EFFICIENT IMPLEMENTATION OF TMVP-BASED PRIME FIELD MULTIPLICATION AND ITS APPLICATIONS TO ECC

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF APPLIED MATHEMATICS
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

HALİL KEMAL TAŞKIN

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF DOCTOR OF PHILOSOPHY
IN
CRYPTOGRAPHY

FEBRUARY 2019

Approval of the thesis:

EFFICIENT IMPLEMENTATION OF TMVP-BASED PRIME FIELD MULTIPLICATION AND ITS APPLICATIONS TO ECC

submitted by **HALİL KEMAL TAŞKIN** in partial fulfillment of the requirements for the degree of **Doctor of Philosophy in Cryptography Department, Middle East Technical University** by,

Prof. Dr. Ömür Uğur
Director, Graduate School of **Applied Mathematics** ____________

Prof. Dr. Ferruh Özbudak
Head of Department, **Cryptography** ____________

Assoc. Prof. Dr. Murat Cenk
Supervisor, **Cryptography, METU** ____________

Examining Committee Members:

Prof. Dr. Ersan Akyıldız
Mathematics, METU ____________

Assoc. Prof. Dr. Murat Cenk
Cryptography, METU ____________

Assoc. Prof. Dr. Ali Doğanaksoy
Mathematics, METU ____________

Assoc. Prof. Dr. Sedat Akleylek
Computer Engineering, Ondokuz Mayıs University ____________

Assoc. Prof. Dr. Oğuz Yayla
Mathematics, Hacettepe University ____________

Date: ____________

ABSTRACT

EFFICIENT IMPLEMENTATION OF TMVP-BASED PRIME FIELD MULTIPLICATION AND ITS APPLICATIONS TO ECC

Taşkın, Halil Kemal
Ph.D., Department of Cryptography
Supervisor : Assoc. Prof. Dr. Murat Cenk

February 2019, 69 pages

The need for faster and practical cryptography is a research topic for decades. For elliptic curve cryptography, which is proposed independently by Koblitz and Miller in 1985 as a more efficient alternative to RSA, the applications of it in real life started after 2000s. Today, most of the popular applications and protocols like Whatsapp, Signal, iOS, Android, TLS, SSH, Bitcoin etc. make use of elliptic curve cryptography.

In this thesis, we present a new representation of finite field multiplication which is one of the basic building blocks for the ECC using Toeplitz matrix-vector product (TMVP) and discuss its arithmetic cost and comparison. In addition, we evaluate the delay complexity of the proposed algorithm when computations are performed using multi-core systems. We also describe how to choose proper prime fields that make use of Toeplitz matrices to get faster field arithmetic. Then, we give parameter choice details to select prime fields that support TMVP operations and propose some prime fields to work on. We propose a new multiplication algorithm over $\mathbb{F}_{2^{255}-19}$ where the de-facto standard Curve25519 algorithm is based on. The proposed algorithm for the underlying finite field multiplication exploits the TMVP and achieves salient results.

We also introduce the safe curve selection rationale and discuss about attacks on ECC. Next, we propose a new curve choice parameter and safe curve generation process. Finally, we introduce the Curve2663 and give details about its implementation and benchmark results and conclude the thesis.

Keywords: Toeplitz matrix-vector product, Elliptic curve cryptography, Polynomial multiplication, Finite field multiplication, Montgomery curve, Safe curves

ÖZ

TMVÇ TABANLI VERİMLİ ASAL CİSİM ÇARPMASI GERÇEKLEMESİ VE ELİPTİK EĞRİ KRİPTOGRAFİ'YE UYGULAMALARI

Taşkın, Halil Kemal
Doktora, Kriptografi Bölümü
Tez Yöneticisi : Doç. Dr. Murat Cenk

Şubat 2019, 69 sayfa

Daha hızlı ve pratik şifrelemeye duyulan ihtiyaç, onlarca yıldır bir araştırma konusudur. 1985 yılında Koblitz ve Miller tarafından bağımsız olarak RSA'ya daha etkin bir alternatif olarak önerilen eliptik eğri kriptografisinin gerçek hayattaki uygulamaları 2000'li yıllardan sonra başlamıştır. Günümüzde Whatsapp, Signal, iOS, Android, TLS, SSH, Bitcoin gibi popüler uygulamaların ve protokollerin çoğu eliptik eğri kriptosistemler (EEK) kullanmaktadırlar.

Bu tez çalışmasında, Toeplitz matris-vektör çarpımını (TMVÇ) kullanarak EEK'nin temel yapı taşlarından birisi olan sonlu cisim çarpımı için yeni bir gösterim sunuyoruz ve bu gösterimin aritmetik maliyetini ve karşılaştırmasını ele alıyoruz. Buna ek olarak, çok çekirdekli sistemler kullanılarak hesaplamalar yapıldığında algoritmamızın gecikme karmaşıklığını hesaplıyoruz. Ayrıca daha hızlı cisim aritmetiği elde etmek için Toeplitz matrislerini kullanabileceğimiz uygun asal cisimlerin nasıl seçileceğini de açıklıyoruz. Ardından, TMVÇ'yi destekleyen asal cisimlerin seçilmesinin ve üzerinde çalışılabilecek asal cisimlerin önerilmesi için parametre seçilmesinin detaylarını veriyoruz. Curve25519 algoritmasının üzerine inşaa edildiği $\mathbb{F}_{2^{255}-19}$ sonlu cismi üzerinde TMVÇ kullanan yeni bir çarpma algoritması gösterimi öneriyoruz. Önerdiğimiz bu algoritma dikkat çekici sonuçlar elde etmektedir.

Ayrıca güvenli eğri seçimi gerekçelerini ortaya koyup, EEK'ye yapılan ataklar hakkında bilgi veriyoruz. Daha sonra yeni bir eliptik eğri seçim parametresi ve güvenli eliptik eğri üretme sürecini öneriyoruz. Son olarak, Curve2663 eğrisini tanıtıyoruz, gerçeklemesi ve kıyaslama sonuçları hakkında ayrıntılı bilgi veriyoruz ve tezi sonuçlandırıyoruz.

Anahtar Kelimeler: Toeplitz matrisi vektör çarpımı, Eliptik eğri kriptografi, Polinom çarpımı, Sonlu cisim çarpımı, Montgomery eğrileri, Güvenli eğriler

To My Mother
(Annem'e)

ACKNOWLEDGMENTS

I would like to express my appreciation to my thesis supervisor Assoc. Prof. Dr. Murat Cenk for his patient guidance and valuable advices during the development and preparation of this thesis. His willingness to give his time and to share his experiences has brightened my path.

I wish to thank the members of my dissertation committee, Prof. Dr. Ersan Akyıldız, Assoc. Prof. Dr. Ali Doğanaksoy, Assoc. Prof. Dr. Sedat Akleylek and Assoc. Prof. Dr. Oğuz Yayla for generously offering their time and good will throughout the preparation and review of this thesis.

I owe my sincere thanks to Prof. Dr. Ali Aydın Selçuk, Dr. Mert Özarar, Assoc. Prof. Dr. Oğuz Yayla, Dr. Çağdaş Çalık and Dr. Cihangir Tezcan for their valueable suggestions, encouragement, guidance and support during my graduate life.

I would like to thank The Scientific and Technological Research Council of Turkey (TÜBİTAK) for supporting part of this research under the Grant No. 115R289.

I express my gratitude to Mehmet Özkan and Murat Demircioğlu for their unconditional friendship, support and patience throughout these years. I will always remember our discussions and projects we did with Mehmet.

I am also grateful to all my friends, especially, Ahmet S., Fuad H., Bartu Y., Elif Ç., Mansoor K., Emre A., Aykut B., Mustafa B. and Salim S. for their close friendship. I also would like to thank all my managers and colleagues in my professional work life for their support and motivation.

A special thanks go to my family, especially to my grandmother (Nene), my uncle Gökhanemmi, my cousins Eren Efem and Elif Vesile (latest and cutest member of our family), my elder cousin Zuhal and also all Semerci family members, especially, Eralp and İlay for being there and supporting me all the time.

I would like to acknowledge and extend my heartfelt gratitude to my mother Havva for supporting me at every turn and moment of my life. Without your support and encouragement, this wouldn't have been possible.

Lastly, my deepest gratitude goes to my grandfather Halil and my father Mustafa. Your ideas enlightened my way of thinking. It is my honor to have and complete your names as my name.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

$\mathtt{A}$	Single-precision word addition
$\mathtt{A_d}$	Double-precision word addition
ECC	Elliptic Curve Cryptography
ECDLP	Elliptic Curve Discrete Logarithm Problem
$\mathbb{F}_p$	Finite field of size p
$\mathtt{M}$	Single-precision word multiplication
$M_{n\times n}$	$n \times n$ Toeplitz Matrix
TMVP	Toeplitz Matrix-vector Product

CHAPTER 1

INTRODUCTION

Classical Diffie-Hellman methods are getting slower due to their key size. In the mean time, the applications of elliptic curve cryptography in real life started after 2000s [11], which led to faster asymmetric operations. Nowadays, Edwards [5] and Montgomery [23] curves are studied in detail, and several elliptic curves in these forms [3, 4, 6, 10, 14, 19] are published and deployed in the field. Thus, this research also focuses specifically on Montgomery curves.

The finite fields that elliptic curves are defined over can be categorized as binary extension fields and prime fields with primes bigger than 2. Due to the developments in the area [16, 34], the interest turned to prime fields over which new elliptic curves are mostly defined. This thesis focuses on prime fields, too.

NIST has several approved curves for ECC. But after the NIST's Dual_EC_DRBG incident [29] and IETF's request to CFRG for alternative elliptic curves [12], the search for new elliptic curves which can be generated in a transparent and verifiable way emerged. Looking for a proper elliptic curve on which a cryptosystem can be built is a trade off between being theoretically secure and being implemented efficiently in real life. This leads to the question if we can find an elliptic curve with the highest security and fastest implementation. To answer this question, there are lots of on going efforts and new elliptic curve proposals. The NIST curves are already efficient enough for this purpose. However, due to the reasons mentioned at the beginning, alternatives such as Curve25519 [4] are also widely deployed in the field and included in new standard drafts [28]. It became the de-facto standard for the industry and used in many popular applications such as Whatsapp, Signal, Threema, iOS, Android etc.[20], which makes it a gripping target to work on. NIST also announced that the upcoming draft of SP 800-186 will specify Curve25519. Additionally, its associated key agreement scheme, X25519, will be considered for inclusion in a subsequent revision to SP 800-56A [28].

The Curve25519 function is an $\mathbb{F}_p$-restricted $x-$coordinate only scalar multiplication on $E(\mathbb{F}_p)$, where p is the prime number $2^{255} - 19$ and E is the elliptic curve $y^2 = x^3 + 486662x^2 + x$. It is known for being faster than NIST Curves and makes use of Montgomery curves. The design choice is based on selection of primes as close as possible to a power of 2 to save time in field operations. There are various studies regarding the implementation for Curve25519, which mostly exploit specific CPU architectures with special instructions [13, 17, 24].

For the elliptic curve operations, the underlying finite field operations are crucial in terms of performance. Thus, improving complexity of multiplication is an important problem in computer algebra. In the literature, there are several methods and approaches dealing with optimizing the complexity of multiplication. This thesis also proposes and discusses some new complexity results based on Toeplitz matrix-vector multiplication a.k.a Toeplitz matrix-vector product (TMVP) in depth.

A measure of efficiency of polynomial multiplication is to count the number of coefficient multiplications required. If polynomials $f(x)$ and $g(x)$ both have degree n, then both have $n + 1$ coefficients, and each coefficient of $f(x)$ is multiplied by every coefficient of $g(x)$. Thus, multiplying two polynomials of degree n in the standard way requires $(n + 1)^2$ number multiplications. Therefore, complexity of polynomial multiplication is one of the major problems in computer algebra. If we can express the finite field multiplications as polynomial multiplications, we can use polynomial multiplication methods on finite fields. In general, these methods focus on integer arithmetic but it is possible to extend and improve these methods to finite fields. Besides, polynomial multiplications can also be expressed as matrix vector product.

For the underlying finite field multiplication operation, the schoolbook algorithm together with the refined Karatsuba algorithm leads to the best known results in elliptic curve based cryptography [4]. It is shown in [1] that using TMVP, new algorithms can be constructed with better complexities.

TMVP method also has advantages when the algorithm is implemented in multi-core CPUs thanks to its independent submatrix computations.

The preliminaries are presented in Chapter 2. The rest of the thesis is organized as follows:

In Chapter 3, we introduce the key parts of the TMVP and their arithmetic costs, then, propose a new finite field multiplication algorithm based on TMVP. We also choose and propose new finite fields and their representations that have Toeplitz matrix suitable field multiplication operations with better complexity compared to schoolbook algorithms. Moreover, to take advantage of this method for the Curve25519, we build a new representation which is in Toeplitz matrix form and get salient results in Chapter 4.

After prime field parts, in Chapter 5, we give details for choosing safe curves with respect to different criteria and attacks. We also propose a new criteria for categorizing curves. At the end, we define the safe elliptic curve selection algorithm and run it with `MAGMA` software [36]. We find and propose a new elliptic curve that meets the criteria we defined. After explaining its details, we conclude the thesis.

CHAPTER 2

PRELIMINARIES

In this chapter, we summarize the basics of finite fields and elliptic curves. First, we briefly introduce the definition of finite fields with its operations. Next, we give details for elliptic curves and its applications to cryptography.

2.1 Finite Fields

A field consists of a set $\mathbb{F}$ together with two operations, namely, addition $(+)$ and multiplication $(\cdot)$. It satisfies:

1. $(\mathbb{F}, +)$ is an additive abelian group with identity 0.
2. $(\mathbb{F}\backslash\{0\}, \cdot)$ is a multiplicative abelian group with identity 1.
3. The distributive property of multiplication; $(x + y) \cdot z = x \cdot z + y \cdot z$ for all $x, y, z \in \mathbb{F}$.

If the defined set $\mathbb{F}$ is a finite set, then, the field is said to be *finite* and we use the notation $\mathbb{F}_q$ such that q is the order of a finite field which is the number of the elements in the field. A finite field exists if and only if the order of the field q is a prime or power of a prime. We can write q as the power of a prime p such that $q = p^m$ for $m \geq 1$. Characteristic of the field is defined as the value of p. If we have $q = p$ that is $m = 1$, $\mathbb{F}$ is called as a prime field and for the case $m \geq 2$, $\mathbb{F}$ is called as an extension field.

Prime Fields

Let p be a prime number. The set $\{0, 1, 2, \ldots, p-1\}$ together with addition and multiplication over modulo p forms a finite field of order p. We will use the $\mathbb{F}_p$ notation for this field throughout the thesis. The study in this thesis focuses on prime fields with characteristic > 2.

Field Operations

By definition, any element $x \in \mathbb{F}$ has an additive inverse $-x$ such that $x + (-x) = 0$, and similarly has a multiplicative inverse x^{-1} such that $x \cdot x^{-1} = 1$. For the subtraction operation $x - y$, the inverse of the y is used, and similarly for the division operation x/y, inverse of the y is used as follows:

$$x - y = x + (-y)$$
$$x/y = x \cdot y^{-1}.$$

Note that, the cost for the subtraction operation is the sum of an addition cost and an additive inversion cost, similarly, the cost for the division operation is the sum of a multiplication cost and a multiplicative inversion cost.

Prime Field Arithmetic

The performance of ellipic curves depends heavily on the prime field arithmetic, especially, field multiplication and inversion. Thus, performing arithmetic in prime field $\mathbb{F}_p$ is at the heart of the elliptic curve cryptography.

We focus on prime fields with big characteristic, e.g. $p > 2^{200}$. Hence, the length of the elements will be larger than $200-$bit. However, CPU architecture sizes are mostly multiples of 8. Most common architectures are in $32-$bit and $64-$bit sizes.

Let elements of $\mathbb{F}_p$ be the integers from 0 to $p-1$ and $m = \lceil \log_2 p \rceil$ be the length of p. Assume the CPU architecture size is $n-$bit. In this case, the word count will be $l = \lceil m/n \rceil$ where each word will fit into the CPU register. After this point, a field element $x \in \mathbb{F}_p$ can be represented as an array $x = (x_{l-1}, x_{l-2}, \ldots, x_2, x_1, x_0)$ where

x_0 is the least significant word and array values $x_0, x_1, \ldots, x_{l-2}$ are $n-$bit, x_{l-1} is $(m - n(l-1))-$bit. From now on, the arithmetic operations will be handled on these arrays. A detailed representation called radix-2^r is discussed in Sections 3.2 and 3.3.1.

Addition and Subtraction

The addition and subtraction operations on $\mathbb{F}_p$ is handled on arrays with adding or subtracting the corresponding values while taking carry values into account. The Algorithm 1 shows prime field addition operation, including the reduction part at the end. Similarly, Algorithm 2 shows prime field subtraction operation.

