FASTER RESIDUE MULTIPLICATION MODULO 521-BIT MERSENNE PRIME AND APPLICATION TO ECC

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

BY

SHOUKAT ALI

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

SEPTEMBER 2017

Approval of the thesis:

FASTER RESIDUE MULTIPLICATION MODULO 521-BIT MERSENNE PRIME AND APPLICATION TO ECC

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

Prof. Dr. Bülent Karasözen _____
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. Ferruh Özbudak
Mathematics, METU _____

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

Prof. Dr. Ersan Akyıldız
Mathematics, METU _____

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

Assist. Prof. Dr. Oğuz Yayla
Mathematics, Hacettepe University _____

Date: _____

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Last Name: SHOUKAT ALI

Signature :

ABSTRACT

FASTER RESIDUE MULTIPLICATION MODULO 521-BIT MERSENNE PRIME AND APPLICATION TO ECC

Ali, Shoukat
Ph.D., Department of Cryptography
Supervisor : Assoc. Prof. Dr. Murat Cenk

September 2017, 94 pages

We present faster algorithms for the residue multiplication modulo 521-bit Mersenne prime on 32- and 64-bit platforms by using Toeplitz Matrix-Vector Product (TMVP). The total arithmetic cost of our proposed algorithms is less than the existing algorithms and we select the ones, 32- and 64-bit residue multiplication, with the best timing results on our testing machine(s). For the 64-bit residue multiplication we have presented three versions of our algorithm along with their arithmetic cost and from implementation point of view, we provide the timing results of each version. The transition from 64- to 32-bit residue multiplication is full of challenges because the number of limbs becomes double and the bitlength of the limbs reduces by half. We propose three technique for 32-bit residue multiplication such that both the arithmetic cost and the timing results of each one is provided. Without use of any intrinsics and SIMD/assembly instructions in our implementation, on Intel(R) Core $\mathrm{i5}-6402\mathrm{P}$ CPU @ 2.80GHz, we find 136- and 550-cycle for our 64- and 32-bit residue multiplications, respectively. We implement constant-time variable- and fixed-base scalar multiplication on the standard NIST curve P-521 and Edwards curve E-521. Using our residue multiplication(s), we find E-521 more efficient than P-521 especially for variable-base scalar multiplication.

Keywords: residue multiplication, Toeplitz matrix-vector product, Mersenne prime, elliptic curve cryptography, variable- and fixed-base scalar multiplication, 32- and 64-bit platforms

ÖZ

521 BİTLİK MERSENNE ASAL MODÜLLERİNDE HİZLİ ÇARPMA VE ECC YE UYGULAMALARI

Ali, Shoukat
Doktora, Kriptografi Bölümü
Tez Yöneticisi : Doç. Dr. Murat Cenk

Eylül 2017 , 94 sayfa

Toeplitz Matris - Vektör Çarpma (TMVP) kullanan, 32-ve 64-bit platformlarda çalışan ve 521-bit Mersenne asal modlarındaki modüler çarpmalar için daha hızlı algoritmalar sunmaktayız. Önerilen algoritmalarımızın toplam aritmetik maliyeti mevcut algoritmalar ile kıyaslandığında daha azdır, ayrıca test makinalarmızda en iyi zamanlama sonuçlarına sahip olan algoritmalar, 32- ve 64-bit modüler çarpmaları seçilmiştir. 64-bit modüler çarpma için aritmetik maliyetleri ile birlikte algoritmamızın üç versiyonu gösterilmiştir ve uygulama bakış açısından, her bir versiyonun zamanlama sonuçları da verilmiştir. Limb sayısı iki katına çıkarken limb bit uzunluğu yarıya azaldığı için 64-ten 32-bit kalıntı çarpımına geçiş zorluklarla doludur. Biz 32-bit modüler çarpmaları için hem aritmetik maliyetleri hem de zamanlama sonuçları ile birlikte üç teknik önermekteyiz. Uygulamamızda herhangi bir yapısal ve SIMD / montaj talimatı kullanmadan Intel® Core i5 - 6402P CPU @ 2.80GHz'de sırasıyla 64- ve 32-bit modüler çarpmalar için 136- ve 550- devir bulunmuştur. Standart NIST eğrisi P-521 ve Edwards eğrisi E-521 için sabit-zamanlı değişken ve sabit-bazlı skaler çarpmalar uygulanmıştır. Bizim modüler çarpmalarımız kullanıldığı zaman, özellikle değişen-bazlı skaler çarpım için E-521, P-521'den daha verimli bulunmuştur.

Anahtar Kelimeler: Modüler çarpma, Toeplitz matris-vektör ürünü, Mersenne prime,

Eliptik eğrisi kriptografi, değişken ve sabit-bazlı skaler çarpımı, 32- ve 64-bit platformları

To My Family

ACKNOWLEDGMENTS

I am thankful to my dissertation supervisor Assoc. Prof. Dr. Murat Cenk for his insightful ideas, constant encouragements and valuable guidance in my research work. I am also deeply grateful to him for all the support, time and openly sharing his experiences.

I am thankful and grateful to Michael Scott for both making his constant-time variable-base scalar multiplication code public and answering our questions related to implementations. Similarly, I would to thank Okash Khawaja, Dr. Çağdaş Çalık and others for their valuable discussions and supports in improving our implementation results.

Embarking on the voyage of PhD would have never been possible without the firm trust, unconditional supports and lasting encouragements of my parents and siblings. They have always been there at the ups and downs of my life.

The time I spent at our Cryptology lab. and Institute of Applied Mathematics (IAM) is and will always be the most memorable moments of my life. In spite of the rough and tough waves of time, I came across many good people at IAM. Those people made my research work and life pleasant and joyous. I shall be thankful to all of them.

Lastly, I would to thank Türkiye Bursları for awarding me full scholarship to study at Middle East Technical University (METU), Ankara, Turkey. This research was also partially supported by the Scientific and Technological Research Council of Turkey (TÜBİTAK) under Grant No. 115R289.

TABLE OF CONTENTS

LIST OF TABLES

TABLES

LIST OF FIGURES

FIGURES

LIST OF ABBREVIATIONS

TMVP	Toeplitz Matrix-Vector Product
ECC	Elliptic Curve Cryptography
ECDLP	Elliptic Curve Discrete logarithm Problem
SCR	Short Coefficient Reduction
EFD	Explicit Formulas Database
GCC	GNU Compiler Collection
NAF	Non-adjacent Form
P-521	Short Weierstrass standard NIST curve
E-521	Standard Edwards curve
$\mathbb{F}_p$	(Large-characteristic) Prime field
M	Single-precision/word Multiplication
S	Single-precision/word Squaring
A	Single-precision/word Addition/Subtraction
$\mathbf{A}_d$	Double-precision/word Addition/Subtraction

CHAPTER 1

INTRODUCTION

Efficient cryptographic primitives have always been among the main objectives of the researchers in cryptography. Therefore, the addition of Elliptic Curve Cryptography (ECC) in the family of public key cryptography is a proof of efficiency over contemporary counterparts e.g. RSA. In ECC, an elliptic curve is defined over a finite field where the characteristic of the field can be small or large prime. So, efficient finite field arithmetic implies efficient ECC. That's why the expensive finite field multiplication and inversion (multiplicative inverse) have received good attention.

The expensive inversion can be avoided by choosing a higher coordinate system e.g. Jacobian coordinate. But if inversions are required for some other efficiency purposes i.e. mixed point addition, then Montgomery's trick — computing the inverse of a batch of elements by actually computing one inversion only — is the most efficient strategy. Fermat's Little Theorem is used to compute the inversion in constant-time so that, the operation computation is guarded against simple timing attacks.

The finite field multiplication is comprised of integer multiplication and the expensive reduction operation. In the literature there are different techniques to speed up the reduction operation and in case of Mersenne prime the operation is optimal in efficiency because the reduction is computed by field addition(s) only. Unfortunately, there are few Mersenne primes for contemporary cryptographic purposes. But there are other kind of special primes such as Crandall primes, special Montgomery primes, Solinas primes, Goldilocks prime etc. The implementation benefits of the Solinas primes [35] for its efficiency, especially on 32-bit platform, made its way to become part of the standards [15, 27]. The newly designed Goldilocks prime by Mike Hamburg [25]

is very useful for Karatsuba algorithm because at each level of Karatsuba, it saves (double-precision) addition(s)/subtraction(s). Above all, the prime is equally suitable for both 32-bit and 64-bit platforms.

For the ECC bitlength, the popular algorithm of Karatsuba [28] is widely used to compute integer multiplication efficiently. Bernstein [4] presented the “refined Karatsuba identity" which takes less number of additions than the original Karatsuba identity. Furthermore, to reduce the cost of reduction in multiply-then-reduce technique, Bernstein et al. [5] eliminated some additions by using “reduced refined Karatsuba" in their implementation. As explained in [5] it is based on the idea of reducing inputs to a multiplication rather outputs of a multiplication. On the other hand, Granger and Scott [23] achieved the (arithmetic) efficiency for their modular multiplication algorithm by observing the structure/pattern of modular multiplication when multiplication and reduction steps are combined as single expression. Similarly, we — Ali and Cenk [1] — observed that the same expression can be represented as a Toeplitz Matrix-Vector Product (TMVP) and further reduced the total arithmetic cost of the modular multiplication for 521-bit Mersenne prime modulus.

The Karatsuba algorithm [28] trades multiplication for addition(s)/subtraction(s) in order to reduce the complexity of multiplication operation and asymptotically this is good. But for the bitlength relevant to contemporary ECC, such techniques are useful only when the cost of saved multiplications is higher than the new addition(s) or subtraction(s) that come as overhead. The ratio of multiplication to additions/subtractions in Karatsuba 3-way, Toom-3 and Granger and Scott technique can be useful in asymptotic analysis but as shown in [1] that a more balanced way of trading multiplication for addition(s)/subtraction(s) is more useful in terms of the arithmetic cost and suitable for contemporary ECC bitlength. Moreover, for multi-precision integers, the use of reduced-radix representation — bitlength of limb is less than the word-size of underlying computer — provides space for saving the result of addition(s)/subtraction(s) that were produced as a result of the saved multiplication(s). Now, the carries produced as a result of those addition(s)/subtraction(s) do not need to be propagated immediately. Hence, the reduced-radix representation provides the advantage of postponding the carry propagation and has shown encouraging results such as [1, 3, 5, 6, 7, 11, 16, 23, 25, 34]. On the other hand, the conventional method

of using the packed-radix representation — bitlength of limb is equal to the word-size of underlying computer — has also been implemented by different reserachers and some of the examples are [13, 14, 17, 22, 24, 30].

Darrel Hankerson et al. have pointed out in their book [19] that the performance of ECC depends heavily on the speed of field multiplication. Therefore, our work is focused on designing new faster residue multiplication modulo 521-bit Mersenne prime p for 32- and 64-bit platforms by taking the finite field multiplication as TMVP. To the best of our knowledge, this is the first use of TMVP to perform residue multiplication in non-binary field and TMVP has been used in binary field for example, [21]. In the first part of our work [1], we presented our residue multiplication for 64-bit platforms such that the total arithmetic cost is less than the previously known best algorithm of Granger and Scott [23] for 521-bit Mersenne prime modulus. The total arithmetic cost was computed by taking the cost ratio of multiplication to addition as 3. Furthermore, we presented three versions of our residue multiplication to provide an extensive comparison — to the best of our knowledge — with respect to the other well-known algorithms using the modulus p. Each version of our residue multiplication has lesser total number of operations than its counterparts. For implementation purpose, Granger and Scott [23] chose 522-bit modulus i.e. $2p$ and in our case, we implemented both modulus p and $2p$. For each version of our residue multiplication, we find the timing results of modulus p better than $2p$. To show the efficiency of their algorithm [23], Granger and Scott implemented constant-time variable-base scalar multiplication on the standard NIST curve P-521 and Edwards curve E-521. They made their C++ code public. So, following their work, we did the same by using modulus p and $2p$. We find the timing results of constant-time variable-base scalar multiplication for P-521 and E-521 using the three versions of our residue multiplications better than the public code of [23] on our machine. Note that to ensure fair comparison, we used the same public code of [23] and the changes required for modulus p were made accordingly. All the codes were executed in the same testing environment.

After the successful completion of the first part of our work, we decided to extend our work to 32-bit platforms using the TMVP approach. So, with respect to 64-bit platforms, on 32-bit platforms the number of limbs becomes double and the bitlength

of the limbs reduces by half. But things are not so simple and easy. Indeed, there are many challenges among which correctness, called numerical stability in [33], comes first due to the overflow on 32-bit platforms. The number of limbs on 32-bit platforms is a multiple of both 2 and 3 which opens the way for two- and three-way decomposition. The structure and decomposition of the TMVP lead us to three techniques of performing the residue multiplication such that the total arithmetic cost is less than the well-known algorithms using the modulus p. Again, we take the cost of one single-precision multiplication equal to three single-precision additions. Furthermore, for the sake of simplicity, the arithmetic costs are presented in terms of single-precision multiplication and additions by taking the cost of one double-precision addition equal to two single-precision additions. Our three 32-bit residue multiplications are discussed in detail along with their arithmetic costs and the timing results. So, among the three techniques we have selected the one with the best timing result on our machine. For the chosen technique, we have provided a proof — using the worst-case scenario — that our 32-bit residue multiplication does not cause overflow. Similarly, efficiency is compromised to avoid overflow and we have ensured the minimal loss of efficiency both in terms of arithmetic cost and clock cycles. For the clock cycles, we performed timing tests on our machine to find out the cross-over point between the schoolbook and TMVP. Using the chosen technique, we have implemented constant-time fixed- and variable-base scalar multiplication for 32-bit platforms. Unlike the first part of our work, this time we have implemented both constant-time fixed- and variable-base scalar multiplication for 64-bit platforms.

