors u = 8, the [12] and new methods are similar when the number of requests r = 4. The new method offers the best execution time when the

r	Sequential	[10]	[11]	[12]	New
4	1.577	1.311	0.699	0.557	0.599
8	3.162	2.552	1.094	1.005	0.734
16	6.311	4.984	2.255	2.226	1.498
32	12.622	9.938	4.316	4.767	2.902
64	25.229	19.875	8.645	8.864	5.81

Table 5.16: Comparison of the methods in which key size = 512-bit and processors u = 8

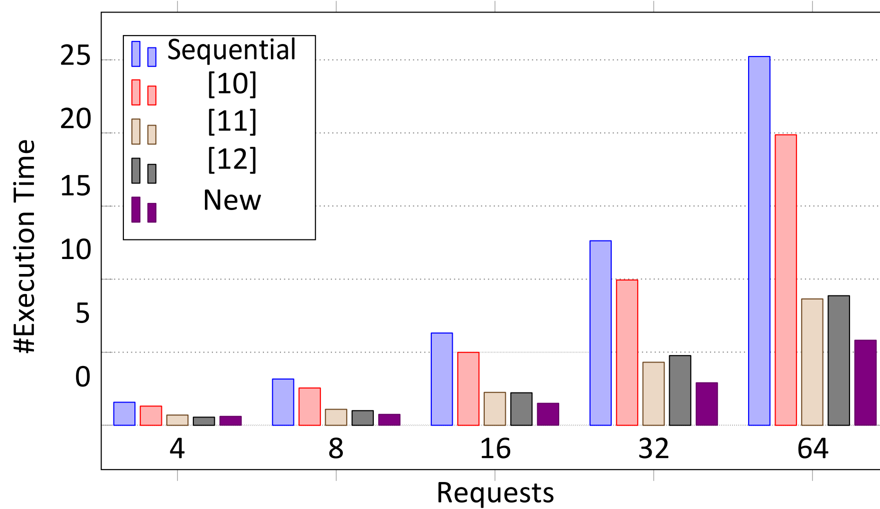

Figure 5.12: Comparison of the methods in which key size = 512-bit and processors u = 8

requests $r > 4$. [11] and [12] are very similar. The rest consume consider- ably more time than the others, as shown in Figure 5.12. The new method demonstrates the best execution time in every case, and [10] demonstrates the worst when $u > 2$. The [11] and [12] methods are enhanced when the processors u increase.

5.3 Summary

Various scenarios are analysed to determine the best formulations to ac- celerate scalar multiplication. The time performance is calculated for each scenario. In the first section of this chapter, the results of finding the best method are shown, and this method is compared with the previous methods [10–12]. A new method with a good execution time is discovered that demon- strates the same results as the method of [12] with a different algorithm, which does not require a buffer. In the second section, the implementation
of all methods is analysed on a Lenovo Y50-70 with an Intel® Core™, an i7-4720HQ CPU @ 2.60GHz processor, a 16-GB RAM and a 64-bit operating system.

The results of each method, tested in one environment, are recorded in Tables 5.5 to 5.16. Furthermore, the tests are conducted in various situ- ations (when processors u = 2, 4 and 8) and for various key sizes (160-bit, 192-bit, 256-bit and 512-bit). The results show how each method performs in different cases. Generally, the new method demonstrates the best execution time overall and [11] demonstrates the second-best execution time. [12] is the worst method when processors $u < 8$. Furthermore, [10] yields similar results with different processors u. However, when the number of requests r increases, the efficiency of the new method is remarkable. Figures 5.1 to 5.12 represent the results in diagram form for clarity. The next and last chapter of this thesis concludes by addressing its accomplishments and discussing future work.

Chapter 6

Conclusion

Precomputation-based methods and postcomputation-based methods have been proposed to accelerate generic-point scalar multiplication, but they have yet to be tested on regular devices. In this thesis, a new method is proposed during the trade-off analysis of the previous methods. Furthermore, the previous methods and the new method are implemented using C++. Each method is tested in one environment and in various cases. The results of each case are recorded in tables and charts. Overall, the new method demonstrates the best execution time. [11] is the second-best method based on the results. Furthermore, [12] is the worst method when the number of processors u is small.

Future work that analyses other new methods or methods thatare not addressed in this thesis may be interesting. Furthermore, such work may implement the method proposed in this thesis on FPGA. It may also be worthwhile to conduct the same analysis on FPGA using different key sizes, numbers of processors and numbers of requests.

Bibliography

[1]N. Koblitz, *Mathematics of computation* **1987**, *48*, 203–209.

[2]V. S. Miller in Conference on the theory and application of crypto- graphic techniques, Springer, **1985**, pp. 417–426.

[3]R. L. Rivest, A. Shamir, L. Adleman, *Communications of the ACM* **1978**, *21*, 120–126.

[4]A. J. Menezes, *Elliptic Curve Public Key Cryptosystems*, *Vol. 234*, Springer Science & Business Media, **1993**.

[5]D. Hankerson, A. J. Menezes, S. Vanstone, *Computing Reviews* **2005**, *46*, 13.

[6]C. Paar, J. Pelzl, *Understanding cryptography: a textbook forstudents and practitioners*, Springer Science & Business Media, **2009**.

[7]E. F. Brickell, D. M. Gordon, K. S. McCurley, D. B. Wilson in Work- shop on the Theory and Application of of Cryptographic Techniques, Springer, **1992**, pp. 200–207.

[8]T. F. Al-Somani, M. K. Ibrahim, *IEICE Electronics Express* **2009**, *6*, 1732–1736.

[9]T. F. Al-Somani, *Global Journal of Computer Science and Technology* **2010**.

[10]T. F. Al-Somani, A. G. Fayoumi, M. K. Ibrahim, *IEICE Electronics Express* **2014**, 11–20140356.

[11]T. F. Al-Somani, *J. Appl. Sci* **2015**, *15*, 1261–1265.

[12]T. F. Al-Somani, *Arabian Journal for Science and Engineering* **2017**, *42*, 507–512.

[13]N. L. Biggs, Discrete Mathematics Oxford University Press, **1999**.

[14]R. J. McEliece, *Finite field for scientists and engineers*, Kluwer Aca- demic Publishers, **1987**.

[15]R. Lidl, H. Niederreiter, *Introduction to finite fields and theirapplications*, Cambridge university press, **1994**.

[16]D. Lochter, ECC brainpool standard curves and curve generation v. 1.0, **2005**.