Algorithm 1 Prime Field Addition

```
Input: $x = (x_{l-1}, x_{l-2}, \ldots, x_2, x_1, x_0), y = (y_{l-1}, y_{l-2}, \ldots, y_2, y_1, y_0); x, y \in \mathbb{F}_p$
Output: $z = x + y; z = (z_{l-1}, z_{l-2}, \ldots, z_2, z_1, z_0) \in \mathbb{F}_p$.
 1: $z_0 \leftarrow x_0 + y_0$
 2: if $z_0 \in [0, 2^n)$ then $c \leftarrow 0$
 3: else $c \leftarrow 1$
 4: end if
 5: for $i$ from $1$ to $l-1$ do
 6:     $z_i \leftarrow x_i + y_i + c$
 7:     if $z_i \in [0, 2^n)$ then $c \leftarrow 0$
 8:     else $c \leftarrow 1$
 9:     end if
10: end for
11: if $c = 1$ then $z \leftarrow z - p$
12: else if $z \geq p$ then $z \leftarrow z - p$
13: end if
14: return z
```

Algorithm 2 Prime Field Subtraction

Input: $x = (x_{l-1}, x_{l-2}, \ldots, x_2, x_1, x_0), y = (y_{l-1}, y_{l-2}, \ldots, y_2, y_1, y_0); x, y \in \mathbb{F}_p$

Output: $z = x - y; z = (z_{l-1}, z_{l-2}, \ldots, z_2, z_1, z_0) \in \mathbb{F}_p$.

```
1: $z_0 \leftarrow x_0 - y_0$
2: if $z_0 \in [0, 2^n)$ then $c \leftarrow 0$
3: else $c \leftarrow 1$
4: end if
5: for $i$ from $1$ to $l-1$ do
6:     $z_i \leftarrow x_i - y_i - c$
7:     if $z_i \in [0, 2^n)$ then $c \leftarrow 0$
8:     else $c \leftarrow 1$
9:     end if
10: end for
11: if $c = 1$ then $z \leftarrow z + p$
12: end if
13: return z
```

Multiplication and Division

Let $x, y, z \in \mathbb{F}_p$ and suppose we want to compute the product $x \cdot y = z$. For a multiplication over $\mathbb{F}_p$, there are two phases: The first phase is multiplication and the second phase is reduction. In Algorithm 3, the schoolbook method for multiplication is given without reduction part.

Considering the array representation with $n-$bit word size of the elements, the multiplication of each array values will result in a $2n-$bit value. To store these values, t_0 and t_1 values are used as temporary state values where size of each value is $n-$bit and $(t_1 t_0)$ is a double-word sized number which is the concenation of these values where t_0 is the least significant word.

For the division operation, one can use the combination of the operations inversion, multiplication and reduction.

Algorithm 3 Prime Field Multiplication Without Reduction

Input: $x = (x_{l-1}, x_{l-2}, \dots, x_2, x_1, x_0), y = (y_{l-1}, y_{l-2}, \dots, y_2, y_1, y_0); x, y \in \mathbb{F}_p$

Output: $Z = x \cdot y; Z = (Z_{2l-2}, Z_{2l-3}, \dots, Z_2, Z_1, Z_0)$ s.t. $Z_i \in [0, 2^n)$.

```
1:  for $i$ from $0$ to $2l-1$ do
2:      $Z_i \leftarrow 0$
3:  end for
4:  for $i$ from $0$ to $l-1$ do
5:      $t_1 \leftarrow 0$
6:      for $j$ from $0$ to $l-1$ do
7:          $(t_1 t_0) \leftarrow Z_{i+j} + x_i \cdot y_i + t_1$
8:          $Z_{i+j} \leftarrow t_0$
9:      end for
10:     $Z_{i+l} \leftarrow t_1$
11: end for
12: return Z
```

Reduction

Reduction part in field multiplication is an important part of the total cost. Especially, elliptic curve operations make heavy use of field multiplication. Hence, reduction with low cost should be considered. Based on the form of the prime number of the field, the reduction algorithms can be less expensive. There are various methods such as Barrett reduction or Montgomery reduction for arbitrary prime numbers.

Considering the prime fields, form of the prime number plays a crucial role. In Section 3.3.2, prime number forms are discussed. In this thesis, we focus on prime numbers with the form $p = 2^k - r$ s.t. $r > 0$, namely, Crandall primes. Thus, we give a straightforward method for reduction over Crandall primes in Algorithm 4. Note that, values t'_0, t'_1 are $(m - n(l-1))-$bit values as introduced in prime field arithmetic part.

Algorithm 4 Reduction for Crandall Primes

```
Input: $Z = (Z_{2l-2}, Z_{2l-3}, \ldots, Z_2, Z_1, Z_0)$ s.t. $Z_i \in [0, 2^n)$.
Output: $z = (z_{l-1}, z_{l-2}, \ldots, z_2, z_1, z_0); z \in \mathbb{F}_p$, s.t. $p = 2^k - r$
1: for $i$ from $0$ to $l-2$ do
2:     $(t_1 t_0) \leftarrow Z_i + Z_{i+l}$
3:     $z_i \leftarrow t_0$
4:     $z_{i+1} \leftarrow z_{i+1} + t_1$
5: end for
6: $(t'_1 t'_0) \leftarrow z_{l-1} + Z_{l-1} + Z_{2l-1}$
7: $z_{l-1} \leftarrow t'_0$
8: $z_0 \leftarrow z_0 + r t'_1$
9: $(t_1 t_0) \leftarrow z_0$
10: $z_0 \leftarrow t_0$
11: $z_1 \leftarrow z_1 + t_1$
12: return z
```

Inversion

The multiplicative inverse of a nonzero element $x \in \mathbb{F}_p$ is denoted as x^{-1}. To calculate x^{-1}, there are various methods such as extended Euclidean algorithm, binary inversion algorithm, Montgomery inversion method and Fermat's little theorem. We will focus on the method that uses Fermat's little theorem as it introduces a constant-time algorithm and can be optimized further for a specific prime number. Fermat's little theorem states that for any nonzero element $x \in \mathbb{F}_p$, we get $x^{p-1} \equiv 1 \bmod p$. From this equation, we can deduce $x^{p-2} \equiv x^{-1} \bmod p$. Thus, computing $x^{p-2} \bmod p$ yields the inverse of the element.

Unfortunately, for the case $p = 2^k - r$, the optimizations are not simple as it depends on both k and r. Generalization of the inversion on Crandall primes are discussed in [33]. Most of the time, the inversion algorithm requires meticulously handcrafted lifting method. We use an optimized example of inversion over $\mathbb{F}_{2^{266}-3}$ as it is introduced in [27]. The pseudocode of the inversion algorithm is given in Algorithm 5.

Algorithm 5 Inverse of an Element on $\mathbb{F}_{2^{266}-3}$ using Fermat's Little Theorem

```
Input: $x \in \mathbb{F}_{2^{266}-3}$
Output: $x^{2^{266}-5} \in \mathbb{F}_{2^{266}-3}$
 1: $t_0 \leftarrow x$
 2: $t_1 \leftarrow t_0 \cdot x$
 3: $t_2 \leftarrow t_1^2$
 4: $t_3 \leftarrow t_2^2$
 5: $t_3 \leftarrow t_3 \cdot t_1$
 6: $t_4 \leftarrow t_3^2$
 7: for $i$ from $1$ to $2$ do
 8:     $t_4 \leftarrow t_4^2$
 9: end for
10: $t_5 \leftarrow t_4^2$
11: $t_5 \leftarrow t_5 \cdot t_3$
12: $t_6 \leftarrow t_5^2$
13: for $i$ from $1$ to $7$ do
14:     $t_6 \leftarrow t_6^2$
15: end for
16: $t_6 \leftarrow t_6 \cdot t_5$
17: $t_7 \leftarrow t_6^2$
18: for $i$ from $1$ to $15$ do
19:     $t_7 \leftarrow t_7^2$
20: end for
21: $t_7 \leftarrow t_7 \cdot t_6$
22: $t_8 \leftarrow t_7^2$
23: for $i$ from $1$ to $31$ do
24:     $t_8 \leftarrow t_8^2$
25: end for
26: $t_8 \leftarrow t_8 \cdot t_7$
27: $t_9 \leftarrow t_8^2$
28: for $i$ from $1$ to $63$ do
29:     $t_9 \leftarrow t_9^2$
30: end for
31: $t_9 \leftarrow t_9 \cdot t_8$
32: $t_{10} \leftarrow t_9^2$
33: for $i$ from $1$ to $127$ do
34:     $t_{10} \leftarrow t_{10}^2$
35: end for
36: $t_{10} \leftarrow t_{10} \cdot t_9$
37: $t_{11} \leftarrow t_{10}^2$
38: for $i$ from $1$ to $6$ do
39:     $t_{11} \leftarrow t_{11}^2$
40: end for
41: $t_{12} \leftarrow t_{11} \cdot t_4$
42: $t_{12} \leftarrow t_{12} \cdot t_2$
43: $t_{12} \leftarrow t_{12} \cdot x$
44: for $i$ from $1$ to $3$ do
45:     $t_{12} \leftarrow t_{12}^2$
46: end for
47: $t_{13} \leftarrow t_{12} \cdot t_1$
48: return $t_{13}$
```

2.2 Elliptic Curves

An elliptic curve E over a field $\mathbb{F}_p$ is defined as the equation

$$E(\mathbb{F}_p) = \{\infty\} \cup \{(x, y) \in \mathbb{F}_p \times \mathbb{F}_p \mid y^2 = x^3 + Ax + B\}$$

where ∞ is the point at infinity and the discriminant of the curve $\Delta = 4A^3 + 27B^2 \neq 0$. The condition $\Delta \neq 0$ ensures that E has no singular point.

There are three different curve forms which are used by majority of the deployed elliptic curves in cryptography. We will be using the following notations, for the short Weierstrass form, twisted Edwards form and Montgomery form, respectively [14]:

$$\begin{aligned} E_W^{A,B} &: \quad y^2 = x^3 + Ax + B, \\ E_{Ed}^{a,d} &: \quad ax^2 + y^2 = 1 + dx^2y^2, \\ E_M^{A,B} &: \quad By^2 = x^3 + Ax^2 + x \end{aligned}$$

The Short Weierstrass form can be used to describe any type of elliptic curve over prime fields. But, the other two forms can only represent elliptic curves with orders satisfying the condition "4 divides $\#E(\mathbb{F}_p)$".

Note that all three forms have two parameters. But, most of the time, due to the efficiency requirements, one of them is fixed. Thus, the curves are only defined by one parameter. Usually, the fixed parameter, for short Weierstrass form, is $A = -3$; for twisted Edwards form, is $a = 1$ and for Montgomery form, is $B = 1$.

Additionally, every twisted Edwards curve defined over a prime field $\mathbb{F}_p$ is birationally equivalent to a Montgomery curve which is defined on the same prime field and vice versa. The transformation from twisted Edwards to Montgomery form are defined as follows [5]: Assume we have the Edwars curve $x^2 + y^2 = 1 + dx^2y^2$ where $d(1 - d)$ is nonzero defined over $\mathbb{F}_p$. Substituting $x = u/v$ and $y = (u - 1)/(u + 1)$, we get the Montgomery curve $Bv^2 = u^3 + Au^2 + u$ where $A = 2(1 + d)/(1 - d)$ and $B = 4/(1 - d)$.

Adding Points on an Elliptic Curve

Let $E_W^{A,B}$ be an elliptic curve. Let $P_1 = (x_1, y_1), P_2 = (x_2, y_2)$ be points on $E_W^{A,B}$ with $P_1, P_2 \neq \infty$. Define $P_1 + P_2 = P_3 = (x_3, y_3)$ as follows:

1. If $x_1 \neq x_2$;
   $x_3 = m^2 - x_1 - x_2, y_3 = m(x_1 - x_3) - y_1$ where $m = (y_2 - y_1)/(x_2 - x_1)$.
2. If $x_1 = x_2$ but $y_1 \neq y_2$, then $P_1 + P_2 = \infty$.
3. If $P_1 = P_2$ and $y_1 \neq 0$;
   $x_3 = m^2 - 2x_1, y_3 = m(x_1 - x_3) - y_1$ where $m = (3x_1^2 + A)/(2y_1)$.
4. If $P_1 = P_2$ and $y_1 = 0$; $P_1 + P_2 = \infty$.

For an explicit database of formulas for different elliptic curve forms, readers are referred to [7].

The Group Law

The following properties are satisfied for the addition of points on an elliptic curve E;

1. (Commutativity) $P_1 + P_2 = P_2 + P_1$ for all P_1, P_2 on E.
2. (Identity) $P + \infty = P$ for all P on E.
3. (Inverse) For a point P on E, there exists a point, denoted as $-P$ satisfying $P + (-P) = \infty$
4. (Associativity) $(P_1 + P_2) + P_3 = P_1 + (P_2 + P_3)$ for all P_1, P_2, P_3 on E.

These properties show the points on E form an abelian additive group.

Scalar Multiplication

Let P be a point on the elliptic curve E and $k > 0$ be an integer. The scalar multiplication (a.k.a point multiplication) $kP = Q$ where P and Q on E is defined as adding the P to itself k times:

$$kP = \underbrace{P + P + \cdots + P}_{k \text{ times}}.$$

This method is the exhaust way to compute the result. There are methods such as Double-and-add, Windowed method, Sliding-window method, Non-adjacent form method and Montgomery ladder method that speeds up the multiplication.

Elliptic Curve Cryptography

Elliptic curve cryptography (ECC) can be used to create Public Key Cryptography (PKC) primitives using elliptic curves over finite fields. ECC was first proposed by Koblitz [22] and Miller [26] independently in 1985 as a more efficient alternative to RSA [31]. Moreover, ECC achieves same security level with smaller key size compared other PKC primitives based on integer factorization or finite field discrete logarithm problem.

Suppose Alice and Bob want to agree on a secret value using an unsecure channel. An elliptic curve based Diffie-Hellman (ECDH) key exchange is the following protocol in its simplest form:

1. Alice and Bob agree on an elliptic curve $E(\mathbb{F}_q)$ with a public point $P \in E(\mathbb{F}_q)$ with large prime order.
2. Alice creates a secret integer a and computes $aP = P_a$. She sends it to Bob.
3. Bob creates a secret integer b and computes $bP = P_b$. He sends it to Alice.
4. Alice computes $aP_b = abP$.
5. Bob computes $bP_a = baP = abP$.
6. Using the common secret abP, Alice and Bob can derive their secret shared encryption key.

CHAPTER 3

TMVP-BASED FIELD MULTIPLICATION

Using Toeplitz matrices to perform field operations more efficient has been showed in several studies [1, 35]. In this chapter, we will give details about TMVP and propose 10-dimensional decomposition for matrix multiplication. We will also define criteria to get a Toeplitz matrix suitable form on every prime field with primes in Crandall form. Depending on this idea, we will choose some prime fields to work on and explain their representations.

3.1 Toeplitz Matrix-vector Multiplication

A Toeplitz matrix is a matrix in which each descending diagonal from left to right is constant. The form of an $n \times n$ Toeplitz matrix can be represented as follows:

$$\begin{bmatrix} a_0 & a_1 & a_2 & \cdots & a_{n-1} \\ a_n & a_0 & a_1 & \ddots & a_{n-2} \\ a_{n+1} & a_n & a_0 & \ddots & a_{n-3} \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ a_{2(n-1)} & \cdots & a_{n+1} & a_n & a_0 \end{bmatrix}$$

The $2-$way and $3-$way approaches for binary extension fields in [15] can easily be converted to the multiplication over the ring of integers. We advise the readers to check [15] for a comprehensive work about TMVP and its cost computation over binary extension fields. For a TMVP of size 2 over the ring of integers, we have:

$$T \cdot V = \begin{bmatrix} T_0 & T_1 \\ T_2 & T_0 \end{bmatrix} \cdot \begin{bmatrix} V_0 \\ V_1 \end{bmatrix} = \begin{bmatrix} P_1 + P_2 \\ P_1 + P_3 \end{bmatrix}$$

where T is 2×2 Toeplitz matrix and

$$\begin{aligned} P_1 &= T_0(V_0 + V_1) \\ P_2 &= (T_1 - T_0)V_1 \\ P_3 &= (T_2 - T_0)V_0. \end{aligned}$$

Using TMVP of size 2, the cost of computing $T \cdot V$ is $3\mathbf{M} + 3\mathbf{A} + 2\mathbf{A_d}$. While using schoolbook method, the cost is $4\mathbf{M} + 2\mathbf{A_d}$ where $\mathbf{M}$ is the cost of a multiplication operation, $\mathbf{A}$ and $\mathbf{A_d}$ are the costs of addition operations for single and double precision words, respectively. Thus, using TMVP, we save $1\mathbf{M}$, but get extra $1\mathbf{A} + 2\mathbf{A_d}$.

For a TMVP of size 3, we have:

$$T \cdot V = \begin{bmatrix} T_0 & T_1 & T_2 \\ T_3 & T_0 & T_1 \\ T_4 & T_3 & T_0 \end{bmatrix} \cdot \begin{bmatrix} V_0 \\ V_1 \\ V_2 \end{bmatrix} = \begin{bmatrix} P_3 + P_4 + P_6 \\ P_2 - P_4 + P_5 \\ P_1 - P_2 - P_3 \end{bmatrix}$$

where T is 3×3 Toeplitz matrix and

$$\begin{aligned} P_1 &= (T_4 + T_3 + T_0)V_0 & P_4 &= T_1(V_1 - V_2) \\ P_2 &= T_3(V_0 - V_1) & P_5 &= (T_0 + T_1 + T_3)V_1 \\ P_3 &= T_0(V_0 - V_2) & P_6 &= (T_0 + T_1 + T_2)V_2 \end{aligned}$$

Using TMVP of size 3, the cost of computing TV is $6\mathbf{M}+8\mathbf{A}+6\mathbf{A_d}$, using schoolbook method, the cost is $9\mathbf{M} + 6\mathbf{A_d}$. Thus, using TMVP, we save $3\mathbf{M}$, but get extra $8\mathbf{A}$.

For a TMVP of size 4, we can split the matrix into four 2×2 Toeplitz matrices which enables the recursive computation. It should be noted that the recursive use of this algorithm results in more gaining in the computation. This will be discussed in the following sections.