The rest of the thesis is organized as follows: In Chapter 2, we briefly introduce our readers to the rudimentary concepts that we belief will set them to a good start to easily comprehend the following chapters and maybe beyond. Next, we discuss our 64-bit residue multiplication in detail and its three versions. The arithmetic cost and timing result of the three version of our residue multiplication are provided in Chapter 3. In Chapter 4, we discuss in detail our three techniques to perform 32-bit residue multiplication. The challenges that arise in the design and implementation of our 32-bit residue multiplication are also discussed. We end the chapter by providing the worst-case bitlength analysis of our most efficient technique in terms of clock cycle. In Chapter 5, we discuss the vital operation of single-scalar multiplication in ECC

and using our 32- and 64-bit residue multiplication, we implemented the constant-time fixed- and variable-base scalar multiplication for the standard NIST curve P-521 and Edwards curve E-521. We have used more than one computer for the timing tests and all the results are provided. Finally, in Chapter 6, we conclude our research work.

CHAPTER 2

PRELIMINARIES

In this chapter we cover the rudimentary topics that are required for the following chapters. First, we briefly introduce the Finite field. Then we discuss the large-characteristic prime field arithmetic such that the pseudocode of each field operation is provided. Next, we briefly introduce the Toeplitz matrix and discuss our decomposition methods that are used as the basis to compute the Toeplitz matrix-vector product (TMVP). There is also a brief section on different forms/shapes of primes that are used in elliptic curve cryptography (ECC). In the next section we briefly introduce the elliptic curve over prime field. Then we briefly discuss the use of Diffie-Hellman key exchange protocol in ECC. Finally, we discuss some of the different coordinate systems that are used to represent a point on elliptic curve.

2.1 Finite Field

A finite set $\mathbb{F}$ with the operations of addition $(+)$ and multiplication $(\cdot)$ is called finite field such that it satisfies the following properties:

- $\mathbb{F}$ with addition is an Abelian group
- $\mathbb{F}$ with multiplication, excluding 0, is an Abelian group
- $(x + y) \cdot z = x \cdot z + y \cdot z$ for all $x, y, z \in \mathbb{F}$.

By definition, addition and multiplication are the default operations on the elements of $\mathbb{F}$ and the properties of Abelian group imply that now, we have also the operations

of “subtraction" and “division". Subtraction is defined as: for all $x, y \in \mathbb{F}, x-y = x+(-y)$ where $-y$ is the unique additive inverse of y in $\mathbb{F}$ such that $y-y = y+(-y) = 0$ and 0 is called the additive identity. Similarly, division is defined as: for all $x, y \in \mathbb{F}$, excluding the element 0, $x/y = x \cdot y^{-1}$ where y^{-1} is the unique multiplicative inverse of y in $\mathbb{F}$ such that $y/y = y \cdot y^{-1} = 1$ and 1 is called the multiplicative identity. Note that the division comprises of the two operations: inversion, and multiplication.

The number of elements in a finite field is called “order" of the field. For a prime number p and a positive integer m, it is known that there exists a finite field $\mathbb{F}$ of order p^m where p is called the “characteristic". $\mathbb{F}$ is called a “prime field" when $m = 1$ and an “extension field" when $m \geq 2$. Furthermore, any two finite fields of order p^m are “isomorphic"; which means that the two fields are structurally the same except their labeling/representation.

From above it is clear that for a prime number p, the set $\{0, 1, \ldots, p-1\}$ with addition and multiplication performed modulo p is a prime field with characteristic and order both p. For any integer x, the “reduction modulo p" is an integer in $\{0, 1, \ldots, p-1\}$ which is actually the remainder of integer division of x by p. The characteristic-two finite field is called “binary field". In practice, the small-characteristic finite fields — characteristic $2, 3$ etc. — are defined over extension fields. In our work, we are interested in large-characteristic prime field and denoted by $\mathbb{F}_p$.

2.2 Prime Field Arithmetic

In this section, we present fundamental algorithms for (large-characteristic) prime field arithmetic that are independent of the shape of the prime p. Actually, the shape of p has considerable impact on the efficiency of the slow reduction operation and that’s why we have different shapes of prime at our disposal in ECC.

Suppose we are working on a W-bit computer and in our case $W \in \{64, 32\}$. So, a W-bit word $B = (b_{W-1}, \ldots, b_1, b_0)$ is a block where $b_i \in \{0, 1\}$ for $i = 0, 1, \ldots, W-1$. The bit b_0 is called the “least significant bit" and b_{W-1} is called the “most significant bit". So, the bitlength of p is $k = \lceil \log_2 p \rceil$ and its word-length is $l = \lceil k/W \rceil$. Furthermore, we assume that the elements of $\mathbb{F}_p$ are k-bit integers and in

our case k is very large than W which implies that an element $x \in \{0, 1, \ldots, p-1\}$ is split in l-word called “chunks/limbs". In practice one can store x in an array $X = [x_0, x_1 \ldots, x_{l-1}]$ of l W-bit words where x_0 is the “least significant limb" and x_{l-1} is the “most significant limb".

The use of the full computer word for each limb is called packed-radix representation. Alternatively, one can use reduced-radix representation such that the bitlength of the limbs is less than W. The reduced-radix representation has shown interestingly results in practice but in this section, all the algorithms that we present are using packed-radix representation. Note that the algorithms presented in this section are taken from Darrel Hankerson et al. book [19] and amendments in the pseudo-code are made according to our naming convention and notations.

2.2.1 Addition and Subtraction

The modular addition and subtraction is performed by adding and subtracting the corresponding limbs along with the carry, respectively. The Algorithm 1 and Algorithm 2 perform addition and subtraction in $\mathbb{F}_p$, respectively. From Algorithm 1 and Algorithm 2, it is evident that in the case of packed-radix the carry propagation must be performed simultaneously along with the limbs addition/subtraction.

2.2.2 Multiplication

One can compute the field multiplication in $\mathbb{F}_p$ using the multiply-then-reduce approach. In other words, first we compute the integer multiplication of two field elements and then reduce the result modulo p. In this section, we discuss the integer multiplication only. Before presenting the pseudo-code of the schoolbook method for integer multiplication, we assume that $(u_1 u_0)$ denote a $(2W)$-bit integer where u_0 and u_1 are the lower and higher W-bit words, respectively. The Algorithm 3 represents the schoolbook method of operand scanning whose complexity is $\mathcal{O}(n^2)$. The complexity of multiplication was reduced from quadratic to $\mathcal{O}(n^{\log_2 3})$ by Karatsuba and Ofman in [28]. The algorithm is based on divide-and-conquer technique where both the input operands are taken as two equal half-size operands with higher and lower

Algorithm 1 Addition in $\mathbb{F}_p$

Input: $X = [x_0, \ldots, x_{l-1}], Y = [y_0, \ldots, y_{l-1}]$ and modulus p where $X, Y \in [0, p-1]$

Output: $Z = [z_0, \ldots, z_{l-1}]$ where $Z \equiv X + Y \pmod{p}$

$z_0 \leftarrow x_0 + y_0 \pmod{2^W}$
if $x_0 + y_0 \in [0, 2^W)$ **then** $carry \leftarrow 0$
else $carry \leftarrow 1$
for i from 1 to $l - 1$ **do**
    $z_i \leftarrow (x_i + y_i + carry) \pmod{2^W}$
    if $(x_i + y_i + carry) \in [0, 2^W)$ **then** $carry \leftarrow 0$
    else $carry \leftarrow 1$
if $carry = 1$ **then** $Z \leftarrow Z - p$
else if $Z \geq p$ **then** $Z \leftarrow Z - p$
Return Z

Algorithm 2 Subtraction in $\mathbb{F}_p$

Input: $X = [x_0, \ldots, x_{l-1}], Y = [y_0, \ldots, y_{l-1}]$ and modulus p where $X, Y \in [0, p-1]$

Output: $Z = [z_0, \ldots, z_{l-1}]$ where $Z \equiv X - Y \pmod{p}$

$z_0 \leftarrow x_0 - y_0 \pmod{2^W}$
if $x_0 - y_0 \in [0, 2^W)$ **then** $carry \leftarrow 0$
else $carry \leftarrow 1$
for i from 1 to $l - 1$ **do**
    $z_i \leftarrow (x_i - y_i - carry) \pmod{2^W}$
    if $(x_i - y_i - carry) \in [0, 2^W)$ **then** $carry \leftarrow 0$
    else $carry \leftarrow 1$
if $carry = 1$ **then** $Z \leftarrow Z + p$
Return Z

Algorithm 3 Integer Multiplication

Input: $X = [x_0, \ldots, x_{l-1}], Y = [y_0, \ldots, y_{l-1}]$ where $X, Y \in [0, p-1]$

Output: $Z = [z_0, \ldots, z_{2l-1}]$ where $Z = X \cdot Y$

1: **for** i from 0 to $l-1$ **do**
2: $\quad z_i \leftarrow 0$
3: **for** i from 0 to $l-1$ **do**
4: $\quad u_1 \leftarrow 0$
5: $\quad$ **for** j from 0 to $l-1$ **do**
6: $\quad\quad (u_1 u_0) \leftarrow z_{i+j} + x_i \cdot y_j + u_1$
7: $\quad\quad z_{i+j} \leftarrow u_0$
8: $\quad z_{i+l} \leftarrow u_1$
9: Return Z

parts. Algebraically, suppose $x = x_1 2^k + x_0$ and $y = y_1 2^k + y_0$ are $2k$-bit integers then the Karatsuba algorithm computes $X \cdot Y$ as follows:

$$\begin{aligned} x \cdot y &= (x_1 2^k + x_0) \cdot (y_1 2^k + y_0) \\ &= x_1 \cdot y_1 2^{2k} + \big((x_1 + x_0) \cdot (y_1 + y_0) - x_1 y_1 - x_0 y_0\big) 2^k + x_0 \cdot y_0. \end{aligned}$$

Unlike the Algorithm 3 where 4 multiplications are required to compute $x \cdot y$, Karatsuba algorithm performs the same operation with 3 multiplications and some extra single- and double-word additions/subtractions. So, at each level of Karatsuba algorithm one save $1/4$ multiplications at the cost of some extra single- and double-word additions/subtractions. Normally, the Karatsuba algorithm is applied recursively until some terminating condition where the schoolbook method is more efficient. For the sake of simplicity, we assume that the terminating condition is single-word multiplication and the inputs X, Y are l-word where $l = 2^k$ for some positive integer k. The pseudocode of Karatsuba algorithm is provided as Algorithm 4.

Instead of splitting the inputs in half, there is a variant of Karatsuba algorithm that splits the inputs in 3 equal parts and we call this technique Karatsuba 3-way multiplication. We use the formula of Weimerskirch and Paar [36] to show the working of Karatsuba 3-way. For the sake of simplicity, let $a = a_2 10^2 + a_1 10 + a_0$ and

Algorithm 4 Karatsuba Algorithm

Input: $X = [x_0, \ldots, x_{l-1}], Y = [y_0, \ldots, y_{l-1}]$ where $X, Y \in [0, p-1]$

Output: $X \cdot Y$

1: **if** $l = 1$ **then** Return $X \cdot Y$
2: **else** split X, Y in half
3: $X = X_1 2^{W(l/2)} + X_0$
4: $Y = Y_1 2^{W(l/2)} + Y_0$
5: $A =$Algorithm 4(X_1, Y_1)
6: $B =$Algorithm 4(X_0, Y_0)
7: $C =$Algorithm 4$(X_1 + X_0, Y_1 + Y_0)$
8: $D = C - A - B$
9: $Z = A2^{Wl} + D2^{W(l/2)} + B$
10: Return Z

$b = b_2 10^2 + b_1 10 + b_0$ be two 3-digit integers then $a \cdot b$ is computed as follows:

$$
\begin{aligned}
a \cdot b &= (a_2 10^2 + a_1 10 + a_0) \cdot (b_2 10^2 + b_1 10 + b_0) \\
&= D_2 10^4 + (D_{1,2} - D_1 - D_2) 10^3 + (D_{0,2} - D_2 - D_0 + D_1) 10^2 \\
&+ (D_{0,1} - D_1 - D_0) 10 + D_0 \\
\text{where} \quad & D_0 = a_0 \cdot b_0, \quad D_1 = a_1 \cdot b_1, \quad D_2 = a_2 \cdot b_2, \\
& D_{0,1} = (a_0 + a_1) \cdot (b_0 + b_1), \quad D_{0,2} = (a_0 + a_2) \cdot (b_0 + b_2), \\
& D_{1,2} = (a_1 + a_2) \cdot (b_1 + b_2)
\end{aligned}
$$

So, instead of 9-mulitplication, the Karatsuba 3-way preforms 6-multiplication to compute $a \cdot b$. Hence, the complexity of multiplication using Karatsuba 3-way is $\mathcal{O}(n^{\log_3 6})$. Toom-Cook 3-way multiplication or simply Toom-3 also splits the inputs in 3 equal parts and for this algorithm, we use the optimized formula of Bodrato [12].

Using the same a, b as two 3-digit integers, $a \cdot b$ is computed as follows:

$$
\begin{aligned}
a \cdot b &= (a_2 10^2 + a_1 10 + a_0) \cdot (b_2 10^2 + b_1 10 + b_0) \\
&= D_4 10^4 + D_3 10^3 + D_2 10^2 + D_1 10 + D_0 \\
\text{where} \quad & d_0 = a_2 + a_0, \quad d_3 = b_2 + b_0, \quad d_2 = d_0 - a_1, \quad d_1 = d_3 - b_1, \\
& d_0 = d_0 + a_1, \quad d_3 = d_3 + b_1, \quad D_1 = d_2 \cdot d_1, \quad D_2 = d_0 \cdot d_3, \\
& d_0 = ((d_0 + a_2) << 1) - a_0, \quad d_3 = ((d_3 + b_2) << 1) - b_0, \\
& D_3 = d_0 \cdot d_3, \quad D_0 = a_0 \cdot b_0, \quad D_4 = a_2 \cdot b_2, \\
& D_3 = (D_3 - D_1)/3, \quad D_1 = (D_2 - D_1) >> 1, \quad D_2 = D_2 - D_0, \\
& D_3 = ((D_3 - D_2) >> 1) - D_4 << 1, \quad D_2 = D_2 - D_1 - D_4, \\
& D_1 = D_1 - D_3,
\end{aligned}
$$

The operators "<<" and ">>" represent the shift-left and -right bit operation, respectively. So, the Bodrato's formula takes only 5 multiplications to compute $a \cdot b$ which is less than the Karatsuba 3-way. But there are more single- and double-word addition(s)/subtraction along with some shifts and division by 3 operations in Bodrato's formula. Hence, the complexity of multiplication using the Toom-3 is $\mathcal{O}(n^{\log_3 5})$.