3.2 The Proposed Decomposition for 10-Dimensional TMVP

In this section, we will propose a 10-dimensional TMVP strategy to be used for field multiplication. Suppose we want to multiply two $10-$term polynomials, namely, $f(x)$

and $g(x)$;

$$f(x) = \sum_{i=0}^{9} f_i x^i \qquad\qquad g(x) = \sum_{i=0}^{9} g_i x^i$$

The straightforward method to multiply these polynomials is the schoolbook multiplication. We can compute $f(x) \cdot g(x) = h(x)$ as follows:

$$h(x) = \sum_{i=0}^{9} \sum_{j=0}^{9} f_i g_j x^{i+j}$$

When the polynomials are multiplied in modulo $x^{10} - 1$, one can write this multiplication using the matrix operations which yields the following matrix equation:

$$\begin{bmatrix} \color{red}{g_0} & g_9 & g_8 & g_7 & g_6 & g_5 & g_4 & g_3 & g_2 & g_1 \\ g_1 & \color{red}{g_0} & g_9 & g_8 & g_7 & g_6 & g_5 & g_4 & g_3 & g_2 \\ g_2 & g_1 & \color{red}{g_0} & g_9 & g_8 & g_7 & g_6 & g_5 & g_4 & g_3 \\ g_3 & g_2 & g_1 & \color{red}{g_0} & g_9 & g_8 & g_7 & g_6 & g_5 & g_4 \\ g_4 & g_3 & g_2 & g_1 & \color{red}{g_0} & g_9 & g_8 & g_7 & g_6 & g_5 \\ g_5 & g_4 & g_3 & g_2 & g_1 & \color{red}{g_0} & g_9 & g_8 & g_7 & g_6 \\ g_6 & g_5 & g_4 & g_3 & g_2 & g_1 & \color{red}{g_0} & g_9 & g_8 & g_7 \\ g_7 & g_6 & g_5 & g_4 & g_3 & g_2 & g_1 & \color{red}{g_0} & g_9 & g_8 \\ g_8 & g_7 & g_6 & g_5 & g_4 & g_3 & g_2 & g_1 & \color{red}{g_0} & g_9 \\ g_9 & g_8 & g_7 & g_6 & g_5 & g_4 & g_3 & g_2 & g_1 & \color{red}{g_0} \end{bmatrix} \cdot \begin{bmatrix} f_0 \\ f_1 \\ f_2 \\ f_3 \\ f_4 \\ f_5 \\ f_6 \\ f_7 \\ f_8 \\ f_9 \end{bmatrix} = \begin{bmatrix} h_0 \\ h_1 \\ h_2 \\ h_3 \\ h_4 \\ h_5 \\ h_6 \\ h_7 \\ h_8 \\ h_9 \end{bmatrix}$$

Please note that the 10×10 matrix is in Toeplitz matrix form. Hence, we can use TMVP operations to compute the result. For the building blocks, we have TMVP computations for the sizes 2×2 and 3×3 as introduced in previous section. Thus, we develop our strategy to mount these operations for the corresponding matrix multiplication.

The size 10 has been chosen specificially for the ease of implementation on both $32-$bit and $64-$bit implementations. The matrix will be splitted into upper and lower triangular matrices where each of them will be splitted into 5×5 matrices later. This decision will enable efficient implementation of the matrix multiplication even with bigger coefficients. The splitted form of the operation is shown below:

$$\left(\left[\begin{array}{ccccc|ccccc}
g_0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
g_1 & g_0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
g_2 & g_1 & g_0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
g_3 & g_2 & g_1 & g_0 & 0 & 0 & 0 & 0 & 0 & 0 \\
g_4 & g_3 & g_2 & g_1 & g_0 & 0 & 0 & 0 & 0 & 0 \\
\hline
g_5 & g_4 & g_3 & g_2 & g_1 & g_0 & 0 & 0 & 0 & 0 \\
g_6 & g_5 & g_4 & g_3 & g_2 & g_1 & g_0 & 0 & 0 & 0 \\
g_7 & g_6 & g_5 & g_4 & g_3 & g_2 & g_1 & g_0 & 0 & 0 \\
g_8 & g_7 & g_6 & g_5 & g_4 & g_3 & g_2 & g_1 & g_0 & 0 \\
g_9 & g_8 & g_7 & g_6 & g_5 & g_4 & g_3 & g_2 & g_1 & g_0
\end{array}\right] + \left[\begin{array}{ccccc|ccccc}
0 & g_9 & g_8 & g_7 & g_6 & g_5 & g_4 & g_3 & g_2 & g_1 \\
0 & 0 & g_9 & g_8 & g_7 & g_6 & g_5 & g_4 & g_3 & g_2 \\
0 & 0 & 0 & g_9 & g_8 & g_7 & g_6 & g_5 & g_4 & g_3 \\
0 & 0 & 0 & 0 & g_9 & g_8 & g_7 & g_6 & g_5 & g_4 \\
0 & 0 & 0 & 0 & 0 & g_9 & g_8 & g_7 & g_6 & g_5 \\
\hline
0 & 0 & 0 & 0 & 0 & 0 & g_9 & g_8 & g_7 & g_6 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & g_9 & g_8 & g_7 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & g_9 & g_8 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & g_9 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{array}\right]\right) \cdot \left[\begin{array}{c}
f_0 \\ f_1 \\ f_2 \\ f_3 \\ f_4 \\ \hline f_5 \\ f_6 \\ f_7 \\ f_8 \\ f_9
\end{array}\right]$$

We can split this matrix into submatrices to mount 2×2 or 3×3 TMVP operations. The matrix can be splitted into submatrices as follows:

$$\left[\begin{array}{c|c}
A_{0_{5\times5}} & 0_{5\times5} \\
\hline
A_{1_{5\times5}} & A_{0_{5\times5}}
\end{array}\right]_{10\times10} \cdot \left[\begin{array}{c}
B_{0_{5\times1}} \\ \hline B_{1_{5\times1}}
\end{array}\right]_{10\times1} + \cdot \left[\begin{array}{c|c}
A_{2_{5\times5}} & A_{1_{5\times5}} \\
\hline
0_{5\times5} & A_{2_{5\times5}}
\end{array}\right]_{10\times10} \cdot \left[\begin{array}{c}
B_{0_{5\times1}} \\ \hline B_{1_{5\times1}}
\end{array}\right]_{10\times1}$$

where matrices A_i for $i = 0, 1, 2$ are 5×5 matrices, B_j for $j = 0, 1$ are 5×1 matrices and $0_{5\times5}$ is 5×5 zero matrix.

Note that, 5×5 matrices can be split asymmetrically into 1 column by 4 columns or 3 columns by 2 columns and vice versa for both case. Thus, considering all split forms that allows to mount 2×2 or 3×3 TMVP, there are $4^4 + 4^4 = 512$ different forms. We have investigated all such forms and figured out the one with the lowest cost as details are introduced in the following subsections.

We need to compute the products: $A_0B_0, A_0B_1, A_1B_0, A_1B_1, A_2B_0$ and A_2B_1. The computation details of every operation is given in the following subsection.

3.2.1 Computation of Submatrices

3.2.1.1 Computing A_0B_0

$$A_0B_0 = \left[\begin{array}{c|c} K_{2_{1x4}} & K_{3_{1x1}} \\ \hline K_{0_{4\times4}} & K_{1_{4x1}} \end{array}\right]_{5\times5} \cdot \left[\begin{array}{c} L_{0_{4x1}} \\ \hline L_{1_{1x1}} \end{array}\right]_{5\times1} = \left[\begin{array}{c} K_2L_{0_{1x1}} + K_3L_{1_{1x1}} \\ \hline K_0L_{0_{4x1}} + K_1L_{1_{4x1}} \end{array}\right]_{5\times5}$$

K_0L_0 : will be calculated later with TMVP approach.

$$K_1L_1 : \begin{bmatrix} 0 \\ 0 \\ 0 \\ g_0 \end{bmatrix} \cdot \begin{bmatrix} f_4 \end{bmatrix} = \cancel{4}\mathtt{M} = 1\mathtt{M}$$

$$K_2L_0 : \begin{bmatrix} g_0 & 0 & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} f_0 \\ f_1 \\ f_2 \\ f_3 \end{bmatrix} = \cancel{4\mathtt{M} + 3\mathtt{A_d}} = 1\mathtt{M}$$

$$K_3L_1 : \begin{bmatrix} 0 \end{bmatrix} \cdot \begin{bmatrix} f_4 \end{bmatrix} = \cancel{1}\mathtt{M} = 0$$

Last additions cost extra $5\mathtt{A_d}$ additions. But, we have K_3L_1 with no cost and K_1L_1 with only $1\mathtt{M}$, thus, total cost is:

$$Cost(K_0L_0) + 2\mathtt{M} + 1\mathtt{A_d}$$

3.2.1.2 Computing A_0B_1

$$A_0B_1 = \left[\begin{array}{c|c} K_{2_{1x4}} & K_{3_{1x1}} \\ \hline K_{0_{4\times4}} & K_{1_{4x1}} \end{array}\right]_{5\times5} \cdot \left[\begin{array}{c} N_{0_{4x1}} \\ \hline N_{1_{1x1}} \end{array}\right]_{5\times1} = \left[\begin{array}{c} K_2N_{0_{1x1}} + K_3N_{1_{1x1}} \\ \hline K_0N_{0_{4x1}} + K_1N_{1_{4x1}} \end{array}\right]_{5\times5}$$

K_0N_0 : will be calculated later with TMVP approach.

$$K_1N_1 : \begin{bmatrix} 0 \\ 0 \\ 0 \\ g_0 \end{bmatrix} \cdot \begin{bmatrix} f_9 \end{bmatrix} \qquad = \cancel{4}\mathbf{M} = 1\mathbf{M}$$

$$K_2N_0 : \begin{bmatrix} g_0 & 0 & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} f_5 \\ f_6 \\ f_7 \\ f_8 \end{bmatrix} \qquad = \cancel{4\mathbf{M} + 3\mathbf{A_d}} = 1\mathbf{M}$$

$$K_3N_1 : \begin{bmatrix} 0 \end{bmatrix} \cdot \begin{bmatrix} f_9 \end{bmatrix} \qquad = \cancel{1}\mathbf{M} = 0$$

Last additions cost extra $5\mathbf{A_d}$ additions. But, we have K_3N_1 with no cost and K_1N_1 with only $1\mathbf{M}$, thus, total cost is:

$$Cost(K_0N_0) + 2\mathbf{M} + 1\mathbf{A_d}$$

3.2.1.3 Computing A_1B_0

$$A_1B_0 = \left[\begin{array}{c|c} P_{0_{4\times4}} & P_{1_{4x1}} \\ \hline P_{2_{1x4}} & P_{3_{1x1}} \end{array}\right]_{5\times5} \cdot \left[\begin{array}{c} L_{0_{4x1}} \\ \hline L_{1_{1x1}} \end{array}\right]_{5\times1} = \left[\begin{array}{c} P_0L_{0_{4x1}} + P_1L_{1_{4x1}} \\ \hline P_2L_{0_{1x1}} + P_3L_{1_{1x1}} \end{array}\right]_{5\times5}$$

P_0L_0 : will be calculated later with TMVP approach.

$$P_1L_1 : \begin{bmatrix} g_1 \\ g_2 \\ g_3 \\ g_4 \end{bmatrix} \cdot \begin{bmatrix} f_4 \end{bmatrix} \qquad = 4\mathbf{M}$$

$$P_2L_0 : \begin{bmatrix} g_9 & g_8 & g_7 & g_6 \end{bmatrix} \cdot \begin{bmatrix} f_0 \\ f_1 \\ f_2 \\ f_3 \end{bmatrix} \qquad = 4\mathbf{M} + 3\mathbf{A_d}$$

$$P_3L_1 : \begin{bmatrix} g_5 \end{bmatrix} \cdot \begin{bmatrix} f_4 \end{bmatrix} \qquad = 1\mathbf{M}$$

Last additions cost extra $5\mathtt{A_d}$ additions, thus, total cost is:

$$Cost(P_0L_0) + 9\mathtt{M} + 8\mathtt{A_d}$$

3.2.1.4 Computing A_1B_1

$$A_1B_1 = \left[\begin{array}{c|c} P_{0_{4\times4}} & P_{1_{4x1}} \\ \hline P_{2_{1x4}} & P_{3_{1x1}} \end{array}\right]_{5\times5} \cdot \left[\begin{array}{c} N_{0_{4x1}} \\ \hline N_{1_{1x1}} \end{array}\right]_{5\times1} = \left[\begin{array}{c} P_0N_{0_{4x1}} + P_1N_{1_{4x1}} \\ \hline P_2N_{0_{1x1}} + P_3N_{1_{1x1}} \end{array}\right]_{5\times5}$$

P_0N_0 : will be calculated later with TMVP approach.

$$P_1N_1 : \begin{bmatrix} g_1 \\ g_2 \\ g_3 \\ g_4 \end{bmatrix} \cdot \begin{bmatrix} f_9 \end{bmatrix} \qquad = 4\mathtt{M}$$

$$P_2N_0 : \begin{bmatrix} g_9 & g_8 & g_7 & g_6 \end{bmatrix} \cdot \begin{bmatrix} f_5 \\ f_6 \\ f_7 \\ f_8 \end{bmatrix} \qquad = 4\mathtt{M} + 3\mathtt{A_d}$$

$$P_3N_1 : \begin{bmatrix} g_5 \end{bmatrix} \cdot \begin{bmatrix} f_9 \end{bmatrix} \qquad = 1\mathtt{M}$$

Last additions cost extra $5A_d$ additions, thus, total cost is:

$$Cost(P_0N_0) + 9\mathtt{M} + 8\mathtt{A_d}$$

3.2.1.5 Computing A_2B_0

$$A_2B_0 = \left[\begin{array}{c|c} R_{1_{4x1}} & R_{0_{4\times4}} \\ \hline R_{3_{1x1}} & R_{2_{1x4}} \end{array}\right]_{5\times5} \cdot \left[\begin{array}{c} L'_{1_{1x1}} \\ \hline L'_{0_{4x1}} \end{array}\right]_{5\times1} = \left[\begin{array}{c} R_0L'_{0_{4x1}} + R_1L'_{1_{4x1}} \\ \hline R_2L'_{0_{1x1}} + R_3L'_{1_{1x1}} \end{array}\right]_{5\times5}$$

$R_0L_0^{'}:$ will be calculated later with TMVP approach.

$$R_1L_1^{'}: \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \cdot \begin{bmatrix} f_0 \end{bmatrix} \qquad = \cancel{4\mathtt{M}} = 0$$

$$R_2L_0^{'}: \begin{bmatrix} 0 & 0 & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} f_1 \\ f_2 \\ f_3 \\ f_4 \end{bmatrix} \qquad = \cancel{4\mathtt{M} + 3\mathtt{A_d}} = 0$$

$$R_3L_1^{'}: \begin{bmatrix} 0 \end{bmatrix} \cdot \begin{bmatrix} f_0 \end{bmatrix} \qquad = \cancel{1\mathtt{M}} = 0$$

Last additions cost extra $5\mathtt{A_d}$ additions. But, we have no cost, thus, total cost is:

$$Cost(R_0L_0^{'})$$

3.2.1.6 Computing A_2B_1

$$A_2B_0 = \left[\begin{array}{c|c} R_{1_{4x1}} & R_{0_{4\times4}} \\ \hline R_{3_{1x1}} & R_{2_{1x4}} \end{array}\right]_{5\times5} \cdot \left[\begin{array}{c} N^{'}_{1_{1x1}} \\ \hline N^{'}_{0_{4x1}} \end{array}\right]_{5\times1} = \left[\begin{array}{c} R_0N^{'}_{0_{4x1}} + R_1N^{'}_{1_{4x1}} \\ \hline R_2N^{'}_{0_{1x1}} + R_3N^{'}_{1_{1x1}} \end{array}\right]_{5\times5}$$

$R_0N_0^{'}:$ will be calculated later with TMVP approach.

$$R_1N_1^{'}: \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \cdot \begin{bmatrix} f_5 \end{bmatrix} \qquad = \cancel{4\mathtt{M}} = 0$$

$$R_2N_0^{'}: \begin{bmatrix} 0 & 0 & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} f_6 \\ f_7 \\ f_8 \\ f_9 \end{bmatrix} \qquad = \cancel{4\mathtt{M} + 3\mathtt{A_d}} = 0$$

$$R_3N_1^{'}: \begin{bmatrix} 0 \end{bmatrix} \cdot \begin{bmatrix} f_5 \end{bmatrix} \qquad = \cancel{1\mathtt{M}} = 0$$

Last additions cost extra $5\mathtt{A_d}$ additions. But, we have no cost, thus, total cost is:

$$Cost(R_0N_0^{'})$$

3.2.1.7 Computing the cost for K_0L_0 and K_0N_0

$$\begin{aligned}
K_0L_0 &= \left[\begin{array}{cc|cc} g_1 & g_0 & 0 & 0 \\ g_2 & g_1 & g_0 & 0 \\ \hline g_3 & g_2 & g_1 & g_0 \\ g_4 & g_3 & g_2 & g_1 \end{array}\right] \cdot \left[\begin{array}{c} f_0 \\ f_1 \\ \hline f_2 \\ f_3 \end{array}\right] \\
&= \left[\begin{array}{c|c} X_0 & X_1 \\ \hline X_2 & X_0 \end{array}\right] \cdot \left[\begin{array}{c} C_0 \\ \hline C_1 \end{array}\right] \\
&= \left[\begin{array}{c} T_1 + T_2 \\ \hline T_1 + T_3 \end{array}\right]
\end{aligned}$$

$T_1 = X_0(C_0 + C_1), T_2 = (X_1 - X_0)C_1, T_3 = (X_2 - X_0)C_0$

Total cost is:

$$Cost(K_0L_0) = 3\,(M(2)) + 8\mathtt{A} + 4\mathtt{A_d}$$

where, $M(2)$ is the cost of computing 2×2 TMVP.

$$\begin{aligned}
K_0N_0 &= \left[\begin{array}{cc|cc} g_1 & g_0 & 0 & 0 \\ g_2 & g_1 & g_0 & 0 \\ \hline g_3 & g_2 & g_1 & g_0 \\ g_4 & g_3 & g_2 & g_1 \end{array}\right] \cdot \left[\begin{array}{c} f_5 \\ f_6 \\ \hline f_7 \\ f_8 \end{array}\right] \\
&= \left[\begin{array}{c|c} X_0 & X_1 \\ \hline X_2 & X_0 \end{array}\right] \cdot \left[\begin{array}{c} D_0 \\ \hline D_1 \end{array}\right] \\
&= \left[\begin{array}{c} T_1 + T_2 \\ \hline T_1 + T_3 \end{array}\right]
\end{aligned}$$

$T_1 = X_0(D_0 + D_1), T_2 = D_1(X_1 - X_0), T_3 = D_0(X_2 - X_0)$

We have $(X_1 - X_0)$ and $(X_2 - X_0)$ computed already, thus, total cost is:

$$\begin{array}{rrrl} Cost(K_0N_0) = & 3\,(M(2)) & +8\mathtt{A} & +4\mathtt{A_d} \\ & & -3\mathtt{A} & \leftarrow (X_1 - X_0) \\ + & & -3\mathtt{A} & \leftarrow (X_2 - X_0) \\ \hline & 3\,(M(2)) & +2\mathtt{A} & +4\mathtt{A_d} \end{array}$$

where, $M(2)$ is the cost of computing 2×2 TMVP.