2.2.3 Squaring

The multiply-then-reduce approach can also be to field squaring in $\mathbb{F}_p$ such that the integer element is first squared and then the result is reduced modulo p. Just like the last section, we discuss only the integer squaring in this section. Clearly, squaring is more efficient than multiplication because the number of single-word multiplications reduces roughly to half in squaring. The integer squaring is presented as Algorithm 5.

2.2.4 Reduction

In field multiplication, the reduction is expensive operation because of the (expensive) division. Therefore, to reduce the computational cost of reduction, the researchers have devised special primes e.g. Solinas primes, Crandall primes, Goldilocks prime

Algorithm 5 Integer Squaring

Input: $X = [x_0, \ldots, x_{l-1}]$ where $X \in [0, p-1]$

Output: $Z = [z_0, \ldots\ldots, z_{2l-1}]$ where $Z = X^2$

1: $T_0 \leftarrow 0, T_1 \leftarrow 0, T_2 \leftarrow 0,$

2: **for** k from 0 to $2l-2$ **do**

3: **for** each element of $\{(i,j)|i+j=k, 0 \leq i \leq j \leq l-1\}$ **do**

4: $(u_1u_0) \leftarrow x_i \cdot x_j$

5: **if** $(i < j)$ **then**

6: $(u_1u_0) \leftarrow 2(u_1u_0) \pmod{2^{2W}}$

7: **if** $2(u_1u_0) \in [0, 2^{2W})$ **then** $carry \leftarrow 0$

8: **else** $carry \leftarrow 1$

9: $T_2 \leftarrow T_2 + carry$

10: $T_0 \leftarrow T_0 + u_0 \pmod{2^W}$

11: **if** $T_0 + u_0 \in [0, 2^W)$ **then** $carry \leftarrow 0$

12: **else** $carry \leftarrow 1$

13: $T_1 \leftarrow T_1 + u_1 + carry \pmod{2^W}$

14: **if** $(T_1 + u_1 + carry) \in [0, 2^W)$ **then** $carry \leftarrow 0$

15: **else** $carry \leftarrow 1$

16: $T_2 \leftarrow T_2 + carry$

17: $z_k \leftarrow T_0, T_0 \leftarrow T_1, T_1 \leftarrow T_2, T_2 \leftarrow 0$

18: $z_{2l-1} \leftarrow T_0$

19: Return Z

etc. The shape of these primes significantly reduce the computational cost of the reduction which directly speedup the computation of the field multiplications. That's why one can find both in the literature and the standard bodies, the use of the special primes. In this section, we discuss the reduction method of Barrett that is used for arbitrary modulus p.

Barrett reduction is based on the idea of replacing the expensive divisions in reduction by less-expensive operations but it does not take advantage of the shape of the prime i.e. modulus. For positive integer Z and p, Barrett reduction computes Z mod p as shown in Algorithm 6. The Barrett reduction is useful when many reductions are performed with a single modulus p because one has to compute the expensive modulus-dependent quantity $\lfloor b^{2k}/p \rfloor$.

Algorithm 6 Barrett Reduction

Input: $p, b \geq 3, k = \lfloor log_b p \rfloor + 1, Z \in [0, b^{2k})$ and $n = \lfloor b^{2k}/p \rfloor$

Output: Z mod p

1: $q' \leftarrow \left\lfloor \lfloor z/b^{k-1} \rfloor \cdot n/b^{k+1} \right\rfloor$
2: $r \leftarrow (z \bmod b^{k+1}) - (q' \cdot p \bmod b^{k+1})$
3: **if** $r < 0$ **then**
4: $\quad r \leftarrow r + b^{k+1}$
5: **while** $r \geq p$ **do**
6: $\quad r \leftarrow r - p$
7: Return r

2.2.5 Inversion

The multiplicative inverse of a nonzero element x in the prime field $\mathbb{F}_p$ is defined as a unique (nonzero) element x^{-1} in $\mathbb{F}_p$ such that $xx^{-1} = 1$ in $\mathbb{F}_p$. There are different ways of computing x^{-1} and some of the techniques are discussed in this section. Note that from here onwards by inversion we mean multiplicative inverse.

The extended Euclidean algorithm is the basic method of computing inversion. Actually, for any two positive integers a, b the algorithm computes $au + bv = gcd(a, b)$ where u, v are integers and gcd stands for greatest common divisor. So, in case of

prime field for the inputs x, p we get $xu + pv = 1$ because x, p are co-prime. Which implies that $xu - 1 = pv$ or $xu \equiv 1 \pmod{p}$. The computation of $x^{-1} \in \mathbb{F}_p$ using extended Euclidean algorithms is shown in Algorithm 7.

Algorithm 7 Extended Euclidean Aglorithm for computing inversion in $\mathbb{F}_p$

Input: x, p where $x \in [1, p-1]$ and p is prime

Output: x^{-1} mod p

1: $u_1 \leftarrow 1, u_2 \leftarrow 0, v_1 \leftarrow a, v_2 \leftarrow p$

2: **while** $v_1 \neq 1$ **do**

3: $\quad q \leftarrow \lfloor v_2/v_1 \rfloor$

4: $\quad r \leftarrow v_2 - qv_1, s \leftarrow u_2 - qu_1$

5: $\quad v_2 \leftarrow v_1, v_1 \leftarrow r, u_2 \leftarrow u_1, u_1 \leftarrow s$

6: Return $(u_1 \text{ mod } p)$

Unfortunately, the extended Euclidean algorithm involves the computation of expensive divisions as shown in step 3 of the Algorithm 7. These expensive divisions are replaced by cheaper shift and subtraction operations in the binary inversion algorithm. Actually, the binary inversion algorithm is the extension of the binary gcd algorithm. The Algorithm 8 shows how to compute inversion in $\mathbb{F}_p$ using the binary inversion algorithm.

Fermat's little theorem can also be used to compute inversion in $\mathbb{F}_p$. According to the theorem, for any nonzero element $x \in \mathbb{F}_p$ we have $x^{p-1} \equiv 1 \pmod{p}$ which implies that $xx^{p-2} \equiv 1 \pmod{p}$. Hence, $(x^{p-2} \text{ mod } p)$ is the multiplicative inverse of x in $\mathbb{F}_p$. The most rudimentary and inefficient way of computing inversion in $\mathbb{F}_p$ using Fermat's little theorem is shown as Algorithm 9. The Algorithm 9 is just for introduction and not recommended in implementation. Because it just performs repeated multiplication and does not take advantage of the field squaring that are computationally faster than field multiplication.

Unlike the other prime field operations, inversion is relatively very expensive. If several inversions are required, then Montgomery's trick of simultaneous inversion is very useful. Because instead of computing several inversions separately, Montgomery's trick compute only one inversion and some field multiplications. Now, the

Algorithm 8 Binary Inversion Aglorithm for computing inversion in $\mathbb{F}_p$

Input: x, p where $a \in [1, p-1]$ and p is prime

Output: x^{-1} mod p

$u_1 \leftarrow 1, u_2 \leftarrow 0, v_1 \leftarrow a, v_2 \leftarrow p$
while $v_1 \neq 1$ and $v_2 \neq 1$ **do**
    while v_1 is even **do**
        $v_1 \leftarrow v_1/2$
        if u_1 is even **then** $u_1 \leftarrow u_1/2$
        else $u_1 \leftarrow (u_1 + p)/2$
    while v_2 is even **do**
        $v_2 \leftarrow v_2/2$
        if u_2 is even **then** $u_2 \leftarrow u_2/2$
        else $u_2 \leftarrow (u_2 + p)/2$
    if $v_1 \geq v_2$ **then**
        $v_1 \leftarrow v_1 - v_2, u_1 \leftarrow u_1 - u_2$
    else $v_2 \leftarrow v_2 - v_1, u_2 \leftarrow u_2 - u_1$
if $v_1 = 1$ **then** return $(u_1$ mod $p)$
else return $(u_2$ mod $p)$

Algorithm 9 Fermat's little theorem for computing inversion in $\mathbb{F}_p$

Input: x, p where $x \in [1, p-1]$ and p is prime

Output: x^{p-2} mod p

$u \leftarrow x$
for i from 1 to $p-3$ **do**
    $u \leftarrow ux$ mod p
Return u

Algorithm 10 represents the computation of $x_i^{-1} \in \mathbb{F}_p$ for $i = 1, \ldots, n$ using the Montgomery's trick.

Algorithm 10 Montgomery's trick of simultaneous inversion in $\mathbb{F}_p$

Input: Nonzero elements $x_1, \ldots, x_n \in \mathbb{F}_p$ and p is prime

Output: Nonzero elements $x_1^{-1}, \ldots, x_n^{-1} \in \mathbb{F}_p$ such that $x_i x_i^{-1} \equiv 1 \pmod{p}$

1: $y_1 \leftarrow x_1$
2: **for** i from 2 to n **do**
3: $\quad y_i \leftarrow x_i y_{i-1} \bmod p$
4: $z \leftarrow y_n^{-1} \bmod p$
5: **for** i from n downto 2 **do**
6: $\quad x_i^{-1} \leftarrow z y_{i-1} \bmod p$
7: $\quad z \leftarrow z x_i \bmod p$
8: $x_1 \leftarrow z$
9: Return $(x_1^{-1}, \ldots, x_n^{-1})$

So, the Algorithm 10 computes the inversion of n-element by performing one inversion and $3(n-1)$ field multiplications. Also, $(n+1)$ field elements are required to store the intermediate results.

2.3 Toeplitz Matrix-Vector Product (TMVP) using integers

A Toeplitz or diagonal-constant matrix is a matrix in which each descending diagonal from left to right is constant. An example of $n \times n$ Toeplitz matrix is give below:

$$\begin{bmatrix} x_0 & x_1 & \ldots & \ldots & \ldots & x_{n-1} \\ x_n & x_0 & \ddots & \ddots & \ddots & x_{n-2} \\ \vdots & \ddots & \ddots & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & x_0 & x_1 \\ x_{2(n-1)} & \ldots & \ldots & \ldots & x_n & x_0 \end{bmatrix}$$

Toeplitz matrices have some great properties that can be exploited to achieve efficiency. The first-and-foremost property is that a Toeplitz matrix can be represented

by the first row and the first column of it entries. Hence, it saves space/memory in computer and one can use one-dimensional array as representation. It has also the property that the splitting into sub-matrices, addition and subtraction of a Toeplitz matrix result in Toeplitz matrix. One of the techniques for TMVP is to use the schoolbook matrix-vector product and for size n the arithmetic complexity is $\mathcal{O}(n^2)$. In literature, there are better algorithms than the schoolbook. For example, a leading study on this subject for multiplication over the binary field can be found in [21].

To exploit the properties of Toeplitz matrices — with integer entries — by using the common expression, one should derive the TMVP formula(s) such that addition is restricted to matrix entries. Because addition is both commutative and associative, one can reuse the the result of the common expressions rather than recomputing them. Moreover, there are more (Toeplitz) matrix entries than the vector which implies more operations on matrix entries. Based on our 64-bit and 32-bit implementation we have two scenarios.

2.3.1 Two-way Decomposition

For a TMVP of size 2 we have

$$\begin{bmatrix} x_0 & x_1 \\ x_2 & x_0 \end{bmatrix} \times \begin{bmatrix} y_0 \\ y_1 \end{bmatrix} = \begin{bmatrix} P_2 + P_1 \\ P_3 - P_1 \end{bmatrix} \tag{2.1}$$

where $\quad P_1 = x_0(y_0 - y_1), \quad P_2 = (x_1 + x_0)y_1, \quad P_3 = (x_2 + x_0)y_0.$

The total cost of (2.1) will be $3\mathbf{M} + 3\mathbf{A} + 2\mathbf{A}_d$ where $\mathbf{M}$ is the cost of a single precision/word multiplication, $\mathbf{A}$ is the cost of a single precision/word addition and $\mathbf{A}_d$ is the cost of a double precision/word addition. As compared to the schoolbook matrix-vector product which requires $4\mathbf{M} + 2\mathbf{A}_d$, (2.1) trades $1\mathbf{M}$ for $3\mathbf{A}$. The recursive application of 2.1 results in $\mathcal{O}(n^{\log_2 3})$ complexity of multiplication. Note that we have restricted the addition operation to matrix entries in order to exploit the common expressions at maximum. The concept of “exploiting the common expressions" will become clear in the following chapters.

2.3.2 Three-way Decomposition

For a TMVP of size 3 we have

$$\begin{bmatrix} x_0 & x_1 & x_2 \\ x_3 & x_0 & x_1 \\ x_4 & x_3 & x_0 \end{bmatrix} \times \begin{bmatrix} y_0 \\ y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} P_3 + P_4 + P_6 \\ P_2 - P_4 + P_5 \\ P_1 - P_2 - P_3 \end{bmatrix} \tag{2.2}$$

$$\begin{aligned} \text{where} \quad & P_1 = (x_4 + x_3 + x_0)y_0, \quad P_2 = x_3(y_0 - y_1), \\ & P_3 = x_0(y_0 - y_2), \qquad\quad P_4 = x_1(y_1 - y_2), \\ & P_5 = (x_0 + x_3 + x_1)y_1, \quad P_6 = (x_2 + x_0 + x_1)y_2. \end{aligned}$$

Using the P_i for $i = 1, \ldots, 6$ the total cost of (2.2) will be $6\mathbf{M} + 8\mathbf{A} + 6\mathbf{A}_d$ where **M**, **A** and $\mathbf{A}_d$ are same explained in previous section. The cost of single precision addition is 8 because one can take common either $(x_3 + x_0)$ between P_1 and P_5 or $(x_0 + x_1)$ between P_5 and P_6. In theory, one can say that for all those machines where the cost ratio of multiplication to addition is greater than or equal to 3 then this observation is worth to try. For larger bitlength, we can use this technique recursively and for size n it results in $\mathcal{O}(n^{\log_3 6})$ complexity of multiplication which is better than the schoolbook. Like (2.1), again the addition operation is restricted to matrix entries in order to take advantage of the common expressions.