3.2.1.8 Computing the cost for P_0L_0 and P_0N_0

$$\begin{aligned} P_0L_0 &= \left[\begin{array}{cc|cc} g_5 & g_4 & g_3 & g_2 \\ g_6 & g_5 & g_4 & g_3 \\ \hline g_7 & g_6 & g_5 & g_4 \\ g_8 & g_7 & g_6 & g_5 \end{array}\right] \cdot \left[\begin{array}{c} f_0 \\ f_1 \\ \hline f_2 \\ f_3 \end{array}\right] \\ &= \left[\begin{array}{c|c} Y_0 & Y_1 \\ \hline Y_2 & Y_0 \end{array}\right] \cdot \left[\begin{array}{c} C_0 \\ \hline C_1 \end{array}\right] \\ &= \left[\begin{array}{c} T_1 + T_2 \\ \hline T_1 + T_3 \end{array}\right] \end{aligned}$$

$T_1 = Y_0(C_0 + C_1)$, $T_2 = C_1(Y_1 - Y_0)$, $T_3 = C_0(Y_2 - Y_0)$

We have $(C_0 + C_1)$ computed already, thus, total cost is:

$$\begin{array}{rrrl} Cost(P_0L_0) = & 3\,(M(2)) & +8\mathtt{A} & +4\mathtt{A_d} \\ + & & -2\mathtt{A} & \leftarrow (C_0 + C_1) \\ \hline & 3\,(M(2)) & +6\mathtt{A} & +4\mathtt{A_d} \end{array}$$

where, $M(2)$ is the cost of computing 2×2 TMVP.

$$P_0N_0 = \left[\begin{array}{cc|cc} g_5 & g_4 & g_3 & g_2 \\ g_6 & g_5 & g_4 & g_3 \\ \hline g_7 & g_6 & g_5 & g_4 \\ g_8 & g_7 & g_6 & g_5 \end{array}\right] \cdot \left[\begin{array}{c} f_5 \\ f_6 \\ \hline f_7 \\ f_8 \end{array}\right]$$

$$= \left[\begin{array}{c|c} Y_0 & Y_1 \\ \hline Y_2 & Y_0 \end{array}\right] \cdot \left[\begin{array}{c} D_0 \\ \hline D_1 \end{array}\right]$$

$$= \left[\begin{array}{c} T_1 + T_2 \\ \hline T_1 + T_3 \end{array}\right]$$

$T_1 = Y_0(D_0 + D_1), T_2 = D_1(Y_1 - Y_0), T_3 = C_0(Y_2 - Y_0)$

We have $(D_0 + D_1)$, $(Y_1 - Y_0)$ and $(Y_2 - Y_0)$ computed already, thus, total cost is:

$$\begin{array}{rlll} Cost(P_0N_0) = & 3\,(M(2)) & +8\mathbf{A} & +4\mathbf{A_d} \\ & & -2\mathbf{A} & \leftarrow (D_0 + D_1) \\ & & -3\mathbf{A} & \leftarrow (Y_1 - Y_0) \\ + & & -3\mathbf{A} & \leftarrow (Y_2 - Y_0) \\ \hline & 3\,(M(2)) & & +4\mathbf{A_d} \end{array}$$

where, $M(2)$ is the cost of computing 2×2 TMVP.

3.2.1.9 Computing the cost for $R_0L_0^{'}$ and $R_0N_0^{'}$

$$R_0L_0^{'} = \left[\begin{array}{cc|cc} g_9 & g_8 & g_7 & g_6 \\ 0 & g_9 & g_8 & g_7 \\ \hline 0 & 0 & g_9 & g_8 \\ 0 & 0 & 0 & g_9 \end{array}\right] \cdot \left[\begin{array}{c} f_1 \\ f_2 \\ \hline f_3 \\ f_4 \end{array}\right]$$

$$= \left[\begin{array}{c|c} Z_0 & Y_2 \\ \hline 0 & Z_0 \end{array}\right] \cdot \left[\begin{array}{c} C_0^{'} \\ \hline C_1^{'} \end{array}\right]$$

$$= \left[\begin{array}{c} T_1 + T_2 \\ \hline T_1 + T_3 \end{array}\right]$$

$T_1 = Z_0(C_0' + C_1'), T_2 = C_1'(Y_2 - Z_0), T_3 = C_0'(0 - Z_0)$

Total cost is:

$$Cost(R_0L_0') = 3\,(M(2)) + 5\mathtt{A} + 4\mathtt{A_d}$$

where, $M(2)$ is the cost of computing 2×2 TMVP.

$$\begin{aligned}
R_0N_0' &= \left[\begin{array}{cc|cc} g_9 & g_8 & g_7 & g_6 \\ 0 & g_9 & g_8 & g_7 \\ \hline 0 & 0 & g_9 & g_8 \\ 0 & 0 & 0 & g_9 \end{array}\right] \cdot \left[\begin{array}{c} f_6 \\ f_7 \\ \hline f_8 \\ f_9 \end{array}\right] \\
&= \left[\begin{array}{c|c} Z_0 & Y_2 \\ \hline 0 & Z_0 \end{array}\right] \cdot \left[\begin{array}{c} D_0' \\ \hline D_1' \end{array}\right] \\
&= \left[\begin{array}{c} T_1 + T_2 \\ \hline T_1 + T_3 \end{array}\right]
\end{aligned}$$

$T_1 = Z_0(D_0' + D_1'), T_2 = D_1'(Y_2 - Z_0), T_3 = D_0'(0 - Z_0)$

We have $(Y_2 - Z_0)$ and $(0 - Z_0)$ computed already, thus, total cost is:

$$\begin{array}{rlll}
Cost(R_0N_0') = & 3\,(M(2)) & +8\mathtt{A} & +4\mathtt{A_d} \\
 & & -3\mathtt{A} & \leftarrow (Y_2 - Z_0) \\
+ & & -3\mathtt{A} & \leftarrow (0 - Z_0) \\ \hline
 & 3\,(M(2)) & +2\mathtt{A} & +4\mathtt{A_d}
\end{array}$$

where, $M(2)$ is the cost of computing 2×2 TMVP.

3.2.2 Arithmetic Cost and Comparison

As discussed in the previous section, we have A_0B_0, A_0B_1, A_1B_0, A_1B_1, A_2B_0 and A_2B_1 computed using TMVP method. The cost decomposition of the proposed algorithm is given in Table 3.1. The costs are based on the products of 5×5 matrices.

Table 3.1: The Cost Computation of Matrices

Product	**Cost**
A_0B_0	$Cost(K_0L_0) + 2\mathtt{M} + 1\mathtt{A_d}$
A_0B_1	$Cost(K_0N_0) + 2\mathtt{M} + 1\mathtt{A_d}$
A_1B_0	$Cost(P_0L_0) + 9\mathtt{M} + 8\mathtt{A_d}$
A_1B_1	$Cost(P_0N_0) + 9\mathtt{M} + 8\mathtt{A_d}$
A_2B_0	$Cost(R_0L_0')$
A_2B_1	$Cost(R_0N_0')$

The $Cost()$ function in the table returns the cost of computing the corresponding TMVP and $K_0N_0, P_0L_0, P_0N_0, R_0L_0'$ and R_0N_0' are corresponding TMVP operations for the matrices A_0B_1, A_1B_0, A_1B_1, A_2B_0 and A_2B_1, respectively.
To compute 4×4 TMVP, we can use 2×2 TMVP method and cost will be as follows:

$$Cost(M_{4\times4}) = Cost(M_{2\times2}) + 8\mathtt{A} + 4\mathtt{A_d}$$

where the cost for $M_{2\times2}$ has been discussed in Section 3.1.
Total cost for computing the TMVP operations are listed as follows:

$$\begin{array}{rclll}
Cost(K_0L_0) & = & 3\,(M_{2\times2}) & +8\mathtt{A} & +4\mathtt{A_d} \\
Cost(K_0N_0) & = & 3\,(M_{2\times2}) & +2\mathtt{A} & +4\mathtt{A_d} \\
Cost(P_0L_0) & = & 3\,(M_{2\times2}) & +6\mathtt{A} & +4\mathtt{A_d} \\
Cost(P_0N_0) & = & 3\,(M_{2\times2}) & & +4\mathtt{A_d} \\
Cost(R_0L_0') & = & 3\,(M_{2\times2}) & +5\mathtt{A} & +4\mathtt{A_d} \\
Cost(R_0N_0') & = & 3\,(M_{2\times2}) & +2\mathtt{A} & +4\mathtt{A_d} \\
 & + & & & \\
\hline
 & & 18\,(M_{2\times2}) & +23\mathtt{A} & +24\mathtt{A_d}
\end{array}$$

The cost of a multiplication by $M_{2\times2}$ can be computed both using TMVP and schoolbook multiplication method. Thus, we get two different costs for it, as discussed in Section 3.1. We obtain two different costs at the end. The cost comparison is given in the Table 3.2.

Table 3.2: The Cost Comparison of Algorithms

Method	**Cost**
The Proposed Method #1	$77\mathtt{M} + 77\mathtt{A} + 109\mathtt{A_d}$
The Proposed Method #2	$95\mathtt{M} + 23\mathtt{A} + 109\mathtt{A_d}$
Bernstein [4]	$101\mathtt{M} + 92\mathtt{A_d}$

Note that, $\mathtt{A}$ is the single word addition operation and it is assumed that the cost of computing $2\mathtt{A}$ is equivalent computing $1\mathtt{A_d}$. The Proposed Method #1 uses TMVP to compute $M_{2\times2}$ and The Proposed Method #2 uses schoolbook multiplication. The cost for the multiplication with the coefficients are ignored since it varies for every prime field and will have same cost on both proposed methods and Bernstein's method.
Comparing the results with Bernstein's complexity, we get the following savings and redundant operations.

- The Proposed Method #1: $-24\mathtt{M}$ and $+56\mathtt{A_d}$
- The Proposed Method #2: $-6\mathtt{M}$ and $+29\mathtt{A_d}$

We saved multiplications but got extra additions and expect to have improvements based on this trade off. Theoretically, using Method #1, if we have a platform that has the cost of $1\mathtt{M}$ is equivalent to at least $2.33\mathtt{A_d}$, our algorithms have better results in terms of total operation cost. The improvements are discussed in Sections 4.3 and 5.3. A reference implementation is given in Appendix A.

3.2.3 Delay Evaluation

The delay complexity for $n \times n$ Toeplitz matrix is defined as $\mathbb{D}(n) = 2\log_2(n)\mathbb{D}_{\mathtt{M}} + \mathbb{D}_{\mathtt{A}}$ where $\mathbb{D}_{\mathtt{M}}$ and $\mathbb{D}_{\mathtt{A}}$ are the delays of computing field multiplication and addition operations, respectively.
We have evaluated the delay complexity of the Proposed Method #1 when it is implemented using a four-core parallel implementation. In this case, we will have computing paths C_1, C_2, C_3 and C_4.

Using first three paths, we compute the matrices mentioned in Section 4.2 and last path is for extra operations. Delay for paths C_1 and C_2 is $18\mathbb{D}_{\mathtt{M}} + 57\mathbb{D}_{\mathtt{A}}$, for C_3 is $18\mathbb{D}_{\mathtt{M}} + 58\mathbb{D}_{\mathtt{A}}$ and for C_4 is $22\mathbb{D}_{\mathtt{M}} + 29\mathbb{D}_{\mathtt{A}}$.

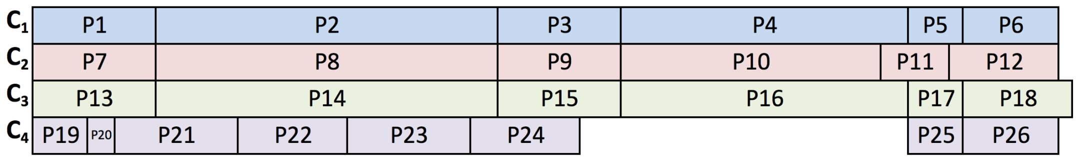

Figure 3.1: Four-core Parallel Implementation Delay Overview

Path C_1 contains the delay of the computations of matrices K_0L_0 and K_0N_0 as stated in Figure 3.1 with parts P_1, P_2, P_3 and P_4 and the delay costs are $9\mathbb{D}_{\mathtt{M}}$, $25\mathbb{D}_{\mathtt{A}}$, $9\mathbb{D}_{\mathtt{M}}$ and $21\mathbb{D}_{\mathtt{A}}$, respectively.
Similarly, path C_2 is for P_0L_0 and P_0N_0 with parts $P_7(9\mathbb{D}_{\mathtt{M}})$, $P_8(25\mathbb{D}_{\mathtt{A}})$, $P_9(9\mathbb{D}_{\mathtt{M}})$ and $P_{10}(19\mathbb{D}_{\mathtt{A}})$, path C_3 is for $R_0L_0^{'}$ and $R_0N_0^{'}$ with parts P_{13}, P_{14}, P_{15} and P_{16} having the same delay costs with path C_1. The paths $P_5(4\mathbb{D}_{\mathtt{A}})$, $P_{11}(5\mathbb{D}_{\mathtt{A}})$, $P_{17}(4\mathbb{D}_{\mathtt{A}})$ and $P_{25}(4\mathbb{D}_{\mathtt{A}})$ represent the final additions.
The path C_4 represents the delay for extra computations mentioned in the Table 3.1 with parts P_{19}, P_{20}, P_{21}, P_{22}, P_{23} and P_{24}. We finalize the computation with the multiplication of the constant as mentioned in Section 4.2 with delay costs represented in parts $P_6(7\mathbb{D}_{\mathtt{A}})$, $P_{12}(8\mathbb{D}_{\mathtt{A}})$, $P_{18}(8\mathbb{D}_{\mathtt{A}})$ and $P_{26}(7\mathbb{D}_{\mathtt{A}})$.
As shown in Figure 3.1, the final addition operations have to wait for matrix operations be completed which causes a gap in path C_4. But, the critical path is C_3 since it has the longest delay that corresponds to the delay of the algorithm.
If algorithm is implemented in single-core, its delay would be $76\mathbb{D}_{\mathtt{M}} + 201\mathbb{D}_{\mathtt{A}}$. Comparing it with the delay of the critical path, we can deduce the proposed four-core parallel implementation is almost embarrassingly parallel.

3.3 TMVP-Friendly Prime Fields

As it is introduced in [2, 35], TMVP method can be used for the finite field multiplication to get faster implementations. But the main problem is to find a proper matrix representation for the field elements which is in Toeplitz matrix form. It is possible to manipulate the coefficients and get TMVP suitable results. Being TMVP-Friendly means the matrix representation of the corresponding prime field yields a matrix in

Toeplitz form. This thesis focuses on finding and improving TMVP-Friendly prime fields and element representations without doing complex manipulations. To accomplish this, we will stick with the radix-2^r representation.

3.3.1 Toeplitz Matrix Formed Field Element Representation

Prime field elements are usually represented as big integers and these integers are usually divided into several small chunks called **limb**s, so that field operations can be carried out as sequences of operations on limbs. A radix-2^r representation represents an element f in a $b-$bit prime field as $(f_0, f_1, \ldots, f_{\lceil b/r \rceil - 1})$, such that

$$f = \sum_{i=0}^{\lceil b/r \rceil - 1} f_i 2^{\lceil ir \rceil}.$$

Field arithmetic can then be carried out using operations on limbs. To calculate limb size, assume the platform that the field operations will be implemented has $n-$bit CPU register size. Let f be an element of prime field $\mathbb{F}_p$ where p is a $k-$bit prime number. Define t as the number of limbs where $\lceil k/t \rceil < n$. We can split f as follows:

$$f = [f_0/\lceil k/t \rceil, f_1/\lceil k/t \rceil, \ldots, f_{t-2}/\lceil k/t \rceil, f_{t-1}/m] \tag{3.1}$$

where $m = k - \lceil k/t \rceil (t-1)$, $0 < m < n$ and f_i/b indicates the limb f_i has $b-$bit length.

For the parameters that satisfy the condition in (3.1), we get a TMVP suitable matrix representation. We expect the coefficients in the matrix as small as possible.

To define the upperbound for the coefficient size, assume we have same parameters as (3.1) with at most $2t - 1$ different coefficients in the matrix representation. Let c_i be a cofficient in the matrix with a length of $b-$bit. For each different coefficient c_i in the matrix, the following condition should be met:

$$2\,(n - \lceil k/t \rceil - 1) \leq b \tag{3.2}$$

Using (3.1) we can decide if the representation can be implemented on a specific platform and we can define an upperbound for the coefficients using the (3.2). If the bound is exceeded, field operation can not be implemented using the defined CPU register size. (3.1) and (3.2) form the essential criterias to choose proper prime fields. Hence, we can choose proper parameters to build a representation that is in Toeplitz matrix form.

3.3.2 Prime Number Forms

There are different prime forms such as Mersenne primes ($2^k - 1$), Crandall primes ($2^k - c$), Special Montgomery primes ($2^k c - 1$), Montgomery-friendly primes ($2^k(2^l - c) - 1$), Solinas primes ($2^k - 2^l \pm \cdots \pm 1$) etc.
Considering (3.2), we expect to have coefficients as small as possible to use much more space in registers to minimize redundancy. Using Mersenne primes would result in getting smallest possible coefficients as the subtraction part of the prime form is fixed to 1. This would be useful but the reduces flexibility of choosing primes.
However, Crandall primes have flexibility with the parameter c. Considering 3.2, primes with $c < 2^{10}$ [10] might have performance similar or close to Mersenne primes. Thus, this property makes Crandall primes a good choice for our case.

3.3.3 New Prime Fields

Using (3.1) and (3.2), we have searched for suitable prime fields with different sizes using `MAGMA` software [36]. We have focused on $32-$bit and $64-$bit implementations and have decided to work on the following prime fields.

1. $\mathbb{F}_{2^{266}-3}$ [21] with 10 terms on $32-$bit platform.
2. $\mathbb{F}_{2^{336}-3}$ [32] with 13 terms on $32-$bit platform.
3. $\mathbb{F}_{2^{452}-3}$ with 10 terms on $64-$bit platform.
4. $\mathbb{F}_{2^{545}-3}$ with 10 terms on $64-$bit platform.
5. $\mathbb{F}_{2^{550}-5}$ with 10 terms on $64-$bit platform.
6. $\mathbb{F}_{2^{607}-1}$ [27] with 10 terms on $64-$bit platform.

Any prime field multiplication operation that is represented with a Toeplitz matrix form with limb size $t = 10$ according to (3.1) can be implemented efficiently using the method introduced in Section 3.2.

3.3.3.1 Representation for $\mathbb{F}_{2^{266}-3}$

Let f be an element of prime field $\mathbb{F}_{2^{266}-3}$. Using (3.1) with the parameters $n = 32, k = 266, t = 10$, we get the following representation:

$$f = [f_0/27, f_1/27, f_2/27, f_3/27, f_4/27, f_5/27, f_6/27, f_7/27, f_8/27, f_9/23]$$

We can also represent the element as a polynomial as follows:

$$f(x) = f_0 + 2^{27}f_1x + 2^{54}f_2x^2 + 2^{81}f_3x^3 + 2^{108}f_4x^4 \\ + 2^{135}f_5x^5 + 2^{162}f_6x^6 + 2^{189}f_7x^7 + 2^{216}f_8x^8 + 2^{243}f_9x^9$$

Let f, g and h in $\in \mathbb{F}_{2^{266}-3}$, we can compute the $f \cdot g = h$ with the following matrix representation:

$$\begin{bmatrix} g_0 & 48g_9 & 48g_8 & 48g_7 & 48g_6 & 48g_5 & 48g_4 & 48g_3 & 48g_2 & 48g_1 \\ g_1 & g_0 & 48g_9 & 48g_8 & 48g_7 & 48g_6 & 48g_5 & 48g_4 & 48g_3 & 48g_2 \\ g_2 & g_1 & g_0 & 48g_9 & 48g_8 & 48g_7 & 48g_6 & 48g_5 & 48g_4 & 48g_3 \\ g_3 & g_2 & g_1 & g_0 & 48g_9 & 48g_8 & 48g_7 & 48g_6 & 48g_5 & 48g_4 \\ g_4 & g_3 & g_2 & g_1 & g_0 & 48g_9 & 48g_8 & 48g_7 & 48g_6 & 48g_5 \\ g_5 & g_4 & g_3 & g_2 & g_1 & g_0 & 48g_9 & 48g_8 & 48g_7 & 48g_6 \\ g_6 & g_5 & g_4 & g_3 & g_2 & g_1 & g_0 & 48g_9 & 48g_8 & 48g_7 \\ g_7 & g_6 & g_5 & g_4 & g_3 & g_2 & g_1 & g_0 & 48g_9 & 48g_8 \\ g_8 & g_7 & g_6 & g_5 & g_4 & g_3 & g_2 & g_1 & g_0 & 48g_9 \\ g_9 & g_8 & g_7 & g_6 & g_5 & g_4 & g_3 & g_2 & g_1 & g_0 \end{bmatrix} \cdot \begin{bmatrix} f_0 \\ f_1 \\ f_2 \\ f_3 \\ f_4 \\ f_5 \\ f_6 \\ f_7 \\ f_8 \\ f_9 \end{bmatrix} = \begin{bmatrix} h_0 \\ h_1 \\ h_2 \\ h_3 \\ h_4 \\ h_5 \\ h_6 \\ h_7 \\ h_8 \\ h_9 \end{bmatrix}$$

3.3.3.2 Representation for $\mathbb{F}_{2^{545}-3}$

Let f be an element of prime field $\mathbb{F}_{2^{545}-3}$. Using (3.1) with the parameters $n = 64, k = 545, t = 10$, we get the following representation:

$$f = [f_0/55, f_1/55, f_2/55, f_3/55, f_4/55, f_5/55, f_6/55, f_7/55, f_8/55, f_9/50]$$

We can also represent the element as a polynomial as follows:

$$f(x) = f_0 + 2^{55}f_1x + 2^{110}f_2x^2 + 2^{165}f_3x^3 + 2^{220}f_4x^4 \\ + 2^{275}f_5x^5 + 2^{330}f_6x^6 + 2^{385}f_7x^7 + 2^{440}f_8x^8 + 2^{495}f_9x^9$$

Let f, g and h in $\in \mathbb{F}_{2^{545}-3}$, we can compute the $f \cdot g = h$ with the following matrix representation:

$$\begin{bmatrix} g_0 & 96g_9 & 96g_8 & 96g_7 & 96g_6 & 96g_5 & 96g_4 & 96g_3 & 96g_2 & 96g_1 \\ g_1 & g_0 & 96g_9 & 96g_8 & 96g_7 & 96g_6 & 96g_5 & 96g_4 & 96g_3 & 96g_2 \\ g_2 & g_1 & g_0 & 96g_9 & 96g_8 & 96g_7 & 96g_6 & 96g_5 & 96g_4 & 96g_3 \\ g_3 & g_2 & g_1 & g_0 & 96g_9 & 96g_8 & 96g_7 & 96g_6 & 96g_5 & 96g_4 \\ g_4 & g_3 & g_2 & g_1 & g_0 & 96g_9 & 96g_8 & 96g_7 & 96g_6 & 96g_5 \\ g_5 & g_4 & g_3 & g_2 & g_1 & g_0 & 96g_9 & 96g_8 & 96g_7 & 96g_6 \\ g_6 & g_5 & g_4 & g_3 & g_2 & g_1 & g_0 & 96g_9 & 96g_8 & 96g_7 \\ g_7 & g_6 & g_5 & g_4 & g_3 & g_2 & g_1 & g_0 & 96g_9 & 96g_8 \\ g_8 & g_7 & g_6 & g_5 & g_4 & g_3 & g_2 & g_1 & g_0 & 96g_9 \\ g_9 & g_8 & g_7 & g_6 & g_5 & g_4 & g_3 & g_2 & g_1 & g_0 \end{bmatrix} \cdot \begin{bmatrix} f_0 \\ f_1 \\ f_2 \\ f_3 \\ f_4 \\ f_5 \\ f_6 \\ f_7 \\ f_8 \\ f_9 \end{bmatrix} = \begin{bmatrix} h_0 \\ h_1 \\ h_2 \\ h_3 \\ h_4 \\ h_5 \\ h_6 \\ h_7 \\ h_8 \\ h_9 \end{bmatrix}$$

A special case for $\mathbb{F}_{2^{545}-3}$ is radix-$2^{54.5}$ representation. This representation with $64-$bit CPU architecture has similar matrix form in with Curve25519's radix-$2^{25.5}$ representation [4] with $32-$bit CPU architecture. Limbs can be represented as follows using radix-$2^{54.5}$ representation:

$$f = [f_0/55, f_1/54, f_2/55, f_3/54, f_4/55, f_5/54, f_6/55, f_7/54, f_8/55, f_9/54]$$

We can also represent the element as a polynomial as follows:

$$f(x) = f_0 + 2^{55}f_1x + 2^{109}f_2x^2 + 2^{164}f_3x^3 + 2^{218}f_4x^4 \\ + 2^{273}f_5x^5 + 2^{327}f_6x^6 + 2^{382}f_7x^7 + 2^{436}f_8x^8 + 2^{491}f_9x^9$$

In this case, we can compute the $f \cdot g = h$ with the following matrix representation:

$$\begin{bmatrix} g_0 & 6g_9 & 3g_8 & 6g_7 & 3g_6 & 6g_5 & 3g_4 & 6g_3 & 3g_2 & 6g_1 \\ g_1 & g_0 & 3g_9 & 3g_8 & 3g_7 & 3g_6 & 3g_5 & 3g_4 & 3g_3 & 3g_2 \\ g_2 & 2g_1 & g_0 & 6g_9 & 3g_8 & 6g_7 & 3g_6 & 6g_5 & 3g_4 & 6g_3 \\ g_3 & g_2 & g_1 & g_0 & 3g_9 & 3g_8 & 3g_7 & 3g_6 & 3g_5 & 3g_4 \\ g_4 & 2g_3 & g_2 & 2g_1 & g_0 & 6g_9 & 3g_8 & 6g_7 & 3g_6 & 6g_5 \\ g_5 & g_4 & g_3 & g_2 & g_1 & g_0 & 3g_9 & 3g_8 & 3g_7 & 3g_6 \\ g_6 & 2g_5 & g_4 & 2g_3 & g_2 & 2g_1 & g_0 & 6g_9 & 3g_8 & 6g_7 \\ g_7 & g_6 & g_5 & g_4 & g_3 & g_2 & g_1 & g_0 & 3g_9 & 3g_8 \\ g_8 & 2g_7 & g_6 & 2g_5 & g_4 & 2g_3 & g_2 & 2g_1 & g_0 & 6g_9 \\ g_9 & g_8 & g_7 & g_6 & g_5 & g_4 & g_3 & g_2 & g_1 & g_0 \end{bmatrix} \cdot \begin{bmatrix} f_0 \\ f_1 \\ f_2 \\ f_3 \\ f_4 \\ f_5 \\ f_6 \\ f_7 \\ f_8 \\ f_9 \end{bmatrix} = \begin{bmatrix} h_0 \\ h_1 \\ h_2 \\ h_3 \\ h_4 \\ h_5 \\ h_6 \\ h_7 \\ h_8 \\ h_9 \end{bmatrix}$$

This representation allows to mount any field operation improvement that is made for $\mathbb{F}_{2^{255}-19}$ on 32−bit to $\mathbb{F}_{2^{545}-3}$ on 64−bit.

3.3.3.3 Representation for $\mathbb{F}_{2^{550}-5}$

Let f be an element of prime field $\mathbb{F}_{2^{550}-5}$. Using (3.1) with the parameters $n = 64, k = 550, t = 10$, we get the following representation:

$$f = [f_0/55, f_1/55, f_2/55, f_3/55, f_4/55, f_5/55, f_6/55, f_7/55, f_8/55, f_9/55]$$

We can also represent the element as a polynomial as follows:

$$\begin{aligned} f(x) = f_0 + 2^{55} f_1 x + 2^{110} f_2 x^2 + 2^{165} f_3 x^3 + 2^{220} f_4 x^4 \\ + 2^{275} f_5 x^5 + 2^{330} f_6 x^6 + 2^{385} f_7 x^7 + 2^{440} f_8 x^8 + 2^{495} f_9 x^9 \end{aligned}$$

Note that the polynomial representation is same as it is in $\mathbb{F}_{2^{545}-3}$. Besides, this TMVP suitable form is also a radix-2^{55} representation, there is no redundancy on the most significant limb.

Let f, g and h be in $\in \mathbb{F}_{2^{550}-5}$. We can compute the $f \cdot g = h$ with the following matrix representation:

$$\begin{bmatrix} g_0 & 5g_9 & 5g_8 & 5g_7 & 5g_6 & 5g_5 & 5g_4 & 5g_3 & 5g_2 & 5g_1 \\ g_1 & g_0 & 5g_9 & 5g_8 & 5g_7 & 5g_6 & 5g_5 & 5g_4 & 5g_3 & 5g_2 \\ g_2 & g_1 & g_0 & 5g_9 & 5g_8 & 5g_7 & 5g_6 & 5g_5 & 5g_4 & 5g_3 \\ g_3 & g_2 & g_1 & g_0 & 5g_9 & 5g_8 & 5g_7 & 5g_6 & 5g_5 & 5g_4 \\ g_4 & g_3 & g_2 & g_1 & g_0 & 5g_9 & 5g_8 & 5g_7 & 5g_6 & 5g_5 \\ g_5 & g_4 & g_3 & g_2 & g_1 & g_0 & 5g_9 & 5g_8 & 5g_7 & 5g_6 \\ g_6 & g_5 & g_4 & g_3 & g_2 & g_1 & g_0 & 5g_9 & 5g_8 & 5g_7 \\ g_7 & g_6 & g_5 & g_4 & g_3 & g_2 & g_1 & g_0 & 5g_9 & 5g_8 \\ g_8 & g_7 & g_6 & g_5 & g_4 & g_3 & g_2 & g_1 & g_0 & 5g_9 \\ g_9 & g_8 & g_7 & g_6 & g_5 & g_4 & g_3 & g_2 & g_1 & g_0 \end{bmatrix} \cdot \begin{bmatrix} f_0 \\ f_1 \\ f_2 \\ f_3 \\ f_4 \\ f_5 \\ f_6 \\ f_7 \\ f_8 \\ f_9 \end{bmatrix} = \begin{bmatrix} h_0 \\ h_1 \\ h_2 \\ h_3 \\ h_4 \\ h_5 \\ h_6 \\ h_7 \\ h_8 \\ h_9 \end{bmatrix}$$

Note that, all three matrix representations are suitable for TMVP and satisfy (3.2).

CHAPTER 4

SPEEDING UP CURVE25519

4.1 Curve25519

The Curve25519 [4] function is a $x-$coordinate only scalar multiplication on $E(\mathbb{F}_p)$, where p is the prime number $2^{255} - 19$ and E is the elliptic curve:

$$y^2 = x^3 + 486662x^2 + x$$

4.1.1 The radix-$2^{25.5}$ Representation for $\mathbb{F}_{2^{255}-19}$

According to Bernstein's representation [4], assume we want to compute $f \cdot g = h$ where f, g and h in $\in \mathbb{F}_{2^{255}-19}$. The integers are splitted into limbs using radix-$2^{25.5}$ representation as follows [4]:

$$f = [f_0/26, f_1/25, f_2/26, f_3/25, f_4/26, f_5/25, f_6/26, f_7/25, f_8/26, f_9/25]$$
$$g = [g_0/26, g_1/25, g_2/26, g_3/25, g_4/26, g_5/25, g_6/26, g_7/25, g_8/26, g_9/25]$$
$$h = [h_0/26, h_1/25, h_2/26, h_3/25, h_4/26, h_5/25, h_6/26, h_7/25, h_8/26, h_9/25]$$

Thus, any element of the field can be represented as the concatenation of the limbs as follows:

$$f = f_9||f_8||f_7||f_6||f_5||f_4||f_3||f_2||f_1||f_0$$
$$g = g_9||g_8||g_7||g_6||g_5||g_4||g_3||g_2||g_1||g_0$$
$$h = h_9||h_8||h_7||h_6||h_5||h_4||h_3||h_2||h_1||h_0$$

where the operation $||$ is the bitwise concatenation and f_0, g_0 and h_0 are the least significant **limb**s. We represent the elements as polynomials with coefficients indicating the cumulative bit-size of the previous limbs:

$$\begin{aligned}
f(x) =& f_0 + 2^{26} f_1 x + 2^{51} f_2 x^2 + 2^{77} f_3 x^3 + 2^{102} f_4 x^4 + 2^{128} f_5 x^5 + \\
& 2^{153} f_6 x^6 + 2^{179} f_7 x^7 + 2^{204} f_8 x^8 + 2^{230} f_9 x^9 \\
g(x) =& g_0 + 2^{26} g_1 x + 2^{51} g_2 x^2 + 2^{77} g_3 x^3 + 2^{102} g_4 x^4 + 2^{128} g_5 x^5 + \\
& 2^{153} g_6 x^6 + 2^{179} g_7 x^7 + 2^{204} g_8 x^8 + 2^{230} g_9 x^9 \\
h(x) =& h_0 + 2^{26} h_1 x + 2^{51} h_2 x^2 + 2^{77} h_3 x^3 + 2^{102} h_4 x^4 + 2^{128} h_5 x^5 + \\
& 2^{153} h_6 x^6 + 2^{179} h_7 x^7 + 2^{204} h_8 x^8 + 2^{230} h_9 x^9
\end{aligned}$$