2.4 Prime shapes

In this section, we briefly introduce the different shapes/forms of primes that we find in the ECC literature. The list may not be comprehensive but it does contain primes that are considered to be the best choices in contemporary ECC.

- **Mersenne Primes:** The primes $2^n - 1$ where n is prime but every prime. Mersenne primes are the best choice to compute the reduction operation (most) efficiently in finite field. Because the reduction is performed by some field addition(s). The bitlength relevant to contemporary ECC leaves us with the only choice of $2^{521} - 1$.
- **Crandall Primes** $(2^k - c)$**:** These primes are a good alternative to the scarcity of Mersenne primes for contemporary ECC and the reduction operation is not

very expensive because c is normally taken as a small integer. Definitely, the efficiency of Crandall primes is directely proportional to the smaller value of c. These primes were introduced by Crandall [18]. These primes have been a popular choice and some of the examples are $2^{255} - 19$, $2^{414} - 17$ etc.

- **Special Montgomery Primes** $(2^k c - 1)$**:** These primes are mainly good in carry propagation when packed-radix is used in implementation. As pointed out by Hamburg in [25] that these primes are not a good choice for prime modulus bitlength greater than 256. Because the vectorized and reduced-radix multiplication that are the efficient ways of implementing larger bitlength ECC. Both these efficient techniques change the carry propagation.

- **Goldilocks Prime** $(2^{448} - 2^{224} - 1)$**:** The prime was introduced by Hamburg in [25] and has several advantages. It enables faster Karastuba multiplication [28] when performed as modular multiplication by saving double-word additions/subtractions. The prime is equal suitable efficient implementation on 32- and 64-bit platforms.

- **Solinas Primes:** These form of primes were introduced by Solinas in [35] and they have become part of the NIST standards [27] for ECC. The primes are of the form $2^k - 2^l \pm \cdots \pm 1$ where the exponents are 32-bit aligned. In other words, these primes are good for 32-bit platforms using packed-radix and the fewer coefficients imply faster reduction.

2.5 Elliptic Curve over Prime Field

An elliptic curve E over a prime field $\mathbb{F}_p$ — characteristic of the $\mathbb{F}_p$ is not equal to 2 and 3 — is a set

$$E(\mathbb{F}_p) = \{(x, y) \in (\mathbb{F}_p \times \mathbb{F}_p) : y^2 = x^3 + ax + b\} \cup \{\infty\}$$

where $a, b \in \mathbb{F}_p$ and the discriminant of the curve is $-16(4a^3 + 27b^2)$. It is required for the elliptic curve that the discriminant of the curve should be nonzero. The point at infinity ∞ is considered to be some infinite point far up the y-axis. The curve equation form, $y^2 = x^3 + ax + b$, is called short Weierstrass form. Actually, the

set $E(\mathbb{F}_p)$ with point addition operation forms a finite abelian group with ∞ as the identity element. The group law is defined as follows

- *Identity:* For all $P \in E(\mathbb{F}_p), P + \infty = \infty + P = P$
- *Negative:* If $P = (x, y) \in E(\mathbb{F}_p)$, then $-P = (x, -y) \in E(\mathbb{F}_p)$ such that $P + (-P) = (x, y) + (x, -y) = \infty$.
- *Point addition:* If $P = (x_1, y_1) \in E(\mathbb{F}_p)$ and $Q = (x_2, y_2) \in E(\mathbb{F}_p)$ such that $P \neq \pm Q$, then $P + Q = (x_3, y_3) \in E(\mathbb{F}_p)$ is defined as follows

$$x_3 = \left(\frac{y_2 - y_1}{x_2 - x_1}\right)^2 - x_1 - x_2$$
$$y_3 = \left(\frac{y_2 - y_1}{x_2 - x_1}\right)(x_1 - x_3) - y_1$$

- *Point doubling:* If $P = (x_1, y_1) \in E(\mathbb{F}_p)$ and $P \neq -P$, then $2P = (x_3, y_3) \in E(\mathbb{F}_p)$ is defined as follows

$$x_3 = \left(\frac{3x_1^2 + a}{2y_1}\right)^2 - 2x_1$$
$$y_3 = \left(\frac{3x_1^2 + a}{2y_1}\right)(x_1 - x_3) - y_1$$

2.6 Elliptic Curve Cryptosystem

The use of elliptic curve over finite fields to construct cryptosystems was proposed independently by Neal Koblitz [29] and V. S. Miller [31]. The operations of public-key generation and shared-secret computation are the direct adaption of the popular Diffie-Hellman key exchange protocol [20] in ECC. Another important cryptosystem based on elliptic curve is the digital signature. These operations are accomplished by the scalar multiplication kP in ECC where k is an integer and P is a point on elliptic curve.

Let's discuss the public-key generation and shared-secret computation using the conventional example of Alice, Bob and Eve. Suppose $P \in E(\mathbb{F}_p)$ is of prime order p' and called standard base point. Alice and Bob secretly and independently select a random integer in $[1, p' - 1]$. Let's denote the secret integers of Alice and Bob by k_A and

k_B, respectively. Alice computes her public key (point) $Q_A = k_A P$ and dispatches it to Bob. Similarly, Bob computes his public key (point) $Q_B = k_B P$ and dispatches it to Alice. Next, Alice and Bob compute the shared-secret (point) on their side as $S = k_A Q_B$ and $S = k_B Q_A$, respectively. Eve is intercepting all these traffics over the insecure channel between Alice and Bob in order to spy on them. So, Eve knows P, Q_A, Q_B and her challenge is to determine the secret integer k_A or k_B in order to break the system. The problem of finding an integer k such that $Q = kP$ is called elliptic-curve discrete-logarithm problem (ECDLP) and the security of ECC relies on the hardness of ECDLP.

2.7 Point representation

In Section 2.5, the point $P = (x, y) \in E(\mathbb{F}_p)$ and curve equation are both given in affine coordinates. Furthermore, the group law was also defined in terms of affine coordinates. But one can also use the projective coordinates for $E(\mathbb{F}_p)$ because there is a $1-1$ correspondence between the set of projective points and affine points. Unlike an affine point, a projective point is represented as a triplet $(X : Y : Z)$. Actually, projective coordinates are very useful in accelerating the curve operations. Some of the proposed projective coordinates are given below.

- *Standard projective coordinates:* The affine point (x, y) represents the projective point $(X : Y : Z)$ with $Z \neq 0$ such that $x = X/Z$ and $y = Y/Z$. In this coordinates system, the elliptic curve equation becomes $Y^2Z = X^3 + aXZ^2 + bZ^3$. The additive inverse or negative of $(X : Y : Z)$ is $(X : -Y : Z)$ and the identity ∞ corresponds to $(0 : 1 : 0)$.

- *Jacobian coordinates:* The affine point (x, y) represents the jacobian point $(X : Y : Z)$ with $Z \neq 0$ such that $x = X/Z^2$ and $y = Y/Z^3$. So, the elliptic curve equation becomes $Y^2 = X^3 + aXZ^4 + bZ^6$. The negative of $(X : Y : Z)$ is $(X : -Y : Z)$ and the identity ∞ corresponds to $(1 : 1 : 0)$.

- *Chudnovsky coordinates:* The chudnovsky coordinates are redundant representation of jacobian coordinates so, the jocobian coordinates $(X : Y : Z)$ with

$Z \neq 0$ represents the chudnovsky coordinates $(X : Y : Z : Z^2 : Z^3)$. This coordinates system is also called Chudnovsky-Jacobian coordinates.

- *Extended coordinates:* The coordinates were introduced for twisted Edwards curve using extended affine coordinates (x, y, t) where $t = xy$. The point (x, y, t) corresponds to the extended point $(X : Y : T : Z)$ with $Z \neq 0$ and $T = XY/Z$ such that $x = X/Z, y = Y/Z$ and $t = T/Z$. The negative of $(X : Y : T : Z)$ is $(-X : Y : -T : Z)$ and the identity is $(0 : 1 : 0 : 1)$. The extended coordinates were introduced by Hisil et al. [26].

The purpose of these different coordinates system is to accelerate the point arithmetic operations in order to have more efficient elliptic curve schemes. Bernstein and Lange maintain a website [9], Explicit-Formulas Database (EFD), where on can find the latest and efficient point arithmetic formulas in different forms of the curve using different coordinate systems.

CHAPTER 3

64-BIT RESIDUE MULTIPLICATION

Recall that for a prime number p there exist a finite field $\mathbb{F}$ of order p and the field is denoted by $\mathbb{F}_p$. In our case, the prime number $p = 2^{521} - 1$ and called 521-bit Mersenne prime. Among the prime field operations, our objective is a new efficient way of computing field multiplication in $\mathbb{F}_p$. Our objective is also supported by Hankerson et al. in their book [19] that the performance of ECC depends heavily on the speed of field multiplication. Hence, we use our residue multiplication to improve the efficient computation of single-scalar multiplication.

In this chapter, we discuss in detail, our residue multiplication modulo p and the target is 64-bit platforms. We present three versions of our 64-bit residue multiplication in order to provide an extensive arithmetic cost comparison with respect to the other well-known algorithms. We also discuss the implementation details and the timing results to select the most efficient version of our algorithm in terms of clock cycles.

3.1 Residue multiplication using TMVP

Suppose X and Y are 521-bit integers and we want to compute $Z \equiv XY \pmod{p}$ where $p = 2^{521} - 1$. Clearly, a 521-bit integer cannot be represented as a single-word on 64-bit computers. Therefore, one splits 521-bit integer in smaller chunks/limbs using either packed-radix or reduced-radix. Like Granger and Scott [23], we use the same reduced-radix representation because it is very suitable for this scenario. The use of reduced-radix has shown interesting results and some of the examples are [1, 3, 5, 6, 7, 11, 16, 23, 25, 34]. Hence, we are computing residue multiplication

modulo p on 64-bit computers using reduced-radix.

As performed in [23], the operands X, Y are split in nine limbs where each limb comprises of at most 58-bit, stored in a 64-bit word. So, one can represent X, Y as follows:

$$X = x_0 + 2^{58}x_1 + 2^{116}x_2 + 2^{174}x_3 + 2^{232}x_4 + 2^{290}x_5 + 2^{348}x_6 + 2^{406}x_7 + 2^{464}x_8$$
$$Y = y_0 + 2^{58}y_1 + 2^{116}y_2 + 2^{174}y_3 + 2^{232}y_4 + 2^{290}y_5 + 2^{348}y_6 + 2^{406}y_7 + 2^{464}y_8$$

Unlike [23], we take the limbs x_8 and y_8 as 57-bit. In other words, we are interested to work with modulus p. Like [23], the choice of modulus $2p$ is open and in that case, x_8 and y_8 are taken 58-bit. We believe modulus p is an efficient approach when modular multiplication is performed thousands of times i.e. scalar multiplication in ECC. Let $Z = [z_0, z_1, z_2, z_3, z_4, z_5, z_6, z_7, z_8]$ where

$$\begin{aligned}
z_0 &= x_0y_0 + 2(x_8y_1 + x_7y_2 + x_6y_3 + x_5y_4 + x_4y_5 + x_3y_6 + x_2y_7 + x_1y_8),\\
z_1 &= x_1y_0 + x_0y_1 + 2(x_8y_2 + x_7y_3 + x_6y_4 + x_5y_5 + x_4y_6 + x_3y_7 + x_2y_8),\\
z_2 &= x_2y_0 + x_1y_1 + x_0y_2 + 2(x_8y_3 + x_7y_4 + x_6y_5 + x_5y_6 + x_4y_7 + x_3y_8),\\
z_3 &= x_3y_0 + x_2y_1 + x_1y_2 + x_0y_3 + 2(x_8y_4 + x_7y_5 + x_6y_6 + x_5y_7 + x_4y_8),\\
z_4 &= x_4y_0 + x_3y_1 + x_2y_2 + x_1y_3 + x_0y_4 + 2(x_8y_5 + x_7y_6 + x_6y_7 + x_5y_8),\\
z_5 &= x_5y_0 + x_4y_1 + x_3y_2 + x_2y_3 + x_1y_4 + x_0y_5 + 2(x_8y_6 + x_7y_7 + x_6y_8),\\
z_6 &= x_6y_0 + x_5y_1 + x_4y_2 + x_3y_3 + x_2y_4 + x_1y_5 + x_0y_6 + 2(x_8y_7 + x_7y_8),\\
z_7 &= x_7y_0 + x_6y_1 + x_5y_2 + x_4y_3 + x_3y_4 + x_2y_5 + x_1y_6 + x_0y_7 + 2x_8y_8,\\
z_8 &= x_8y_0 + x_7y_1 + x_6y_2 + x_5y_3 + x_4y_4 + x_3y_5 + x_2y_6 + x_1y_7 + x_0y_8
\end{aligned} \tag{3.1}$$

The constant 2 in (3.1) appears as a result of the reduction and there are two important things that need to taken into account. Firstly, it should be guaranteed that the maximum value of z_i can not be greater than $2^{128} - 1$ for $i \in \{1, \ldots, 8\}$. In other words, overflow should be avoid in double-word (128-bit) to ensure the correctness of the values. Secondly, each z_i is a double-word and it contains the carry that should be propagated to the next limb. We observed that the whole expressions in (3.1) can be presented as a TMVP as shown in (3.2). One can compute the TMVP using the schoolbook matrix-vector product method. Which results in the complexity of $\mathcal{O}(n^2)$ multiplications for matrix-vector product of size n. There are efficient ways of computing TMVP other than the schoolbook method and one can find better algorithms

in literature. For example, a leading study on this subject for multiplication over $\mathbb{F}_2$ can be found in [21].