As a result, each polynomial represents its value at 1. We get the following matrix representation:

$$\begin{bmatrix}
g_0 & 38g_9 & 19g_8 & 39g_7 & 19g_6 & 38g_5 & 19g_4 & 38g_3 & 19g_2 & 38g_1 \\
g_1 & g_0 & 19g_9 & 19g_8 & 19g_7 & 19g_6 & 19g_5 & 19g_4 & 19g_3 & 19g_2 \\
g_2 & 2g_1 & g_0 & 38g_9 & 19g_8 & 38g_7 & 19g_6 & 38g_5 & 19g_4 & 38g_3 \\
g_3 & g_2 & g_1 & g_0 & 19g_9 & 19g_8 & 19g_7 & 19g_6 & 19g_5 & 19g_4 \\
g_4 & 2g_3 & g_2 & 2g_1 & g_0 & 38g_9 & 19g_8 & 38g_7 & 19g_6 & 38g_5 \\
g_5 & g_4 & g_3 & g_2 & g_1 & g_0 & 19g_9 & 19g_8 & 19g_7 & 19g_6 \\
g_6 & 2g_5 & g_4 & 2g_3 & g_2 & 2g_1 & g_0 & 38g_9 & 19g_8 & 38g_7 \\
g_7 & g_6 & g_5 & g_4 & g_3 & g_2 & g_1 & g_0 & 19g_9 & 19g_8 \\
g_8 & 2g_7 & g_6 & 2g_5 & g_4 & 2g_3 & g_2 & 2g_1 & g_0 & 38g_9 \\
g_9 & g_8 & g_7 & g_6 & g_5 & g_4 & g_3 & g_2 & g_1 & g_0
\end{bmatrix} \cdot \begin{bmatrix} f_0 \\ f_1 \\ f_2 \\ f_3 \\ f_4 \\ f_5 \\ f_6 \\ f_7 \\ f_8 \\ f_9 \end{bmatrix} = \begin{bmatrix} h_0 \\ h_1 \\ h_2 \\ h_3 \\ h_4 \\ h_5 \\ h_6 \\ h_7 \\ h_8 \\ h_9 \end{bmatrix}$$

Basic schoolbook method is used to compute the h. Thus, cost of computing h is $100\mathbf{M}+81\mathbf{A_d}$ and $1\mathbf{M}+11\mathbf{A_d}$ for final reduction to get reduced coefficiens of h, namely, $h_0, \ldots, h_9$. Note that, shift and masking (bitwise AND) operation costs are ignored. Hence, total cost is;

$$101\mathbf{M} + 92\mathbf{A_d}.$$

4.2 Multiplication Over $\mathbb{F}_{2^{255}-19}$ Using TMVP

The radix-$2^{25.5}$ representation which is described in Section 4.1.1, is not suitable for mounting TMVP operations since the matrix is not in Toeplitz form.

Using (3.1) and (3.2) with the parameters $n = 32, k = 255, t = 10$, we get the following matrix representation which is in Toeplitz form. The representations of f, g and h are defined as follows:

$$\begin{aligned}
f =&[f_0/26, f_1/26, f_2/26, f_3/26, f_4/26, f_5/26, f_6/26, f_7/26, f_8/26, f_9/21]\\
g =&[g_0/26, g_1/26, g_2/26, g_3/26, g_4/26, g_5/26, g_6/26, g_7/26, g_8/26, g_9/21]\\
h =&[h_0/26, h_1/26, h_2/26, h_3/26, h_4/26, h_5/26, h_6/26, h_7/26, h_8/26, h_9/21]
\end{aligned}$$

We represent the elements as polynomials with coefficients indicating the cumulative bit-size of the previous limbs as follows:

$$\begin{aligned}
f(x) =&f_0 + 2^{26}f_1x + 2^{52}f_2x^2 + 2^{78}f_3x^3 + 2^{104}f_4x^4 + 2^{130}f_5x^5 +\\
&2^{156}f_6x^6 + 2^{182}f_7x^7 + 2^{208}f_8x^8 + 2^{234}f_9x^9\\
g(x) =&g_0 + 2^{26}g_1x + 2^{52}g_2x^2 + 2^{78}g_3x^3 + 2^{104}g_4x^4 + 2^{130}g_5x^5 +\\
&2^{156}g_6x^6 + 2^{182}g_7x^7 + 2^{208}g_8x^8 + 2^{234}g_9x^9\\
h(x) =&h_0 + 2^{26}h_1x + 2^{52}h_2x^2 + 2^{78}h_3x^3 + 2^{104}h_4x^4 + 2^{130}h_5x^5 +\\
&2^{156}h_6x^6 + 2^{182}h_7x^7 + 2^{208}h_8x^8 + 2^{234}h_9x^9
\end{aligned}$$

We get the following matrix representation for the new field multiplication method:

$$\begin{bmatrix}
g_0 & 608g_9 & 608g_8 & 608g_7 & 608g_6 & 608g_5 & 608g_4 & 608g_3 & 608g_2 & 608g_1\\
g_1 & g_0 & 608g_9 & 608g_8 & 608g_7 & 608g_6 & 608g_5 & 608g_4 & 608g_3 & 608g_2\\
g_2 & g_1 & g_0 & 608g_9 & 608g_8 & 608g_7 & 608g_6 & 608g_5 & 608g_4 & 608g_3\\
g_3 & g_2 & g_1 & g_0 & 608g_9 & 608g_8 & 608g_7 & 608g_6 & 608g_5 & 608g_4\\
g_4 & g_3 & g_2 & g_1 & g_0 & 608g_9 & 608g_8 & 608g_7 & 608g_6 & 608g_5\\
g_5 & g_4 & g_3 & g_2 & g_1 & g_0 & 608g_9 & 608g_8 & 608g_7 & 608g_6\\
g_6 & g_5 & g_4 & g_3 & g_2 & g_1 & g_0 & 608g_9 & 608g_8 & 608g_7\\
g_7 & g_6 & g_5 & g_4 & g_3 & g_2 & g_1 & g_0 & 608g_9 & 608g_8\\
g_8 & g_7 & g_6 & g_5 & g_4 & g_3 & g_2 & g_1 & g_0 & 608g_9\\
g_9 & g_8 & g_7 & g_6 & g_5 & g_4 & g_3 & g_2 & g_1 & g_0
\end{bmatrix}
\cdot
\begin{bmatrix} f_0\\ f_1\\ f_2\\ f_3\\ f_4\\ f_5\\ f_6\\ f_7\\ f_8\\ f_9 \end{bmatrix}
=
\begin{bmatrix} h_0\\ h_1\\ h_2\\ h_3\\ h_4\\ h_5\\ h_6\\ h_7\\ h_8\\ h_9 \end{bmatrix}$$

Please note that, using this representation with schoolbook method does not make sense since the constant coefficient here is 608 (a 10-bit integer) and the limbs wouldn't fit into registers. But, total result fits in registers. Thus, this representation can work and be useful. This is where the TMVP has advantages and makes this presentation possible to be implemented in an efficient way. To achieve this, we can split the matrix above into chunks and handle each part as follows:

$$\left(\left[\begin{array}{ccccc|ccccc}
g_0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0\\
g_1 & g_0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0\\
g_2 & g_1 & g_0 & 0 & 0 & 0 & 0 & 0 & 0 & 0\\
g_3 & g_2 & g_1 & g_0 & 0 & 0 & 0 & 0 & 0 & 0\\
g_4 & g_3 & g_2 & g_1 & g_0 & 0 & 0 & 0 & 0 & 0\\
\hline
g_5 & g_4 & g_3 & g_2 & g_1 & g_0 & 0 & 0 & 0 & 0\\
g_6 & g_5 & g_4 & g_3 & g_2 & g_1 & g_0 & 0 & 0 & 0\\
g_7 & g_6 & g_5 & g_4 & g_3 & g_2 & g_1 & g_0 & 0 & 0\\
g_8 & g_7 & g_6 & g_5 & g_4 & g_3 & g_2 & g_1 & g_0 & 0\\
g_9 & g_8 & g_7 & g_6 & g_5 & g_4 & g_3 & g_2 & g_1 & g_0
\end{array}\right] + 608\left[\begin{array}{ccccc|ccccc}
0 & g_9 & g_8 & g_7 & g_6 & g_5 & g_4 & g_3 & g_2 & g_1\\
0 & 0 & g_9 & g_8 & g_7 & g_6 & g_5 & g_4 & g_3 & g_2\\
0 & 0 & 0 & g_9 & g_8 & g_7 & g_6 & g_5 & g_4 & g_3\\
0 & 0 & 0 & 0 & g_9 & g_8 & g_7 & g_6 & g_5 & g_4\\
0 & 0 & 0 & 0 & 0 & g_9 & g_8 & g_7 & g_6 & g_5\\
\hline
0 & 0 & 0 & 0 & 0 & 0 & g_9 & g_8 & g_7 & g_6\\
0 & 0 & 0 & 0 & 0 & 0 & 0 & g_9 & g_8 & g_7\\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & g_9 & g_8\\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & g_9\\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{array}\right]\right) \cdot \left[\begin{array}{c}
f_0\\ f_1\\ f_2\\ f_3\\ f_4\\ \hline f_5\\ f_6\\ f_7\\ f_8\\ f_9
\end{array}\right]$$

To compute this, one can use the strategy introduced in Section 3.2. The only extra part is the multiplication with the constant. Using the advantage of splitting the matrix into upper and lower triangular matrices, we can easily handle the multiplication operation with 608 at the end of the computation. Note that, 608 can be decomposed as follows:

$$608 = 2^5 \cdot 19 = 2^5(2^4 + 2^1 + 1) = 2^9 + 2^6 + 2^5$$

So, to multiply an integer x with 608, one can use 3 **shift**s and 2$\mathtt{A_d}$:

$$x \cdot 608 = (x << 9) + (x << 6) + (x << 5)$$

The cost of the shift operation can be ignored and the total cost for computing a total of 10 multiplications by 608 will cost extra 20$\mathtt{A_d}$ operations. Hence, the total cost will be as described in Table 3.2 with an extra 20$\mathtt{A_d}$ operations.

4.3 Implementation and Benchmark Results

Our algoritm is based on 32-bit word size, thus, we focused on 32-bit implementations. To benchmark our algorithm, we have used different platforms including x86 and 32-bit ARM and compiled it on each platform with different configurations.

For x86 implementation, we use macOS Sierra with Apple LLVM gcc 4.2.1 compiler on Intel i7-4750HQ CPU with 16GB RAM and for ARM implementation, we use Raspbian 8 with gcc 4.9.2 on Raspberry Pi 2 with ARMv7l CPU and 1GB RAM.

We have implemented our field multiplication algorithm using C programming language and compared our results with the finite field multiplication function `fe_mul` of `ref10` [37] implementation.

We have compiled the finite field multiplication functions for both implementations on each platform using every optimization level of gcc and executed each result multiple times and got an avarage timing for each execution. Our results are better at optimization level 2 and 3 [18] on x86 implementation up to $\%13$ and has close timing results on ARM.

Table 4.1: Implementation Benchmark Comparison Over $\mathbb{F}_{2^{255}-19}$

Platform	**Algorithm**	**Opt. Level**	**Timing (Avg.)**
x86	Proposed Method #1	$-O2$	$75ns$
x86	ref10	$-O2$	$80ns$
x86	Proposed Method #1	$-O3$	$70ns$
x86	ref10	$-O3$	$80ns$
ARMv7l	Proposed Method #1	$-O2$	$2600ns$
ARMv7l	ref10	$-O2$	$2600ns$

We also give implementation cycles of the algortihms as an extended result in Table 4.2. The results are obtained with gcc 4.2.1 compiler using optimization level 3 and `fast-math` and `no-common` options enabled on Intel i7-4750HQ with Hyper Threading disabled. The counts listed in the table are obtained by taking a minimum of 10^7 operations for each field multiplication algorithm. TT means TMVP and TMVP, TSB means TMVP and Schoolbook for 4×4 and 2×2 matrix operations respectively.

Table 4.2: Field Multiplication Implementation Cycles Over $\mathbb{F}_{2^{255}-19}$

Implementation	**Reference**	**Cycles**	**Comparison to TSB**
`ref10`	[4, 37]	104.94	-12.59%
Proposed Method #1 (TT)	This Thesis, [35]	93.81	-2.22%
Proposed Method #2 (TSB)	This Thesis, [35]	91.73	

CHAPTER 5

ELLIPTIC CURVES SEARCH

5.1 Choosing Safe Elliptic Curves

In Chapter 3, we have introduced proper prime fields and in this chapter we discuss how to find safe elliptic curves based on those prime fields. To accomplish this, we first introduce known attacks against against elliptic curves and give our safe curve search process details. The search codes and process details mentioned in this chapter are given in Appendix B.

5.1.1 Curve Parameters

To define curve parameters, it is vital to define the underlying prime field first. As we discussed on previous sections, we already have our prime fields defined. Therefore, we should consider our parameters to define a curve.

In this thesis, we focus on Montgomery curves with B fixed to 1 and with $x-$coordinate only representation. After defining the curve, one should specify a base point $B = (x_1, y_1)$ on the curve with a prime order. We expect to find a base point with x_1 value as small as possible.

5.1.2 Attacks

When the parameters are fixed to define an elliptic curve, attacks against the curves should be tested to verify if the selected parameters satisfy the security requirements.

For this purpose, we follow the SafeCurves project [8] which offers an elegant approach for addressing different point of views.

Assume we have an elliptic curve $E_M^{A,1}$ defined over prime field $\mathbb{F}_p$ with a base point $B = (x_1, y_1)$ where order of the base point is k, order of the curve "$n = tk$", such that k is prime and 4 divides t. The curves should be tested against at least the attacks defined below.

5.1.2.1 Pollard's Rho Attack

[30] is the best known method for solving the elliptic curve discrete logarithm problem [25] which has exponential time complexity $\mathcal{O}(\sqrt{n})$ where n is the order of the elliptic curve. But the improved attacks [9] reduces the complexity around $\left(\sqrt{\frac{\pi k}{4}}\right) \approx 0.8862\sqrt{k}$ additions where k is the order of the base point. Thus, we expect to have this value at least 2^{100} as described in SafeCurves. However, for the sake of being lightweight and short-term key usage, this limit can be streched down to around 2^{80} which also allows to find new alternative curves.

5.1.2.2 Small-subgroup Attack

Our curve choice assumption have order $n = tk$, where k is the large prime order of the specified base point B and t is a small cofactor. The possible orders of curve points can be divisors of t and k times divisors of t.

Assume Alice and Bob are exchanging keys with an ECDH protocol and an attacker impersonated Alice is sending a point Q with small order instead of Alice's legitimate public key to Bob. Then, Bob computes bQ where b is Bob's secret key and reveal the result to complete key exchange. Since Q has small order, the attacker can check all possibilites for b under the modulo of order Q which possibly results in finding the correct value for b modulo order of Q.

To protect the curve against these types of attack, one can choose base point's order as a prime and equals to curve order but this is not possible in our case since 4 divides t due to Montgomery form. Another protection is choosing b as a multiple of t such that $b = tr$ for a random r mod k. At worst case, this attack reduces security at most $\lceil \log_2(t) \rceil$-bit [8].

5.1.2.3 Twist Security

Twist security is first introduced in [4] for the special case when the curve form is Montgomery with x-coordinate only representation. Considering any point $(x_i, y_i) \in \mathbb{F}_p$, around half of the x_i values are in $\mathbb{F}_p$ and the remaining half of the points correspond to a point on the quadratic twist of the $\mathbb{F}_p$ [14]. Instead of checking if a point corresponds to the curve itself of its quadratic twist every time, choosing a safe twist is a much more elegant way for defining a safe curve. Thus, while searching for new curves, every check that is done for the curve itself should be repeated for its twist.

5.1.2.4 Other Attacks

As well as the attacks mentioned here, SafeCurves lists some other attack parameters to consider such as additive and multiplicative transfers, multiplicative embedding degree, invalid curve attacks and its combination with small-subgroup attacks and complex-multiplication field discriminant etc.

Our curve selection method covers only attacks mentioned in this chapter. Readers are advised to check SafeCurves [8] for details.

5.1.3 New Curve Choice Parameter

We introduce a new parameter called Order Type and define types from 1 to 5 in Table 5.1:

Table 5.1: Order Type Parameter Details

Order Type	**Cofactor for Curve Order**	**Cofactor for Twist Order**	**Example Curves**
Type 1	4	4	[10], [19], E-382, E-521
Type 2	4 or 8	4 or 8	This Thesis (Curve2663), Curve383187, [4], [6]
Type 3	4d	4 or 8	-
Type 4	4 or 8	4d	-
Type 5	4d	4d	-

Order Type 1 curves are the hardest to discover but the safest ones against Pollard's Rho attack. Most of the popular Edwards or Montgomery curves are in the class of Order Type 2 such as Curve25519 etc.

Type 3, Type 4 and Type 5 curves have at least one order cofactor as $4d$ where d is an arbitrary integer. Type 5 elliptic curves have the most relaxed cofactors with both curve itself and its twist can have order cofactos upto $4d$.

Relaxing the parameters enables finding new curves but affects the security margin because of Pollard's Rho and Small-subgroup attack's cost. Hence, this trade off makes sense to find new alternative curves without compromising the expected security level. Here, we expect to have d as small as possible and also expect smallest number of different primes in its canonical factorization. Especially, for high-security curves (where security margins are bigger than 256-bit) having flexible d values could allow to discover new elliptic curves.

5.1.4 The Proposed Safe Curve Generation Process

We introduce a safe curve generation algorithm which focuses on the criterias defined above. The algorithm is given in Algorithm 6.

5.2 New Curve Parameters

5.2.1 Curve2663

We follow the popular naming convention and use Curve2663 for our proposed curve name. Following the algorithm described above, we find the Montgomery curve

$$E_M^{20710,1} : y^2 = x^3 + 20710x^2 + x$$

defined over prime field $\mathbb{F}_{2^{266}-3}$ with order $2^3 k$ where k is a prime.

$$k = 2^{262} + 74106937111882365071085430405560261 0\backslash$$
$$2607498625779807844927895183684117667522393$$

Algorithm 6 Safe Curve Search

```
Choose a big prime $p \equiv 1 \bmod 4$ or $p \equiv 3 \bmod 4$.
**if** Not IsPrime($p$) **then return** false
**end if**
Use Equations 3.1 and 3.2 to check if the prime field $\mathbb{F}_p$ is suitable for TMVP-Based field arithmetic.
Choose a Montgomery curve $y^2 = x^3 + Ax^2 + x$, where $A^2 - 4$ is nonzero in $\mathbb{F}_p$.
Choose a base point $B = (x_1, y_1)$ of prime order $k$ on the curve.
Check if $(x_1, y_1)$ is on the curve.
Check if $k$ is prime.
Check if $kB = 0$.
Check the cost for Pollard's rho attack is above $2^{100}$, i.e. $0.8862\sqrt{k} > 2^{100}$.
Check if the security against the twist attacks is above $2^{100}$.
Check the cost for Pollard's rho attack against the twist is above $2^{100}$, i.e. $0.8862\sqrt{k'} > 2^{100}$.
**if** All checks above are satisfied **then return** $E_M^{A,1} : y^2 = x^3 + Ax^2 + x$ defined over $\mathbb{F}_p$ with base point $B = (x_1, y_1)$.
**else**
    Select a new A value and check the conditions again.
**end if**
```

Simirlarly, twist of the curve has order $2^2k'$ where k' is a prime.

$$k' = 2^{264} + 35607856423765071947803856446452588867021$$

Base Point $B = (x_1, y_1)$ has order k with x-coordinate equals 17.

$$(x_1, y_1) = (17, 94350722641181309376645675877820961\backslash \\ 23496581348891673081138291902948925782376305 4)$$

The cost for Pollard's rho attack is around $2^{131.3257}$ and for Twist's is around $2^{131.8257}$. The details for the search process are given in Appendix B.

5.3 Implementation and Benchmark Results

We have proposed the TMVP-Friendly representation for $\mathbb{F}_{2^{266}-3}$ in Section 3.3.3.1 and followed the strategy introduced in Section 3.2 for its implementation. The only extra part is the multiplication with the constant which is similarly discussed for $\mathbb{F}_{2^{255}-19}$ in Section 4.2. We can easily handle the multiplication operation with 48 at the end of the computation and 48 can be decomposed as follows:

$$48 = 2^5 + 2^4$$

Thus, to multiply an integer x with 48, one can use 2 **shift**s and $1\mathbf{A_d}$:

$$x \cdot 48 = (x << 5) + (x << 4)$$

The cost of the shift operation can be ignored and the total cost for computing 10 multiplications by 48 will cost extra $10\mathbf{A_d}$ operations. Hence, the total cost will be as described in Table 3.2 with an extra $10\mathbf{A_d}$ operations.

Source code for a 32-bit reference implementation of Curve2663 can be accessed online. [1]

The code is written in ANSI C which is portable. Thus, it can be compiled for any platform. Reference implementation contains five different field multiplication implementations. For three of them, we have followed the implementation strategies

[1] See `https://gitlab.com/hktaskin/curve2663` for the code.

Table 5.2: Field Multiplication Implementation Cycles

Implementation	**Reference**	**Cycles**	**Comparison to TT**
`ref10`	[4, 37]	119.07	-16.58%
`donna`	[4, 37]	108.43	-8.39%
Kummer	[27]	263.96	-62.37%
TT	This Thesis, [35]	99.33	
TSB	This Thesis, [35]	101.35	-1.99%

for `ref10` and `donna` implementations of Curve25519 [4] in SUPERCOP [37] on $\mathbb{F}_{2^{266}-3}$ and modified the implementation given in [27] to make it work with ANSI C. Besides these field multiplication implementations, we have implemented two different TMVP-Based implementations of the proposed methods using the strategy introduced in [35]. First implementation of us makes use of TMVP on both 4×4 and 2×2 matrix multiplications. The second implementation uses schoolbook matrix multiplication algorithm instead of TMVP for the 2×2 matrix multiplications.

The compiler for benchmarking we use is gcc [18] on macOS Sierra with Apple LLVM gcc 4.2.1 on 2.0 GHz Intel i7-4750HQ with Hyper Threading disabled. We have tested several different compiler optimization arguments and combinations to get faster results for all implementations. We have selected the compiler optimization level 3 along with `-ffast-math` and `-fno-common` options.

Table 5.2 shows the avarage implementation cycle counts of 5 different field multiplications. The counts listed are obtained by taking a minimum of 10^6 operations for each field multiplication algorithm. TT means TMVP and TMVP, TSB means TMVP and Schoolbook for 4×4 and 2×2 matrix operations respectively.

Our reference scalar multiplication implementation uses Montgomery Ladder to compute scalar multiples of points on the curve and only implements variable base point scalar multiplication. We follow the same strategy as described in [4]. For modular inversion, the algorithm is optimized using the fact $x^{-1} = x^{p-2}$ resulting in a constant time algorithm with the strategies introduced in [4, 27].

Table 5.3 shows the avarage implementation cycle counts of scalar multiplication with 5 different field multiplications. The counts listed are obtained by taking a minimum of $50K$ operations for each scalar multiplication algorithm. The Kummer implemen-

Table 5.3: Scalar Multiplication Implementation Cycles

Implementation	**Scalar Mult. Cycles**	**Comparison to TSB**
Ref10	319837	-14.56%
Donna	278875	-2.01%
Kummer	673724	-59.44%
TT	281392	-2.89%
TSB	273261	

tation focuses specifically on Intel AVX2 implementation, besides, our implementation strategy is making the code portable. In this respect, we have re-implemented the Kummer algorithm using ANSI C intrinsics. Thus, the drastic improvement ratio of our algorithm against Kummer implementation can be explained in this fashion.

CHAPTER 6

CONCLUSION

The need for faster and practical cryptography has been an attractive research area due to the fact that cryptographic operations are the most expensive part in an application in general. Especially, asymmetric operations are the slowest part. With the introduction of ECC, speeding up ECC operations became a focused research area.

ECC operations are based on finite field arithmetic. Therefore, in this thesis, we have focused on optimizations for finite field arithmetic, especially, finite field multiplication. Furthermore, we also gave details of choosing safe primes and proposed a new curve.

Overall, the contributions of this thesis can be summarized as follows:

In Chapter 3, we have presented the work related to TMVP and proposed a decompostion for 10-dimensional TMVP for field multiplication. We have searched for all possible combinations to find a new representation that is efficient, and the proposed algorithm is built on the new representation. Next, we have discussed the arithmetic cost of our proposed multiplication algorithm which resulted in better complexity compared to schoolbook multiplication. We have also evaluated the delay complexity of the proposed algorithm using four-core implementation to stress its efficiency when it is implemented in multi-core systems and showed that it can be implemented as almost embarrassingly parallel. Finally, we have introduced a method to build TMVP-Based element representation on prime fields and proposed some prime fields that have promising forms for implementation.

In Chapter 4, we have proposed a new algorithm for finite field multiplication over

$\mathbb{F}_{2^{255}-19}$ using TMVP method and implemented the proposed algorithm on different platforms including x86 and ARM. Next, we have tested and compared our implementation with different configurations on these platforms and showed that the proposed algorithm has promising results indicating TMVP method can be used efficiently for multiplication in $\mathbb{F}_{2^{255}-19}$.

In Chapter 5, we have introduced a methodology to search and choose safe curves and gave details of safe curve generation process. Next, we have defined a new criteria called "Order Type" for cofactor categorization of the curves. Finally, we have proposed a new curve called Curve2663 that we have found using the proposed safe curve generation algorithm and demonstrated its details and benchmark results.

REFERENCES

[1] S. Ali and M. Cenk, A new algorithm for residue multiplication modulo $2^{521}-1$, in *Proceedings of the 19th International Conference on Information Security and Cryptology Volume 10157*, ICISC 2016, pp. 181–193, Springer-Verlag New York, Inc., New York, NY, USA, 2017, ISBN 978-3-319-53176-2.

[2] S. Ali and M. Cenk, Faster residue multiplication modulo 521-bit mersenne prime and an application to ecc, IEEE Transactions on Circuits and Systems I: Regular Papers, 65(8), pp. 2477–2490, 2018.

[3] D. F. Aranha, P. S. L. M. Barreto, G. C. C. F. Pereira, and J. E. Ricardini, A note on high-security general-purpose elliptic curves, Cryptology ePrint Archive, Report 2013/647, 2013, `https://eprint.iacr.org/2013/647`.

[4] D. J. Bernstein, *Curve25519: New Diffie-Hellman Speed Records*, pp. 207–228, Springer Berlin Heidelberg, Berlin, Heidelberg, 2006, ISBN 978-3-540-33852-9.

[5] D. J. Bernstein, P. Birkner, M. Joye, T. Lange, and C. Peters, Twisted edwards curves, in S. Vaudenay, editor, *Progress in Cryptology – AFRICACRYPT 2008*, pp. 389–405, Springer Berlin Heidelberg, Berlin, Heidelberg, 2008, ISBN 978-3-540-68164-9.

[6] D. J. Bernstein, C. Chuengsatiansup, and T. Lange, Curve41417: Karatsuba revisited, in *Cryptographic Hardware and Embedded Systems - CHES 2014 - 16th International Workshop, Busan, South Korea, September 23-26, 2014. Proceedings*, pp. 316–334, 2014, `https://doi.org/10.1007/978-3-662-44709-3_18`.

[7] D. J. Bernstein and T. Lange, Explicit-formulas database, 2019, `https://www.hyperelliptic.org/EFD/`.

[8] D. J. Bernstein and T. Lange, Safecurves: choosing safe curves for elliptic-curve cryptography, 2019, `https://safecurves.cr.yp.to`.

[9] D. J. Bernstein, T. Lange, and P. Schwabe, On the correct use of the negation map in the pollard rho method, Cryptology ePrint Archive, Report 2011/003, 2011, `https://eprint.iacr.org/2011/003`.

[10] J. W. Bos, C. Costello, P. Longa, and M. Naehrig, Selecting elliptic curves for cryptography: an efficiency and security analysis, J. Cryptographic Engineering, 6(4), pp. 259–286, 2016.

[11] Certicom Research, Standards for Efficient Cryptography. SEC 2: Recommended Elliptic Curve Domain Parameters, 2000, `http://www.secg.org/SEC2-Ver-1.0.pdf`.

[12] CFRG, Formal request from TLS WG to CFRG for new elliptic curves, 2014, `https://www.ietf.org/mail-archive/web/cfrg/current/msg04655.html`.

[13] T. Chou, Sandy2x: New curve25519 speed records, in *Selected Areas in Cryptography - SAC 2015 - 22nd International Conference, Sackville, NB, Canada, August 12-14, 2015, Revised Selected Papers*, pp. 145–160, 2015.

[14] C. Costello, P. Longa, and M. Naehrig, A brief discussion on selecting new elliptic curves, June 2015, position paper presented at the NIST Workshop on Elliptic Curve Cryptography Standards (http://www.nist.gov/itl/csd/ct/ecc-workshop.cfm).

[15] H. Fan and M. A. Hasan, A new approach to subquadratic space complexity parallel multipliers for extended binary fields, IEEE Transactions on Computers, 56(2), pp. 224–233, Feb 2007, ISSN 0018-9340.

[16] J.-C. Faugère, L. Perret, C. Petit, and G. Renault, Improving the complexity of index calculus algorithms in elliptic curves over binary fields, in D. Pointcheval and T. Johansson, editors, *Advances in Cryptology – EUROCRYPT 2012*, pp. 27–44, Springer Berlin Heidelberg, Berlin, Heidelberg, 2012, ISBN 978-3-642-29011-4.

[17] A. Faz-Hernández and J. López, *Fast Implementation of Curve25519 Using AVX2*, pp. 329–345, Springer International Publishing, Cham, 2015, ISBN 978-3-319-22174-8.

[18] GNU Project, GCC Options That Control Optimization, 2019, `https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html`.

[19] M. Hamburg, Ed448-goldilocks, a new elliptic curve, Cryptology ePrint Archive, Report 2015/625, 2015, `https://eprint.iacr.org/2015/625`.

[20] IANIX, Things that use Curve25519, 2019, `https://ianix.com/pub/curve25519-deployment.html`.

[21] S. Karati and P. Sarkar, Kummer for genus one over prime order fields, in *Advances in Cryptology - ASIACRYPT 2017 - 23rd International Conference on the Theory and Applications of Cryptology and Information Security, Hong Kong, China, December 3-7, 2017, Proceedings, Part II*, pp. 3–32, 2017.

[22] N. Koblitz, Elliptic curve cryptosystems, Mathematics of Computation, 48(177), pp. 203–203, jan 1987.

[23] P. L. Montgomery, Montgomery, p.l.: Speeding the pollard and elliptic curve methods of factorization. math. comp. 48, 243-264, 48, pp. 243–243, 01 1987.

[24] P. Longa, Fourqneon: Faster elliptic curve scalar multiplications on ARM processors, in *Selected Areas in Cryptography - SAC 2016 - 23rd International Conference, St. John's, NL, Canada, August 10-12, 2016, Revised Selected Papers*, pp. 501–519, 2016.

[25] A. Menezes, The elliptic curve discrete logarithm problem: State of the art, in K. Matsuura and E. Fujisaki, editors, *Advances in Information and Computer Security*, pp. 218–218, Springer Berlin Heidelberg, Berlin, Heidelberg, 2008, ISBN 978-3-540-89598-5.

[26] V. S. Miller, Use of elliptic curves in cryptography, in H. C. Williams, editor, *Advances in Cryptology — CRYPTO '85 Proceedings*, pp. 417–426, Springer Berlin Heidelberg, Berlin, Heidelberg, 1986, ISBN 978-3-540-39799-1.

[27] K. Nath and P. Sarkar, Efficient inversion in (pseudo-)mersenne prime order fields, Cryptology ePrint Archive, Report 2018/985, 2018, `https://eprint.iacr.org/2018/985`.

[28] NIST, Transition Plans for Key Establishment Schemes using Public Key Cryptography, 2017, `https://csrc.nist.gov/News/2017/Transition-Plans-for-Key-Establishment-Schemes`.

[29] N. Perlroth, Government announces steps to restore confidence on encryption standards, Sep 2013, `https://bits.blogs.nytimes.com/2013/09/10/government-announces-steps-to-restore-confidence-on-encryption-standards`.

[30] J. M. Pollard, Theorems on factorization and primality testing, Mathematical Proceedings of the Cambridge Philosophical Society, 76(3), p. 521–528, 1974.

[31] R. L. Rivest, A. Shamir, and L. Adleman, A method for obtaining digital signatures and public-key cryptosystems, Commun. ACM, 21(2), pp. 120–126, February 1978, ISSN 0001-0782.

[32] M. Scott, Ed3363 (highfive) – an alternative elliptic curve, Cryptology ePrint Archive, Report 2015/991, 2015, `https://eprint.iacr.org/2015/991`.

[33] M. Scott, On inversion modulo pseudo-mersenne primes, Cryptology ePrint Archive, Report 2018/1038, 2018, `https://eprint.iacr.org/2018/1038`.

[34] I. Semaev, New algorithm for the discrete logarithm problem on elliptic curves, Cryptology ePrint Archive, Report 2015/310, 2015, `https://eprint.iacr.org/2015/310`.

[35] H. K. Taskin and M. Cenk, Speeding up curve25519 using toeplitz matrix-vector multiplication, in *Proceedings of the Fifth Workshop on Cryptography and Security in Computing Systems*, CS2 '18, pp. 1–6, ACM, New York, NY, USA, 2018, ISBN 978-1-4503-6374-7.

[36] The University of Sydney, Magma Computational Algebra System, 2019, `http://magma.maths.usyd.edu.au/magma/`.

[37] Virtual Applications and Implementations Research Lab, System for Unified Performance Evaluation Related to Cryptographic Operations and Primitives, 2019, `http://bench.cr.yp.to/supercop.html`.