$$\begin{bmatrix} x_0 & 2x_8 & 2x_7 & 2x_6 & 2x_5 & 2x_4 & 2x_3 & 2x_2 & 2x_1 \\ x_1 & x_0 & 2x_8 & 2x_7 & 2x_6 & 2x_5 & 2x_4 & 2x_3 & 2x_2 \\ x_2 & x_1 & x_0 & 2x_8 & 2x_7 & 2x_6 & 2x_5 & 2x_4 & 2x_3 \\ x_3 & x_2 & x_1 & x_0 & 2x_8 & 2x_7 & 2x_6 & 2x_5 & 2x_4 \\ x_4 & x_3 & x_2 & x_1 & x_0 & 2x_8 & 2x_7 & 2x_6 & 2x_5 \\ x_5 & x_4 & x_3 & x_2 & x_1 & x_0 & 2x_8 & 2x_7 & 2x_6 \\ x_6 & x_5 & x_4 & x_3 & x_2 & x_1 & x_0 & 2x_8 & 2x_7 \\ x_7 & x_6 & x_5 & x_4 & x_3 & x_2 & x_1 & x_0 & 2x_8 \\ x_8 & x_7 & x_6 & x_5 & x_4 & x_3 & x_2 & x_1 & x_0 \end{bmatrix} \times \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \\ y_8 \end{bmatrix} \tag{3.2}$$

3.1.1 Proposed Technique

Since the size of the TMVP (3.2) is 9, therefore, one can apply the three-way decomposition (2.2) to decompose and compute (3.2) as follows:

$$\left[\begin{array}{c|c|c} X_0 & 2X_2 & 2X_1 \\ \hline X_1 & X_0 & 2X_2 \\ \hline X_2 & X_1 & X_0 \end{array}\right] \times \left[\begin{array}{c} Y_0 \\ \hline Y_1 \\ \hline Y_2 \end{array}\right] = \begin{bmatrix} P_3 + P_4 + P_6 \\ P_2 - P_4 + P_5 \\ P_1 - P_2 - P_3 \end{bmatrix}$$

where the sub-matrices X_i and the sub-vectors Y_i are of size 3 for $i = 0, 1, 2$. Note that the sub-matrices $2X_2$ and $2X_1$ provide the opportunity to exploit the common expressions by simplifying the formulas in (2.2) as shown below.

$$\begin{aligned} P_1 &= (X_2 + X_1 + X_0)Y_0, & P_2 &= X_1(Y_0 - Y_1) \\ P_3 &= X_0(Y_0 - Y_2), & P_4 &= 2X_2(Y_1 - Y_2) \\ P_5 &= (X_0 + X_1 + 2X_2)Y_1, & P_6 &= (2(X_2 + X_1) + X_0)Y_2. \end{aligned}$$

In this technique, we perform the schoolbook method when the size of matrix-vector product becomes 3. From here onwards, we represent a Toeplitz matrix by its first row and first column because by definition we know that the descending diagonal entries from left to right are constant, therefore, we leave the other entries blank. So, in a programming language a Toeplitz matrix can be presented by one-dimensional array rather than two-dimensional array. Since we are working on a 64-bit computer, therefore, in this chapter by single-word we mean 64-bit and double-word 128-bit.

Computing P_2:

$$Y_0 - Y_1 = \begin{bmatrix} y_0 \\ y_1 \\ y_2 \end{bmatrix} - \begin{bmatrix} y_3 \\ y_4 \\ y_5 \end{bmatrix} = \begin{bmatrix} y_0 - y_3 \\ y_1 - y_4 \\ y_2 - y_5 \end{bmatrix} = \begin{bmatrix} U_1 \\ U_2 \\ U_3 \end{bmatrix}$$

$$X_1(Y_0 - Y_1) = \begin{bmatrix} x_3 & x_2 & x_1 \\ x_4 & & \\ x_5 & & \end{bmatrix} \times \begin{bmatrix} U_1 \\ U_2 \\ U_3 \end{bmatrix}$$

Hence, the total arithmetic cost of the TMVP P_2 is $9\mathbf{M} + 3\mathbf{A} + 6\mathbf{A}_d$.

Computing P_4:

$$Y_1 - Y_2 = \begin{bmatrix} y_3 \\ y_4 \\ y_5 \end{bmatrix} - \begin{bmatrix} y_6 \\ y_7 \\ y_8 \end{bmatrix} = \begin{bmatrix} y_3 - y_6 \\ y_4 - y_7 \\ y_5 - y_8 \end{bmatrix} = \begin{bmatrix} U_7 \\ U_8 \\ U_9 \end{bmatrix}$$

$$2X_2(Y_1 - Y_2) = \begin{bmatrix} 2x_6 & 2x_5 & 2x_4 \\ 2x_7 & & \\ 2x_8 & & \end{bmatrix} \times \begin{bmatrix} U_7 \\ U_8 \\ U_9 \end{bmatrix}$$

Hence, the total arithmetic cost of the TMVP P_4 is $9\mathbf{M} + 3\mathbf{A} + 6\mathbf{A}_d + 5$**-shift** where the (left) shifts are due to multiplication of x_i by 2 for $i = 4, \ldots, 8$. Note that, some of the elements of $2X_2$ appear in different places therefore, it is computed once and used in different places.

Computing P_3:

$$Y_0 - Y_2 = \begin{bmatrix} y_0 \\ y_1 \\ y_2 \end{bmatrix} - \begin{bmatrix} y_6 \\ y_7 \\ y_8 \end{bmatrix} = \begin{bmatrix} y_0 - y_6 \\ y_1 - y_7 \\ y_2 - y_8 \end{bmatrix} = \begin{bmatrix} U_4 \\ U_5 \\ U_6 \end{bmatrix}$$

$$X_0(Y_0 - Y_2) = \begin{bmatrix} x_0 & 2x_8 & 2x_7 \\ x_1 & & \\ x_2 & & \end{bmatrix} \times \begin{bmatrix} U_4 \\ U_5 \\ U_6 \end{bmatrix}$$

Where $2x_8$ and $2x_7$ have already been computed by P_4. Hence, the total arithmetic cost of the TMVP P_3 is $9\mathbf{M} + 3\mathbf{A} + 6\mathbf{A}_d$.

Computing P_1:

$$X_2 + X_1 = \begin{bmatrix} x_6 & x_5 & x_4 \\ x_7 & & \\ x_8 & & \end{bmatrix} + \begin{bmatrix} x_3 & x_2 & x_1 \\ x_4 & & \\ x_5 & & \end{bmatrix}$$

$$= \begin{bmatrix} x_6 + x_3 & x_5 + x_2 & x_4 + x_1 \\ x_7 + x_4 & & \\ x_8 + x_5 & & \end{bmatrix} = \begin{bmatrix} S_1 & S_2 & S_3 \\ S_4 & & \\ S_5 & & \end{bmatrix}$$

$$(X_2 + X_1) + X_0 = \begin{bmatrix} S_1 & S_2 & S_3 \\ S_4 & & \\ S_5 & & \end{bmatrix} + \begin{bmatrix} x_0 & 2x_8 & 2x_7 \\ x_1 & & \\ x_2 & & \end{bmatrix} = \begin{bmatrix} S_6 & S_7 & S_8 \\ S_9 & & \\ S_{10} & & \end{bmatrix}$$

$$(X_2 + X_1 + X_0)Y_0 = \begin{bmatrix} S_6 & S_7 & S_8 \\ S_9 & & \\ S_{10} & & \end{bmatrix} \times \begin{bmatrix} y_0 \\ y_1 \\ y_2 \end{bmatrix}$$

Again $2x_8$ and $2x_7$ are used but already computed by P_4. Hence, the total arithmetic cost of the TMVP P_1 is $9\mathbf{M} + 10\mathbf{A} + 6\mathbf{A}_d$.

Computing P_6: For the sub-matrices addition, we have

$$2(X_2 + X_1) + X_0 = (X_2 + X_1 + X_0) + (X_2 + X_1)$$

and we have already computed both the parenthesized expressions on the right-hand side so

$$(X_2 + X_1 + X_0) + (X_2 + X_1) = \begin{bmatrix} S_6 & S_7 & S_8 \\ S_9 & & \\ S_{10} & & \end{bmatrix} + \begin{bmatrix} S_1 & S_2 & S_3 \\ S_4 & & \\ S_5 & & \end{bmatrix}$$

$$= \begin{bmatrix} S_{11} & S_{12} & S_{13} \\ S_{14} & & \\ S_{15} & & \end{bmatrix}$$

$$((X_2 + X_1 + X_0) + (X_2 + X_1))Y_2 = \begin{bmatrix} S_{11} & S_{12} & S_{13} \\ S_{14} & & \\ S_{15} & & \end{bmatrix} \times \begin{bmatrix} y_6 \\ y_7 \\ y_8 \end{bmatrix}$$

Hence, the total arithmetic cost of the TMVP P_6 is $9\mathbf{M} + 5\mathbf{A} + 6\mathbf{A}_d$.

Computing P_5: Here, for the addition of sub-matrices one has to compute $S_{16} = S_6 + f_6$ only. While the other four elements have already been computed i.e. two by P_1 and two by P_6 as shown below

$$(X_0 + X_1) + 2X_2 = \begin{bmatrix} x_0 + x_3 & 2x_8 + x_2 & 2x_7 + x_1 \\ x_1 + x_4 & & \\ x_2 + x_5 & & \end{bmatrix} + \begin{bmatrix} 2x_6 & 2x_5 & 2x_4 \\ 2x_7 & & \\ 2x_8 & & \end{bmatrix}$$

$$= \begin{bmatrix} S_{16} & S_{15} & S_{14} \\ S_8 & & \\ S_7 & & \end{bmatrix}$$

$$(X_0 + X_1 + 2X_2)Y_1 = \begin{bmatrix} S_{16} & S_{15} & S_{14} \\ S_8 & & \\ S_7 & & \end{bmatrix} \times \begin{bmatrix} y_3 \\ y_4 \\ y_5 \end{bmatrix}$$

Hence, the total arithmetic cost of the TMVP P_5 is $9\mathbf{M} + 1\mathbf{A} + 6\mathbf{A}_d$.

Final Computation: At last, we have to compute

$$\begin{bmatrix} P_3 + P_4 + P_6 \\ P_2 - P_4 + P_5 \\ P_1 - P_2 - P_3 \end{bmatrix}$$

where each P_i is a 3×1 vector and the elements are of double-word size so the total arithmetic cost is $18\mathbf{A}_d$. Now, if we sum up all the costs, then the overall cost of the whole technique is $54\mathbf{M} + 25\mathbf{A} + 54\mathbf{A}_d + 5$**-shift**.

Alternatively, one can take $P_1 = (X_2+(X_1+X_0))Y_0, P_5 = ((X_0+X_1)+2X_2)Y_1, P_6 = ((2X_2 + (X_1 + X_0)) + X_1)Y_2$ where P_2, P_3, P_4 remain unchanged and $2X_2$ is computed once. But we found the total arithmetic cost as $54\mathbf{M} + 29\mathbf{A} + 54\mathbf{A}_d + 5$**-shift** which is more than the aforementioned technique.

3.2 Algorithms and Comparison

The psuedocode of our proposed technique, discussed in detail in Section 3.1.1, is presented as Algorithm 11. Our technique provides the freedom to compute P_i in different ways so that one can achieve efficient implementation on the machine at one's disposal, for $i \in \{1, \ldots, 6\}$. We introduce three versions of our algorithm to achieve two objectives: extensive arithmetic cost comparison, and efficient implementation. To provide an extensive arithmetic cost comparison, we have computed the total arithmetic cost of the three versions of our algorithm separately with respect to its counterpart(s). The counterparts are based on using the well-known algorithms. The objective of efficient implementation is achieved by testing the three versions of our algorithm on 64-bit computer for clock cycles count. The three versions of our algorithm are discussed in detail in the following sections and they are: (i) Hybrid version, (ii) Recursive version, and (iii) Mixed version.

3.2.1 Residue Representation

In this section, we explain the representation of a residue modulo p using the same idea as in [23]. As aforementioned, our work is directly based on modulus p, whereas the modulus $2p$ was chosen in [23]. One can easily switch between modulus p and $2p$ with some (minor) changes. Like [23], the residue representation is strictly followed for the output coordinates of point addition and doubling routines. Likewise, the coordinates within the respective routines of point addition and doubling do not need to follow the residue representation except for the output of the coordinate multiplication and squaring.

From our residue multiplication, we find through testing that if either of the inputs/operands (X or Y) is $2^{521} - 2$ and the other in $[2^{521} - 17, 2^{521} - 2]$, then the output limbs are in $[0, 2^{59} - 1] \times [0, 2^{58} - 1]^7 \times [0, 2^{57} - 1]$. The least significant limb z_0 and the most significant limb z_8 are in $[0, 2^{59} - 1]$ and $[0, 2^{57} - 1]$, respectively. For all the other residue values as inputs, carry propagation in our residue multiplication results in the unique residue modulo p. Hence, the output of our residue multiplication is always in $[0, 2^{59} - 1] \times [0, 2^{58} - 1]^7 \times [0, 2^{57} - 1]$. Similarly, the

selection of modulus p also requires changes in squaring algorithm. This time, we find through testing that if the input is in $[2^{521} - 5, 2^{521} - 2]$, then the output will be in $[0, 2^{59} - 1] \times [0, 2^{58} - 1]^7 \times [0, 2^{57} - 1]$.

Like [23], for the scalar mulitplication in ECC, the intermediate results are not fully reduced to the unique residue in modulo p. Instead, the output coordinates of point addition and doubling routines are ensured to be in $[0, 2^{59}-1] \times [0, 2^{58}-1]^7 \times [0, 2^{57}-1]$ using the idea of *short coefficient reduction* (SCR) routine in [23].