APPENDIX A

SOURCE CODE OF 10-DIMENSIONAL TMVP IMPLEMENTATION

We provide a naive reference ANSI C implementation of the proposed 10-dimensional TMVP Implementation for field multiplication over $\mathbb{F}_{2^{266}-3}$.

```
//  fe_mul_tmvp.c

#include <stdio.h>

#define FE_MUL_TYPE_TMVP_TMVP
//#define FE_MUL_TYPE_TMVP_SCHOOLBOOK

typedef uint32_t fe[10];
static const uint64_t cmask27 = 0x7ffffff;
static const uint64_t cmask23 = 0x7fffff;

uint64_t P1, P2, P3, P4,
         PF1, PF2, PF3, PF4, PF5, PF6, PF7, PF8,
         PT1, PT2, PT3;
uint64_t R[20];

// T (2x2) * V (2x1) = M (2x1)
void Two_tmvp(uint64_t *M0, uint64_t *M1,
              uint64_t T0, uint64_t T1, uint64_t T2,
              uint64_t V0, uint64_t V1)
{
    // 3M + 3A + 2Ad
    PT1 = T0 * (V0 + V1);
    PT2 = (T1 - T0) * V1;
    PT3 = (T2 - T0) * V0;
    *M0 = (PT1 + PT2);
    *M1 = (PT1 + PT3);
}

// T (2x2) * V (2x1) = M (2x1)
```

```
void Two_schoolbook(uint64_t *M0, uint64_t *M1,
                    uint64_t T0, uint64_t T1, uint64_t T2,
                    uint64_t V0, uint64_t V1)
{
    // 4M + 2Ad
    *M0 = (T0 * V0);
    *M0 +=(T1 * V1);
    *M1 = (T2 * V0);
    *M1 +=(T0 * V1);
}

// T is TMVP (4x4) * V (4x1) = M (4x1)
// &MO,&M1,&M2,&M3
// T0, T1, T2, T3, T4, T5, T6
// V0, V1, V2, V3
void Four_by_four(uint64_t *M0, uint64_t *M1,uint64_t *M2,
                  uint64_t *M3, uint64_t T0, uint64_t T1,
                  uint64_t T2, uint64_t T3, uint64_t T4,
                  uint64_t T5,uint64_t T6, uint64_t V0,
                  uint64_t V1,uint64_t V2, uint64_t V3)
{
    // *M0 = (T0*V0) + (T1*V1) + (T2*V2) + (T3*V3);
    // *M1 = (T4*V0) + (T0*V1) + (T1*V2) + (T2*V3);
    // *M2 = (T5*V0) + (T4*V1) + (T0*V2) + (T1*V3);
    // *M3 = (T6*V0) + (T5*V1) + (T4*V2) + (T0*V3);

#ifdef FE_MUL_TYPE_TMVP_SCHOOLBOOK
    Two_schoolbook(&PF1,&PF2,T0,T1,T4,V0,V1);
    Two_schoolbook(&PF3,&PF4,T5,T4,T6,V0,V1);
    Two_schoolbook(&PF5,&PF6,T2,T3,T1,V2,V3);
    Two_schoolbook(&PF7,&PF8,T0,T1,T4,V2,V3);
#elif defined FE_MUL_TYPE_TMVP_TMVP
    Two_tmvp(&PF1,&PF2,T0,T1,T4,V0,V1);
    Two_tmvp(&PF3,&PF4,T5,T4,T6,V0,V1);
    Two_tmvp(&PF5,&PF6,T2,T3,T1,V2,V3);
    Two_tmvp(&PF7,&PF8,T0,T1,T4,V2,V3);
#endif

    *M0 = (uint64_t)(PF1 + PF5);
    *M1 = (uint64_t)(PF2 + PF6);
    *M2 = (uint64_t)(PF3 + PF7);
    *M3 = (uint64_t)(PF4 + PF8);
}

void fe_mul_tmvp(fe h,fe f,fe g)
{
    // A0B0
    Four_by_four(&R[1],&R[2],&R[3],&R[4],
                 (uint64_t)f[1],(uint64_t)f[0],0LL,0LL,
                 (uint64_t)f[2],(uint64_t)f[3],(uint64_t)f[4],
                     (uint64_t)g[0],(uint64_t)g[1],
                     (uint64_t)g[2],(uint64_t)g[3]);

    R[0] = ((uint64_t)f[0] * (uint64_t)g[0]);
    R[4] += ((uint64_t)f[0] * (uint64_t)g[4]);

    // A1B0
    Four_by_four(&R[5],&R[6],&R[7],&R[8],
                     (uint64_t)f[5],(uint64_t)f[4],(uint64_t)f[3],
                     (uint64_t)f[2],(uint64_t)f[6],(uint64_t)f[7],
                     (uint64_t)f[8],(uint64_t)g[0],(uint64_t)g[1],
                     (uint64_t)g[2],(uint64_t)g[3]);

    R[5] += ((uint64_t)f[1] * (uint64_t)g[4]);
    R[6] += ((uint64_t)f[2] * (uint64_t)g[4]);
    R[7] += ((uint64_t)f[3] * (uint64_t)g[4]);
    R[8] += ((uint64_t)f[4] * (uint64_t)g[4]);

    R[9]  = ((uint64_t)f[9] * (uint64_t)g[0]);
    R[9] += ((uint64_t)f[8] * (uint64_t)g[1]);
    R[9] += ((uint64_t)f[7] * (uint64_t)g[2]);
    R[9] += ((uint64_t)f[6] * (uint64_t)g[3]);
    R[9] += ((uint64_t)f[5] * (uint64_t)g[4]);

    // A0B1
    Four_by_four(&P1,&P2,&P3,&P4,
                     (uint64_t)f[1],(uint64_t)f[0],0LL,0LL,
                     (uint64_t)f[2],(uint64_t)f[3],(uint64_t)f[4],
                     (uint64_t)g[5],(uint64_t)g[6],(uint64_t)g[7],
                     (uint64_t)g[8]);

    R[5] += ((uint64_t)f[0] * (uint64_t)g[5]);
    R[6] += (uint64_t)P1;
    R[7] += (uint64_t)P2;
    R[8] += (uint64_t)P3;
    R[9] += (uint64_t)P4;
    R[9] += ((uint64_t)f[0] * (uint64_t)g[9]);

    // A2B0
    Four_by_four(&R[10],&R[11],&R[12],&R[13],
                     (uint64_t)f[9],(uint64_t)f[8],(uint64_t)f[7],
                     (uint64_t)f[6],0LL,0LL,0LL,(uint64_t)g[1],
                     (uint64_t)g[2],(uint64_t)g[3],(uint64_t)g[4]);

    // A1B1
    Four_by_four(&P1,&P2,&P3,&P4,
                     (uint64_t)f[5],(uint64_t)f[4],(uint64_t)f[3],
                     (uint64_t)f[2],(uint64_t)f[6],(uint64_t)f[7],
                     (uint64_t)f[8],(uint64_t)g[5],(uint64_t)g[6],
                     (uint64_t)g[7],(uint64_t)g[8]);

    R[10] += (uint64_t)P1;
    R[10] += ((uint64_t)f[1] * (uint64_t)g[9]);

    R[11] += (uint64_t)P2;
    R[11] += ((uint64_t)f[2] * (uint64_t)g[9]);

    R[12] += (uint64_t)P3;
    R[12] += ((uint64_t)f[3] * (uint64_t)g[9]);

    R[13] += (uint64_t)P4;
    R[13] += ((uint64_t)f[4] * (uint64_t)g[9]);

    R[14] =  ((uint64_t)f[9] * (uint64_t)g[5]);
    R[14] += ((uint64_t)f[8] * (uint64_t)g[6]);
    R[14] += ((uint64_t)f[7] * (uint64_t)g[7]);
    R[14] += ((uint64_t)f[6] * (uint64_t)g[8]);
    R[14] += ((uint64_t)f[5] * (uint64_t)g[9]);

    // A2B1
    Four_by_four(&R[15],&R[16],&R[17],&R[18],
                 f[9],f[8],f[7],f[6],0LL,0LL,0LL,
                 g[6],g[7],g[8],g[9]);

    R[19] = 0LL;

    // Adding matrices and multiplication with 48
    R[0] += ((R[10] << 5) + (R[10] << 4));
    R[1] += ((R[11] << 5) + (R[11] << 4));
    R[2] += ((R[12] << 5) + (R[12] << 4));
    R[3] += ((R[13] << 5) + (R[13] << 4));
    R[4] += ((R[14] << 5) + (R[14] << 4));
    R[5] += ((R[15] << 5) + (R[15] << 4));
    R[6] += ((R[16] << 5) + (R[16] << 4));
    R[7] += ((R[17] << 5) + (R[17] << 4));
    R[8] += ((R[18] << 5) + (R[18] << 4));
    //R[9] += (R[19] << 5) + (R[19] << 4);

    // Reduction
    P1 = R[0] >> 27; R[0] &= cmask27; R[1] += P1;
    P1 = R[1] >> 27; R[1] &= cmask27; R[2] += P1;
    P1 = R[2] >> 27; R[2] &= cmask27; R[3] += P1;
    P1 = R[3] >> 27; R[3] &= cmask27; R[4] += P1;
    P1 = R[4] >> 27; R[4] &= cmask27; R[5] += P1;
    P1 = R[5] >> 27; R[5] &= cmask27; R[6] += P1;
    P1 = R[6] >> 27; R[6] &= cmask27; R[7] += P1;
    P1 = R[7] >> 27; R[7] &= cmask27; R[8] += P1;
    P1 = R[8] >> 27; R[8] &= cmask27; R[9] += P1;
    P1 = R[9] >> 23; R[9] &= cmask23; R[0] += (P1 * 3);
    P1 = R[0] >> 27; R[0] &= cmask27; R[1] += P1;

    h[0] = (uint32_t)R[0];
    h[1] = (uint32_t)R[1];
    h[2] = (uint32_t)R[2];
    h[3] = (uint32_t)R[3];
    h[4] = (uint32_t)R[4];
    h[5] = (uint32_t)R[5];
    h[6] = (uint32_t)R[6];
    h[7] = (uint32_t)R[7];
    h[8] = (uint32_t)R[8];
    h[9] = (uint32_t)R[9];
}
```

Listing 1: Reference Implementation of TMVP-Based Field Multiplication

APPENDIX B

SOURCE CODES AND PROCESS DETAILS FOR CURVE SEARCH

To find a finite field with prime characteristic of the form $2^n \pm c$, we use the following MAGMA code for $n \in [255, 600]$ and $c \in [1, 1000]$.

```
FILE := "~/Desktop/fieldsearch_output.txt";
for j:=255 to 600 do
for i:=1 to 1000 do
  if (IsPrime(2^j+i)) then
    fprintf FILE, "2^%o+%o,+,%o,%o\n",j,i,j,i;
  end if;
  if (IsPrime(2^j-i)) then
    fprintf FILE, "2^%o-%o,-,%o,%o\n",j,i,j,i;
  end if;
end for;
end for;
```

Listing 2: Prime Number Search Code

We have found 2562 different prime numbers as the result of the computation above. We chose the following primes as start point: $2^{266} - 3, 2^{336} - 3, 2^{452} - 3, 2^{545} - 3, 2^{550} - 5$.

To find proper elliptic curves on these finite fields, we look for the following conditions:

Define a montgomery curve $y^2 = x^3 + Ax^2 + x$;

1. Where $(A - 2)/4$ is as small as possible
2. Order of the curve is close to a prime (like 4p or 8p);
3. Order of the twist of the curve is close to a prime (like 4p or 8p);

To find the order of the curve we use MAGMA's `Order()` function which is defined as "Order(E): The order of the group of K-rational points of E, where E is an elliptic curve defined over the finite field K." in Magma Handbook [36].
To find the twist of the curve we use MAGMA's `Twists()` function which is defined as "Twists(E): Given an elliptic curve over a finite field K, returns the sequence of all nonisomorphic elliptic curves over K which are isomorphic over an extension field. The first of these curves is isomorphic to E. " in Magma Handbook. This function returns two elliptic curves where the first one is the curve itself and second one is the twisted form of the curve. After finding a suitable curve, one can easily check other conditions against attacks mentioned in Section 5.1.2.
Before we start exhaustive search, to verify our code works properly, we have tested the code with Elliptic Curve defined by $y^2 = x^3 + 486662x^2 + x$ over $\mathbb{F}_{2^{55}-19}$ which is the Curve25519.
We use the following MAGMA code for this purpose.

```
n := 266;
c := 3;
A := 20710;

K := GF(2^n - c);
Qx<x> := PolynomialRing(K);
I := Integers();

printf "n = %o\n",n;
printf "c = %o\n",c;
printf "A = %o\n",A;
printf "K: %o\n",K;
printf "Qx: %o\n",Qx;
printf "I: %o\n",I;

f := x^3 + A*x^2 + x;
E := EllipticCurve(f);
printf "f(x) = %o\n",f;
printf "E: %o\n",E;

TList := Twists(E);
printf "Number of Twists = %o\n",#TList;
for j := 1 to #TList do
    printf "\n---------- Twist #%o ----------\n%o\n",j,TList[j];
    ord := Order(TList[j]);
    printf ">>> Order = %o\n",ord;
    try
    if IsPrime(I!(ord/2)) then
```

```
            printf ">>> 2p\n";
        elif IsPrime(I!(ord/4)) then
            printf ">>> 4p\n";
        elif IsPrime(I!(ord/8)) then
            printf ">>> 8p\n";
        end if;
        catch e
            z := 1;
        end try;
    end for;
```

Listing 3: Safe Curve Search Code

The search code supplied above finds the Curve2663. The code takes 3 fixed parameters, namely, n, c and A. It is possible to extend this code by creating loops on the parameter A to search massively on a wide range of curves. Using this kind of parallelization, search space can be distributed over multiple cores.

We have searched for curves over different prime fields using 23 cores for the ranges for each finite field mentioned in the Table B.1

Table B.1: Search Ranges for Elliptic Curves

Prime Field	**"A" Coefficient Range**	**# of Curves**	**Avg. Time**	**CPU Hours**
$2^{266}-3$	$6-80002$	20000	32 sec	177 hours
$2^{336}-3$	$6-1140002$	285000	94 sec	7441 hours
$2^{452}-3$	$6-68002$	17000	328 sec	1548 hours
$2^{545}-3$	$6-140002$	35000	710 sec	6902 hours
$2^{550}-5$	$6-60002$	15000	959 sec	3995 hours

We have spent a total of 20063 CPU hours (nearly 836 CPU days) for curve search process. First result that fits into our conditions was the elliptic curve defined by $y^2 = x^3 + 20710x^2 + x$ over $\mathbb{F}_{2^{266}-3}$ where details were introduced in Section 5.2.1. After finding this result, we have stopped working on the field $2^{266}-3$, and focused on other fields.

CURRICULUM VITAE

PERSONAL INFORMATION

Surname, Name: Taşkın, Halil Kemal
Nationality: Turkish
Year and Place of Birth: 1987, Ödemiş

EDUCATION

Degree	**Institution**	**Year of Graduation**
M.Sc., Cryptography	Middle East Technical University	2011
B.Sc., Mathematics Ed. (Elt.)	Gazi University	2009
High School	Ödemiş Anatolian Teacher Training High School	2005

RESEARCH INTERESTS

Cryptography, Efficient implementation of Cryptographic Primitives, Cryptographic Protocols, End-to-end Encryption, Application Security, Reverse Engineering, Security Testing.

PUBLICATIONS

International Conference Publications

- Halil Kemal Taşkın and Murat Cenk. Speeding up Curve25519 using Toeplitz Matrix-vector Multiplication. In Proceedings of the Fifth Workshop on Cryptography and Security in Computing Systems (CS2 '18). ACM, New York, NY, USA, 1-6. 2018. DOI: https://doi.org/10.1145/3178291.3178292 .

- Halil Kemal Taşkın, Murat Demircioğlu and Salim Sarımurat, *End-to-end Encrypted Communication Between Multi-device Users*, Information Security and Cryptology Conference, İstanbul, Turkey, October 2014.

- Murat Demircioğlu, Halil Kemal Taşkın and Salim Sarımurat, *Security Analysis of the Encrypted Mobile Communication Applications*, Information Security and Cryptology Conference, İstanbul, Turkey, October 2014.

- Cihangir Tezcan, Halil Kemal Taşkın and Murat Demircioğlu, *Improbable Differential Attacks on Serpent using Undisturbed Bits*, In Proceedings of the 7th International Conference on Security of Information and Networks (SIN '14). Glasgow, UK. 2014. DOI: https://doi.org/10.1145/2659651.2659660 .

- Halil Kemal Taşkın and Murat Demircioğlu, *Off-the-Record Communication with Location Hiding*, Information Security and Cryptology Conference, Ankara, Turkey, September 2013.

Proceedings/Posters

- Halil Kemal Taşkın and Murat Demircioğlu, *Applications of Cryptology in Cyber security and End-to-end Encryption*, TBD 31st National IT Congress, Ankara, Turkey, November 2014.

- Halil Kemal Taşkın, *Using Context Triggered Piecewise Hashing on Computer Forensics", International Symposium on Digital Forensics*, Ankara, Turkey, May 2014. (Poster)

Invited Talks/Presentations

- Halil Kemal Taşkın, *On Blockchain*, CyberEge 2018 Ege University, İzmir, May 2018.

- Halil Kemal Taşkın, *Cloud Computing and Security*, 6th Cyber Security Platform, TOBB ETU, Ankara, November 2017.

- Halil Kemal Taşkın, *Cryptology and End-to-end Encryption*, Osmangazi University Informatics Days 2016, Eskişehir, May 2016.
- Halil Kemal Taşkın, *A Mini Course on Magma CAS Programming*, METU SIAM, Ankara, March 2016. (A 4-hour crash course for graduate students)
- Halil Kemal Taşkın, *Cyberspace: The Fifth Domain of Warfare*, METU SIAM Seminar Series, Ankara, April 2016.
- Halil Kemal Taşkın, *Applications of Cryptology in Cyber security and End-to-end Encryption*, TSE 4th IT Standards Conference, Ankara, October 2015.
- Halil Kemal Taşkın, *Anonymity Online: Computer networks and Security, Tor & I2P*, Ankara Cryptology Seminars, METU, Ankara, March 26th 2013.

Projects

- TÜBİTAK 1001 #115R289, *Developing and Implementation of Efficient Algorithms for Public Key Cryptography*, Student Researcher, 2016-2018.
- TÜBİTAK 1001 #112E101, *Improbable Differential Cryptanalysis of Block Ciphers*, Student Researcher, 2012-2013.