3.2.2 Implementation Results

For the timing tests, we use Ubuntu 16.04 LTS on an Intel Pentium CPU G2010 @ 2.80GHz desktop machine with 4GB RAM and the Turbo Boost being disabled. There are two cores and from BIOS one can change the number of cores. Therefore, we have tested our programs both with one core and two cores. We find testing on two cores a better choice especially in case of scalar multiplication in ECC.

We have implemented all the three versions of our residue multiplication in C language using GCC $5.3.1$. For the clock cycles count, we use the technique proposed by Paoloni in his white paper [32]. All the three versions of our algorithm are tested on the same set of 10^3 (random) integers by calling the function twice in 10^3 iterations loop. The values are read limb-by-limb from separate files for each operand. We find 164 as the minimum mean cycles count for the multiplication function, gmul(), of Scott which is more than the report cycles of 155 in [23]. Although our interest is in modulus p but we provide the timing results of our residue multiplication using both modulus p and $2p$.

3.2.2.1 Hybrid version

In this version, first we apply (2.2) for the matrix-vector decomposition to obtain P_i for $i = 1, \ldots, 6$ then use the schoolbook matrix-vector product to compute each P_i. This version [1] is already explained in detail in Section 3.1.1 and the pseudo-code is

[1] https://github.com/Shoukat-Ali/521-bit-Mersenne-Prime/blob/master/hybrid.c

Algorithm 11 (Hybrid version) 64-bit residue Multiplication

Input: $X = [x_0, \ldots\ldots, x_8], Y = [y_0, \ldots\ldots, y_8] \in [0, 2^{59} - 1] \times [0, 2^{58} - 1]^7 \times [0, 2^{57} - 1]$

Output: $Z = [z_0, \ldots\ldots, z_8] \in [0, 2^{59}-1] \times [0, 2^{58}-1]^7 \times [0, 2^{57}-1]$ where $Z \equiv XY \pmod{2^{521} - 1}$

$T_5 \leftarrow 2x_8, \quad c \leftarrow 2x_7, \quad T_1[3] \leftarrow 2x_6, \quad T_1[4] \leftarrow 2x_5$

$T_1[0] \leftarrow y_0 - y_3, \quad T_1[1] \leftarrow y_1 - y_4, \quad T_1[2] \leftarrow y_2 - y_5$

$P_0 \leftarrow (x_3 \cdot T_1[0]) + (x_2 \cdot T_1[1]) + (x_1 \cdot T_1[2])$

$P_1 \leftarrow (x_4 \cdot T_1[0]) + (x_3 \cdot T_1[1]) + (x_2 \cdot T_1[2])$

$P_2 \leftarrow (x_5 \cdot T_1[0]) + (x_4 \cdot T_1[1]) + (x_3 \cdot T_1[2])$

$T_1[0] \leftarrow y_3 - y_6, \quad T_1[1] \leftarrow y_4 - y_7, \quad T_1[2] \leftarrow y_5 - y_8$

$P_6 \leftarrow (T_1[3] \cdot T_1[0]) + (T_1[4] \cdot T_1[1]) + (2x_4 \cdot T_1[2])$

$P_7 \leftarrow (c \cdot T_1[0]) + (T_1[3] \cdot T_1[1]) + (T_1[4] \cdot T_1[2])$

$P_8 \leftarrow (T_5 \cdot T_1[0]) + (c \cdot T_1[1]) + (T_1[3] \cdot T_1[2])$

$T_1[0] \leftarrow y_0 - y_6, \quad T_1[1] \leftarrow y_1 - y_7, \quad T_1[2] \leftarrow y_2 - y_8$

$P_3 \leftarrow (x_0 \cdot T_1[0]) + (T_5 \cdot T_1[1]) + (c \cdot T_1[2])$

$P_4 \leftarrow (x_1 \cdot T_1[0]) + (x_0 \cdot T_1[1]) + (T_5 \cdot T_1[2])$

$P_5 \leftarrow (x_2 \cdot T_1[0]) + (x_1 \cdot T_1[1]) + (x_0 \cdot T_1[2])$

$T_6[0] \leftarrow x_4 + x_1, \quad T_6[1] \leftarrow x_5 + x_2, \quad T_6[2] \leftarrow x_6 + x_3$

$T_6[3] \leftarrow x_7 + x_4, \quad T_6[4] \leftarrow x_8 + x_5$

$T_1[0] \leftarrow T_6[0] + c, \quad T_1[1] \leftarrow T_6[1] + T_5, \quad T_1[2] \leftarrow T_6[2] + x_0$

$T_1[3] \leftarrow T_6[3] + x_1, \quad T_1[4] \leftarrow T_6[4] + x_2$

$T_6[0] \leftarrow T_1[0] + T_6[0], \quad T_6[1] \leftarrow T_1[1] + T_6[1], \quad T_6[2] \leftarrow T_1[2] + T_6[2]$

$T_6[3] \leftarrow T_1[3] + T_6[3], \quad T_6[4] \leftarrow T_1[4] + T_6[4]$

$T_5 \leftarrow T_1[2] + x_6$

$C \leftarrow (T_1[2] \cdot y_2) + (T_1[3] \cdot y_1) + (T_1[4] \cdot y_0) - P_2 - P_5$

$c \leftarrow C \mod 2^{57}$

$C \leftarrow (T_6[0] \cdot y_8) + (T_6[1] \cdot y_7) + (T_6[2] \cdot y_6) + P_3 + P_6 + (C \gg 57)$

$z_0 \leftarrow C \mod 2^{58}$

$C \leftarrow (T_6[1] \cdot y_8) + (T_6[2] \cdot y_7) + (T_6[3] \cdot y_6) + P_4 + P_7 + (C \gg 58)$

$z_1 \leftarrow C \mod 2^{58}$

Algorithm 12 (Hybrid version) 64-bit residue Multiplication: Continued

$C \leftarrow (T_6[2] \cdot y_8) + (T_6[3] \cdot y_7) + (T_6[4] \cdot y_6) + P_5 + P_8 + (C \ \gg \ 58)$

$z_2 \leftarrow C \ \bmod \ 2^{58}$

$C \leftarrow (T_6[3] \cdot y_5) + (T_6[4] \cdot y_4) + (T_5 \cdot y_3) + P_0 - P_6 + (C \ \gg \ 58)$

$z_3 \leftarrow C \ \bmod \ 2^{58}$

$C \leftarrow (T_6[4] \cdot y_5) + (T_5 \cdot y_4) + (T_1[0] \cdot y_3) + P_1 - P_7 + (C \ \gg \ 58)$

$z_4 \leftarrow C \ \bmod \ 2^{58}$

$C \leftarrow (T_5 \cdot y_5) + (T_1[0] \cdot y_4) + (T_1[1] \cdot y_3) + P_2 - P_8 + (C \ \gg \ 58)$

$z_5 \leftarrow C \ \bmod \ 2^{58}$

$C \leftarrow (T_1[0] \cdot y_2) + (T_1[1] \cdot y_1) + (T_1[2] \cdot y_0) - P_0 - P_3 + (C \ \gg \ 58);$

$z_6 \leftarrow C \ \bmod \ 2^{58}$

$C \leftarrow (T_1[1] \cdot y_2) + (T_1[2] \cdot y_1) + (T_1[3] \cdot y_0) - P_1 - P_4 + (C \ \gg \ 58)$

$z_7 \leftarrow C \ \bmod \ 2^{58}$

$c \leftarrow c + (C \ \gg \ 58)$

$z_8 \leftarrow c \ \bmod \ 2^{57}$

$z_0 \leftarrow z_0 + (c \ \gg \ 57)$

Return Z

given as Algorithm 11. After testing and running multiple times we find the minimum mean clock cycles count as 179 and 181 at -O3 for modulus p and $2p$ respectively.

3.2.2.2 Mixed version

Rather applying (2.2) individually on each P_i for $i = 1, \ldots, 6$ one can further exploit the formula to find the common expressions (having same result) on the matrix elements of P_i. Unfortunately, the vectors do not have common expressions. Which implies that one has to exploit the $P_{2,i}$ for $i = 1, 5, 6$ where $P_{2,i}$ represent the P_i when (2.2) is applied the second time or call it level-2. From (2.2), we know that within a P_i there is one intra-common expression — involving two elements — between two $P_{2,i}$ and therefore, one has to exploit the third one for inter-common expression with other P_i. For example, for P_2 we have $P_{2,1} = (x_3 + x_4 + x_5)U_1, P_{2,5} = (x_2 + x_3 + x_4)U_2, P_{2,6} = (x_1 + x_2 + x_3)U_3$ and by taking $x_3 + x_4$ as intra-common, leaves $x_1 + x_2 + x_3$ for inter-common.

The total number of single-word addition can be reduced if one finds inter-common expression among more than two P_i for $i = 1, \ldots, 6$. However, in this particular case we find commonality between two P_i only. There are two candidate groups $\{P_2, P_3, P_4\}$ and $\{P_1, P_5, P_6\}$ for inter-common expression. In our implementation, we have taken $\{P_2, P_3\}$ and $\{P_5, P_6\}$ for applying (2.2) at second level to exploit the inter-common expression while using schoolbook for P_4 and P_1. Instead of two P_i one may take either $\{P_2, P_3, P_4\}$ or $\{P_1, P_5, P_6\}$ for inter-common expression but that will not reduce the total number of single-word addition because there is no inter-common expression among more than two P_i. Since at level-2 all the elements of the Toeplitz sub-matrix are independent therefore, it is impossible to find inter-common expression involving more than two matrix elements.

Based on the arithmetic cost, we have tested three implementations of this version: (i) computing P_2, P_3, P_4 by applying (2.2) through a function call, (ii) in-lining the computation of P_2, P_3, P_4 rather making a separate call, and (iii) computing P_2, P_3 and P_5, P_6 through function calls as discussed above. We find (ii) as the optimal implementation [2] and the minimum mean cycles count as 195 and 197 at -O3 for

[2] https://github.com/Shoukat-Ali/521-bit-Mersenne-Prime/blob/master/mixed_inline.c

modulus p and $2p$ respectively.

3.2.2.3 Recursive version

Instead of applying schoolbook to compute P_i for $i = 1, \ldots, 6$ when the TMVP size becomes 3, one may re-apply (2.2). For this version, we have tested three implementations: (i) function calls, (ii) in-lining rather than making calls, and (iii) using the Mixed version exploitation for $\{P_2, P_3\}$ and $\{P_5, P_6\}$ where P_1, P_4 are computed by additional function calls. Note that (iii) is implemented differently from the Mixed version. Also, for all P_i we apply (2.2) no matter how the inter-common expressions are exploited. We find (i) as the optimal implementation [3] and the minimum mean cycles count as 193 and 194 at -O3 for modulus p and $2p$ respectively.

3.2.3 Arithmetic Cost Comparison

In the earlier sections, we discussed the three versions of our residue multiplication and provided their respective timing results. Now, in this section, we provide the arithmetic cost of each version of our residue multiplication along with its counterpart(s) by using the well-known algorithms. The objective is twofold: firstly, to provide an extensive arithmetic cost comparison; and secondly, to show the robustness of our residue multiplication modulo 521-bit Mersenne prime on 64-bit platforms. Since the number of limbs is nine, therefore, the algorithms of Karatusba 3-way and Toom-Cook (or Toom-3) are good candidates in this situation. We have selected the Karatsuba 3-way formula defined by Weimerskirch and Paar in [36]. Similarly, for the Toom-3, we have selected the optimized formula of Bodrato in [12]. We did not find the recursive version of Toom-3 useful for two reasons: (i) the trading of $4\mathbf{M}$ for $18\mathbf{A}$ plus some shifts and a division by 3 is not good at this situation, and (ii) we could not find any common expression(s) between the two levels.

We find the arithmetic cost of each selected algorithm as provided in Table 3.1 and the cost of shifts, division by small constant(s) and carry propagation are precluded. Because we did not find the precluded costs very significant. The arithmetic cost of

[3] https://github.com/Shoukat-Ali/521-bit-Mersenne-Prime/blob/master/recursive_v1.c

Table 3.1: Number of operations for modular multiplication, [1] the cost of **MUL** algorithm, [2] the cost of the optimal implementation in terms of cycles count

Technique	**Arithmetic cost**
Karatsuba 3-way recursive [36]	$36\mathbf{M} + 54\mathbf{A} + 93\mathbf{A}_d$
Recursive version[2] (this chapter)	$36\mathbf{M} + 73\mathbf{A} + 54\mathbf{A}_d$
Toom-3 plus Schoolbook [12]	$45\mathbf{M} + 30\mathbf{A} + 76\mathbf{A}_d$
Granger-Scott[1] [23]	$45\mathbf{M} + 72\mathbf{A} + 52\mathbf{A}_d$
Mixed version[2] (this chapter)	$45\mathbf{M} + 48\mathbf{A} + 54\mathbf{A}_d$
Karatsuba 3-way plus Schoolbook [36]	$54\mathbf{M} + 18\mathbf{A} + 75\mathbf{A}_d$
Hybrid version (this chapter)	$54\mathbf{M} + 25\mathbf{A} + 54\mathbf{A}_d$

all the algorithms in Table 3.1 include the cost of reduction operation. It is evident from the Table 3.1 that each version of our residue multiplication has lesser arithmetic cost than its counterpart(s) for residue multiplication modulo 521-bit Mersenne prime on 64-bit platform.

CHAPTER 4

32-BIT RESIDUE MULTIPLICATION

In the last chapter, we discussed in detail our 64-bit residue multiplication modulo 521-bit Mersenne prime and by following the same TMVP approach on 32-bit platforms, the new size of TMVP becomes double and the bit-length of the its entries is reduced by half. But these changes in sizes — the size of TMVP and its entries on 32-bit platform — bring different challenges that directly affect the arithmetic cost and timing result. Unlike the 64-bit residue multiplication, the free bit space in 32-bit word of a limb shrinks and as a result the number of single-word operations reduces in order to avoid overflow. In other words, overflow is avoided at the cost of efficiency in arithmetic cost and timing result. Our objective of speeding up the computation of single-scalar multiplication using our residue multiplication is also affected by the changes in sizes and the related issues are discussed in detail in Section 5.4.

In this Chapter, we discuss in detail our three techniques to perform residue multiplication modulo 521-bit Mersenne prime on 32-bit platforms. Unlike the 64-bit residue multiplication, this time we work only in modulus $p = 2^{521} - 1$. We also discuss our strategies of dealing with the challenges that arise due to the change in the size of the TMVP and its entries. For the implementation result, we find the size 6 as the cross-over point between the schoolbook and TMVP. The comparison of our three techniques to other algorithms with respect to the arithmetic cost and the timing results are provided in this chapter. Lastly, based on the timing results of our techniques, we select the one with the least number of clock cycles count and provide the worst-case bitlength analysis to show that overflow can not occur in that technique.

4.1 Residue multiplication using TMVP

Recall that X and Y are 521-bit integers and we want to compute $Z \equiv XY \pmod{p}$ where $p = 2^{521} - 1$. Our 64-bit residue multiplication implies that on 32-bit platforms the number of limbs will become double and the bitlength of the limbs will reduce by half. In other words, for 32-bit implementation we have 18-limb such that each limb can be at most 29-bit, as shown below.

$$X = x_0 + 2^{29}x_1 + 2^{58}x_2 + \cdots + 2^{464}x_{16} + 2^{493}x_{17}$$
$$Y = y_0 + 2^{29}y_1 + 2^{58}y_2 + \cdots + 2^{464}y_{16} + 2^{493}y_{17}$$

The limbs x_{17} and y_{17} are 28-bit and again we take the modulus $p = 2^{521} - 1$. In this case, the TMVP is given in Fig. 4.1. Since 18 is a multiple of 2 and 3, both (2.1) and (2.2) are applicable but there are challenges in the form of overflow and efficiency. As pointed out by Scott in [33], to ensure numerical stability on 32-bit computers the reduced-radix representation is useful when the limbs size are at most 29-bit. Unfortunately, in our case the size of the entries of the Toeplitz matrix becomes 30-bit because of the constant 2 as a result of the reduction as shown in Fig. 4.1. Moreover, the large number of limbs makes things challenging and perplexing due to overflow. Therefore, one must avoid overflow not only in single-word (32-bit) but also in the double-word (64-bit) otherwise, the result is not correct.

Since (2.1) and (2.2) are both applicable to TMVP in Fig. 4.1, we find three techniques with different arithmetic costs. It's not just the arithmetic cost that we are interested in but also the timing result. Therefore, we discuss the three techniques in detail by providing their arithmetic costs and timing results. We select the one with minimum clock cycles count and show that the technique does not cause overflow using the worst-case analysis. Moreover, the chosen technique is used in the implementation of the scalar multiplication on 32-bit platforms.

4.2 Technique-1

Our most efficient technique is based on the idea of applying (2.2) to TMVP in Fig. 4.1 followed by (2.1) where possible. So, at level-1 we apply (2.2) which results in

$$\begin{bmatrix}
x_0 & 2x_{17} & 2x_{16} & 2x_{15} & 2x_{14} & 2x_{13} & 2x_{12} & 2x_{11} & 2x_{10} & 2x_9 & 2x_8 & 2x_7 & 2x_6 & 2x_5 & 2x_4 & 2x_3 & 2x_2 & 2x_1 \\
x_1 & x_0 & 2x_{17} & 2x_{16} & 2x_{15} & 2x_{14} & 2x_{13} & 2x_{12} & 2x_{11} & 2x_{10} & 2x_9 & 2x_8 & 2x_7 & 2x_6 & 2x_5 & 2x_4 & 2x_3 & 2x_2 \\
x_2 & x_1 & x_0 & 2x_{17} & 2x_{16} & 2x_{15} & 2x_{14} & 2x_{13} & 2x_{12} & 2x_{11} & 2x_{10} & 2x_9 & 2x_8 & 2x_7 & 2x_6 & 2x_5 & 2x_4 & 2x_3 \\
x_3 & x_2 & x_1 & x_0 & 2x_{17} & 2x_{16} & 2x_{15} & 2x_{14} & 2x_{13} & 2x_{12} & 2x_{11} & 2x_{10} & 2x_9 & 2x_8 & 2x_7 & 2x_6 & 2x_5 & 2x_4 \\
x_4 & x_3 & x_2 & x_1 & x_0 & 2x_{17} & 2x_{16} & 2x_{15} & 2x_{14} & 2x_{13} & 2x_{12} & 2x_{11} & 2x_{10} & 2x_9 & 2x_8 & 2x_7 & 2x_6 & 2x_5 \\
x_5 & x_4 & x_3 & x_2 & x_1 & x_0 & 2x_{17} & 2x_{16} & 2x_{15} & 2x_{14} & 2x_{13} & 2x_{12} & 2x_{11} & 2x_{10} & 2x_9 & 2x_8 & 2x_7 & 2x_6 \\
x_6 & x_5 & x_4 & x_3 & x_2 & x_1 & x_0 & 2x_{17} & 2x_{16} & 2x_{15} & 2x_{14} & 2x_{13} & 2x_{12} & 2x_{11} & 2x_{10} & 2x_9 & 2x_8 & 2x_7 \\
x_7 & x_6 & x_5 & x_4 & x_3 & x_2 & x_1 & x_0 & 2x_{17} & 2x_{16} & 2x_{15} & 2x_{14} & 2x_{13} & 2x_{12} & 2x_{11} & 2x_{10} & 2x_9 & 2x_8 \\
x_8 & x_7 & x_6 & x_5 & x_4 & x_3 & x_2 & x_1 & x_0 & 2x_{17} & 2x_{16} & 2x_{15} & 2x_{14} & 2x_{13} & 2x_{12} & 2x_{11} & 2x_{10} & 2x_9 \\
x_9 & x_8 & x_7 & x_6 & x_5 & x_4 & x_3 & x_2 & x_1 & x_0 & 2x_{17} & 2x_{16} & 2x_{15} & 2x_{14} & 2x_{13} & 2x_{12} & 2x_{11} & 2x_{10} \\
x_{10} & x_9 & x_8 & x_7 & x_6 & x_5 & x_4 & x_3 & x_2 & x_1 & x_0 & 2x_{17} & 2x_{16} & 2x_{15} & 2x_{14} & 2x_{13} & 2x_{12} & 2x_{11} \\
x_{11} & x_{10} & x_9 & x_8 & x_7 & x_6 & x_5 & x_4 & x_3 & x_2 & x_1 & x_0 & 2x_{17} & 2x_{16} & 2x_{15} & 2x_{14} & 2x_{13} & 2x_{12} \\
x_{12} & x_{11} & x_{10} & x_9 & x_8 & x_7 & x_6 & x_5 & x_4 & x_3 & x_2 & x_1 & x_0 & 2x_{17} & 2x_{16} & 2x_{15} & 2x_{14} & 2x_{13} \\
x_{13} & x_{12} & x_{11} & x_{10} & x_9 & x_8 & x_7 & x_6 & x_5 & x_4 & x_3 & x_2 & x_1 & x_0 & 2x_{17} & 2x_{16} & 2x_{15} & 2x_{14} \\
x_{14} & x_{13} & x_{12} & x_{11} & x_{10} & x_9 & x_8 & x_7 & x_6 & x_5 & x_4 & x_3 & x_2 & x_1 & x_0 & 2x_{17} & 2x_{16} & 2x_{15} \\
x_{15} & x_{14} & x_{13} & x_{12} & x_{11} & x_{10} & x_9 & x_8 & x_7 & x_6 & x_5 & x_4 & x_3 & x_2 & x_1 & x_0 & 2x_{17} & 2x_{16} \\
x_{16} & x_{15} & x_{14} & x_{13} & x_{12} & x_{11} & x_{10} & x_9 & x_8 & x_7 & x_6 & x_5 & x_4 & x_3 & x_2 & x_1 & x_0 & 2x_{17} \\
x_{17} & x_{16} & x_{15} & x_{14} & x_{13} & x_{12} & x_{11} & x_{10} & x_9 & x_8 & x_7 & x_6 & x_5 & x_4 & x_3 & x_2 & x_1 & x_0
\end{bmatrix} \times \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \\ y_8 \\ y_9 \\ y_{10} \\ y_{11} \\ y_{12} \\ y_{13} \\ y_{14} \\ y_{15} \\ y_{16} \\ y_{17} \end{bmatrix}$$

Figure 4.1: TMVP of size 18 for 32-bit implementation

six TMVP of size 6, called P_i for $i = 1, \ldots, 6$, such that half of them are eligible for the application of (2.1) at level-2. Because application of (2.1) requires at least one-bit space and the matrix entries of P_2, P_3 and P_4 are 30-, 31- and 31-bit signed integer at the worst-case, respectively. The decision to use either the schoolbook or a TMVP variant for size 6 is driven by our timing results. So, the cross-over point between the schoolbook and TMVP variants of size 6 is discussed in detail in Section 4.5.1. Hence, we have designed our residue multiplication based on the arithmetic and timing results. The technique is discussed in detail in the next section.

4.2.1 Level-1

By using (2.2) the Fig. 4.1 can be represented as follows:

$$\left[\begin{array}{c|c|c} X_0 & 2X_2 & 2X_1 \\ \hline X_1 & X_0 & 2X_2 \\ \hline X_2 & X_1 & X_0 \end{array}\right] \times \left[\begin{array}{c} Y_0 \\ \hline Y_1 \\ \hline Y_2 \end{array}\right] = \begin{bmatrix} P_3 + P_4 + P_6 \\ P_2 - P_4 + P_5 \\ P_1 - P_2 - P_3 \end{bmatrix}$$

$$\text{where} \quad P_1 = (X_2 + X_1 + X_0)Y_0, \quad P_2 = X_1(Y_0 - Y_1),$$
$$P_3 = X_0(Y_0 - Y_2), \quad P_4 = 2X_2(Y_1 - Y_2),$$
$$P_5 = (X_0 + X_1 + 2X_2)Y_1, \quad P_6 = (2(X_2 + X_1) + X_0)Y_2.$$

The sub-matrices X_i for $i \in \{0, 1, 2\}$ are of size 6×6 and contain (some) common elements, whereas, the vectors Y_i are of size 6×1. Again, we represent a Toeplitz matrix by first row and first column.

Computing P_2:

$$Y_0 - Y_1 = \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \end{bmatrix} - \begin{bmatrix} y_6 \\ y_7 \\ y_8 \\ y_9 \\ y_{10} \\ y_{11} \end{bmatrix} = \underbrace{\begin{bmatrix} U_0 \\ U_1 \\ U_2 \\ U_3 \\ U_4 \\ U_5 \end{bmatrix}}_{\texttt{30-bit signed}}$$

The size of the matrix entries of P_2 is 29-bit unsigned but we take it as 30-bit signed integer. Still, P_2 is a good candidate for level-2 computation because of the two-bit free space. So, neither the matrix nor the vector entries will cause overflow in single- and double-word. Hence, the arithmetic cost of this vector computation is $6\mathbf{A}$.

Computing P_4:

$$Y_1 - Y_2 = \begin{bmatrix} y_6 \\ y_7 \\ y_8 \\ y_9 \\ y_{10} \\ y_{11} \end{bmatrix} - \begin{bmatrix} y_{12} \\ y_{13} \\ y_{14} \\ y_{15} \\ y_{16} \\ y_{17} \end{bmatrix} = \underbrace{\begin{bmatrix} U_6 \\ U_7 \\ U_8 \\ U_9 \\ U_{10} \\ U_{11} \end{bmatrix}}_{\texttt{30-bit signed}}$$

We have to also compute $2X_2 = \{2x_7, 2x_8, \ldots, 2x_{17}\}$ and as a result we take the size of the matrix entries of P_4 as 31-bit signed integer at the worst-case. But still we have one-bit free space, therefore, P_4 is eligible for level-2 computation without causing overflow. Hence, the arithmetic cost of these operations is $6\mathbf{A} + 11$**-shift**.

Computing P_3:

$$Y_0 - Y_2 = \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \end{bmatrix} - \begin{bmatrix} y_{12} \\ y_{13} \\ y_{14} \\ y_{15} \\ y_{16} \\ y_{17} \end{bmatrix} = \underbrace{\begin{bmatrix} U_{12} \\ U_{13} \\ U_{14} \\ U_{15} \\ U_{16} \\ U_{17} \end{bmatrix}}_{\texttt{30-bit signed}}$$

The lower triangular part of the matrix of P_3 consists of 29-bit entries while the rest of the entries are 30-bit unsigned integer. At the worst-case, we consider the matrix entries as 31-bit signed integer. Like P_4, we have one-bit free space and a candidate for level-2 computation. Note that some of the matrix entries of P_3 are found in $2X_2$. Instead of re-computation, we reuse the computed values. Hence, the arithmetic cost of this vector computation is $6\mathbf{A}$.

Computing P_1:

$$C_1 = X_1 + X_2 = \begin{bmatrix} x_6 & x_5 & x_4 & x_3 & x_2 & x_1 \\ x_7 & & & & & \\ x_8 & & & & & \\ x_9 & & & & & \\ x_{10} & & & & & \\ x_{11} & & & & & \end{bmatrix} + \begin{bmatrix} x_{12} & x_{11} & x_{10} & x_9 & x_8 & x_7 \\ x_{13} & & & & & \\ x_{14} & & & & & \\ x_{15} & & & & & \\ x_{16} & & & & & \\ x_{17} & & & & & \end{bmatrix}$$

$$= \underbrace{\begin{bmatrix} S_1 & S_2 & S_3 & S_4 & S_5 & S_6 \\ S_7 & & & & & \\ S_8 & & & & & \\ S_9 & & & & & \\ S_{10} & & & & & \\ S_{11} & & & & & \end{bmatrix}}_{\texttt{30-bit unsigned}}$$

$$C_2 = C_1 + X_0 = \begin{bmatrix} S_1 & S_2 & S_3 & S_4 & S_5 & S_6 \\ S_7 & & & & & \\ S_8 & & & & & \\ S_9 & & & & & \\ S_{10} & & & & & \\ S_{11} & & & & & \end{bmatrix} + \begin{bmatrix} x_0 & 2x_{17} & 2x_{16} & 2x_{15} & 2x_{14} & 2x_{13} \\ x_1 & & & & & \\ x_2 & & & & & \\ x_3 & & & & & \\ x_4 & & & & & \\ x_5 & & & & & \end{bmatrix}$$

$$= \underbrace{\begin{bmatrix} S_{12} & S_{13} & S_{14} & S_{15} & S_{16} & S_{17} \\ S_{18} & & & & & \\ S_{19} & & & & & \\ S_{20} & & & & & \\ S_{21} & & & & & \\ S_{22} & & & & & \end{bmatrix}}_{\texttt{31-bit unsigned}}$$

$$C_2 \times Y_0 = \begin{bmatrix} S_{12} & S_{13} & S_{14} & S_{15} & S_{16} & S_{17} \\ S_{18} & & & & & \\ S_{19} & & & & & \\ S_{20} & & & & & \\ S_{21} & & & & & \\ S_{22} & & & & & \end{bmatrix} \times \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \end{bmatrix}$$

At the worst-case, the size of the matrix entries is 31-bit unsigned or 32-bit signed integer. We exploit P_1 for level-2 computation which results in multiplication of signed by unsigned integer. Unfortunately, in our pure C-implementation we find mixed-sign multiplication inefficient. We also used the strategies of type casting to avoid mixed-sign multiplication but still the timing results are not good. Although the application of (2.1) to compute P_1 at level-2 is efficient in terms of arithmetic cost, our timing results are not good. Hence, we compute P_1 using the schoolbook at this level and the total arithmetic cost will be $36\mathbf{M} + 22\mathbf{A} + 30\mathbf{A}_d$.

Computing P_6: We have

$$2(X_2 + X_1) + X_0 = (X_2 + X_1 + X_0) + (X_2 + X_1) = C_2 + C_1$$

then

$$C_2 + C_1 = \begin{bmatrix} S_{12} & S_{13} & S_{14} & S_{15} & S_{16} & S_{17} \\ S_{18} & & & & & \\ S_{19} & & & & & \\ S_{20} & & & & & \\ S_{21} & & & & & \\ S_{22} & & & & & \end{bmatrix} + \begin{bmatrix} S_1 & S_2 & S_3 & S_4 & S_5 & S_6 \\ S_7 & & & & & \\ S_8 & & & & & \\ S_9 & & & & & \\ S_{10} & & & & & \\ S_{11} & & & & & \end{bmatrix}$$

$$= \underbrace{\begin{bmatrix} S_{23} & S_{24} & S_{25} & S_{26} & S_{27} & S_{28} \\ S_{29} & & & & & \\ S_{30} & & & & & \\ S_{31} & & & & & \\ S_{32} & & & & & \\ S_{33} & & & & & \end{bmatrix}}_{\texttt{32-bit unsigned}}$$

$$(C_2 + C_1) \times Y_2 = \begin{bmatrix} S_{23} & S_{24} & S_{25} & S_{26} & S_{27} & S_{28} \\ S_{29} & & & & & \\ S_{30} & & & & & \\ S_{31} & & & & & \\ S_{32} & & & & & \\ S_{33} & & & & & \end{bmatrix} \times \begin{bmatrix} y_{12} \\ y_{13} \\ y_{14} \\ y_{15} \\ y_{16} \\ y_{17} \end{bmatrix}$$

The size of the matrix entries will be 32-bit unsigned at the worst-case. Therefore, we compute P_6 using the schoolbook at this level. Hence, the total arithmetic cost of computing P_6 is $36\mathbf{M} + 11\mathbf{A} + 30\mathbf{A}_d$.

Computing P_5:

$$X_0 + X_1 = \begin{bmatrix} x_0 & 2x_{17} & 2x_{16} & 2x_{15} & 2x_{14} & 2x_{13} \\ x_1 & & & & & \\ x_2 & & & & & \\ x_3 & & & & & \\ x_4 & & & & & \\ x_5 & & & & & \end{bmatrix} + \begin{bmatrix} x_6 & x_5 & x_4 & x_3 & x_2 & x_1 \\ x_7 & & & & & \\ x_8 & & & & & \\ x_9 & & & & & \\ x_{10} & & & & & \\ x_{11} & & & & & \end{bmatrix}$$

$$= \begin{bmatrix} x_0 + x_6 & 2x_{17} + x_5 & 2x_{16} + x_4 & 2x_{15} + x_3 & 2x_{14} + x_2 & 2x_{13} + x_1 \\ x_1 + x_7 & & & & & \\ x_2 + x_8 & & & & & \\ x_3 + x_9 & & & & & \\ x_4 + x_{10} & & & & & \\ x_5 + x_{11} & & & & & \end{bmatrix}$$

$$(X_0 + X_1) + 2X_2 = \begin{bmatrix} S_{12} + x_{12} & S_{33} & S_{32} & S_{31} & S_{30} & S_{29} \\ S_{17} & & & & & \\ S_{16} & & & & & \\ S_{15} & & & & & \\ S_{14} & & & & & \\ S_{13} & & & & & \end{bmatrix}$$

$$((X_0 + X_1) + 2X_2) \times Y_1 = \begin{bmatrix} S_{12} + x_{12} & S_{33} & S_{32} & S_{31} & S_{30} & S_{29} \\ S_{17} & & & & & \\ S_{16} & & & & & \\ S_{15} & & & & & \\ S_{14} & & & & & \\ S_{13} & & & & & \end{bmatrix} \times \begin{bmatrix} y_6 \\ y_7 \\ y_8 \\ y_9 \\ y_{10} \\ y_{11} \end{bmatrix}$$

From the computation of P_1 and P_6, we already know that the matrix entries S_{13} to S_{17} and S_{29} to S_{33} are 31- and 32-bit unsigned, respectively. At the worst-case, the entry $S_{12} + x_{12}$ is 32-bit unsigned. So, we P_5 using the schoolbook at this level. Hence, the total arithmetic cost of computing P_5 is $36\mathbf{M} + 1\mathbf{A} + 30\mathbf{A}_d$.

Final Computation: Note that P_2, P_3 and P_4 are not fully computed yet and once all the P_i for $i = 1, \ldots, 6$ are computed then we have to compute

$$\begin{bmatrix} P_3 + P_4 + P_6 \\ P_2 - P_4 + P_5 \\ P_1 - P_2 - P_3 \end{bmatrix}$$

where each P_i is a double-word vector of size 6. Hence, the arithmetic cost of this final computation is $36\mathbf{A}_d$.

4.2.2 Level-2

There are three calls from level-1 to compute TMVP of size 6 at this level. From the arithmetic cost and (our) timing results, it is evident that (2.1) is efficient than the schoolbook by using its proper variant. The further details are given in Section 4.5.1. The application of (2.1) results in three TMVP of size 3. By the worst-case bitlength analysis, P_4 is an ideal candidate to take into account. So, we have

$$\underbrace{\begin{bmatrix} 2x_{12} & 2x_{11} & 2x_{10} & 2x_9 & 2x_8 & 2x_7 \\ 2x_{13} & & & & & \\ 2x_{14} & & & & & \\ 2x_{15} & & & & & \\ 2x_{16} & & & & & \\ 2x_{17} & & & & & \end{bmatrix}}_{\texttt{31-bit signed}} \times \underbrace{\begin{bmatrix} U_6 \\ U_7 \\ U_8 \\ U_9 \\ U_{10} \\ U_{11} \end{bmatrix}}_{\texttt{30-bit signed}}$$

$$= \begin{bmatrix} X_0 & X_1 \\ X_2 & X_0 \end{bmatrix} \times \begin{bmatrix} Y_0 \\ Y_1 \end{bmatrix} = \begin{bmatrix} P_{4,2} + P_{4,1} \\ P_{4,3} - P_{4,1} \end{bmatrix}$$

$$\text{where} \quad P_{4,1} = X_0(Y_0 - Y_1), \qquad P_{4,2} = (X_1 + X_0)Y_1,$$
$$P_{4,3} = (X_2 + X_0)Y_0$$

Computing $P_{4,1}$:

$$\begin{bmatrix} U_6 \\ U_7 \\ U_8 \end{bmatrix} - \begin{bmatrix} U_9 \\ U_{10} \\ U_{11} \end{bmatrix} = \underbrace{\begin{bmatrix} U_{19} \\ U_{20} \\ U_{21} \end{bmatrix}}_{\texttt{31-bit signed}}$$

$$\begin{bmatrix} 2x_{12} & 2x_{11} & 2x_{10} \\ 2x_{13} & & \\ 2x_{14} & & \end{bmatrix} \times \begin{bmatrix} U_{19} \\ U_{20} \\ U_{21} \end{bmatrix}$$

Hence, the total arithmetic cost of $P_{4,1}$ is $9\mathbf{M} + 3\mathbf{A} + 6\mathbf{A}_d$.

Computing $P_{4,2}$:

$$\begin{bmatrix} 2x_9 & 2x_8 & 2x_7 \\ 2x_{10} & & \\ 2x_{11} & & \end{bmatrix} + \begin{bmatrix} 2x_{12} & 2x_{11} & 2x_{10} \\ 2x_{13} & & \\ 2x_{14} & & \end{bmatrix} = \underbrace{\begin{bmatrix} T_1 & T_2 & T_3 \\ T_4 & & \\ T_5 & & \end{bmatrix}}_{\texttt{32-bit signed}}$$

$$\begin{bmatrix} T_1 & T_2 & T_3 \\ T_4 & & \\ T_5 & & \end{bmatrix} \times \begin{bmatrix} U_9 \\ U_{10} \\ U_{11} \end{bmatrix}$$

Hence, the total arithmetic cost of $P_{4,2}$ is $9\mathbf{M} + 5\mathbf{A} + 6\mathbf{A}_d$.

Computing $P_{4,3}$:

$$\begin{bmatrix} 2x_{15} & 2x_{14} & 2x_{13} \\ 2x_{16} & & \\ 2x_{17} & & \end{bmatrix} + \begin{bmatrix} 2x_{12} & 2x_{11} & 2x_{10} \\ 2x_{13} & & \\ 2x_{14} & & \end{bmatrix} = \underbrace{\begin{bmatrix} T_6 & T_5 & T_4 \\ T_7 & & \\ T_8 & & \end{bmatrix}}_{\texttt{32-bit signed}}$$

$$\begin{bmatrix} T_6 & T_5 & T_4 \\ T_7 & & \\ T_8 & & \end{bmatrix} \times \begin{bmatrix} U_6 \\ U_7 \\ U_8 \end{bmatrix}$$

Note that the entries T_4 and T_5 are already computed by $P_{4,2}$. Hence, the total arithmetic cost of $P_{4,3}$ is $9\mathbf{M} + 3\mathbf{A} + 6\mathbf{A}_d$.

Final Computation: Finally, we have to compute $P_{4,2} + P_{4,1}$ and $P_{4,3} - P_{4,1}$ where each $P_{4,i}$ is a vector of size 3 for $i = 1, 2, 3$. Hence, the arithmetic cost of this final computation is $6\mathbf{A}_d$.

4.2.3 Overall Cost

At level-1, the arithmetic costs are: P_2 and P_3 are each $6\mathbf{A}$, P_4 is $6\mathbf{A} + 11$**-shift**, P_1 is $36\mathbf{M} + 22\mathbf{A} + 30\mathbf{A}_d$, P_6 is $36\mathbf{M} + 11\mathbf{A} + 30\mathbf{A}_d$, P_5 is $36\mathbf{M} + 1\mathbf{A} + 30\mathbf{A}_d$ and the final computation step is $36\mathbf{A}_d$. So, the total arithmetic cost of level-1 is $108\mathbf{M}+52\mathbf{A}+126\mathbf{A}_d+11$**-shift**. There are three calls to level-2 so, the total arithmetic cost of level-2 is $3\big(27\mathbf{M}+11\mathbf{A}+24\mathbf{A}_d\big) = 81\mathbf{M}+33\mathbf{A}+72\mathbf{A}_d$. Hence, the overall cost

of our residue multiplication for 32-bit platforms is $189\mathbf{M}+85\mathbf{A}+198\mathbf{A}_d+11$**-shift** $= 189\mathbf{M} + 481\mathbf{A} + 11$**-shift**. Note that we are taking $1\mathbf{A}_d = 2\mathbf{A}$.

4.3 Technique-2 and Technique-3

The two techniques that we discuss in this section are based on the idea of applying (2.1) to the TMVP in Fig. 4.1 followed by (2.2) and some tricks where (2.2) is not directly applicable. If we apply (2.1) straightaway, then it is not a good approach at all. Because the application of (2.1) at level-1 results in three TMVP of size 9 that are P_1, P_2 and P_3. Clearly, the matrix entries of P_1, P_2 and P_3 differ in sizes but for the sake of simplicity and the worst-case, we take the largest size. So, the matrix entries of P_1, P_2 and P_3 will be at most 30-, 31- and 31-bit unsigned/non-negative integer, respectively. It is evident from [1, 23, 33] that in a pure C-implementation — no intrinsics and SIMD/assembly instructions — the schoolbook is not efficient for matrix-vector product of size 9 or polynomials product of degree 8. So, if we apply (2.2) at level-2, then only P_1 is eligible because there are two-bit space in 32-bit word while P_2 and P_3 do not have enough free space to avoid overflow. We didn't find it efficient to exploit the fact that the matrix entries of P_1, P_2 and P_3 are not the same bitlength. Does it mean a dead end? No, there is a way out and the better approach is to exploit the structure of Fig. 4.1 by applying (2.1) as follows.

$$\left[\begin{array}{c:c} X_0 & 2X_1 \\ \hdashline X_1 & X_0 \end{array}\right] \times \left[\begin{array}{c} Y_0 \\ \hdashline Y_1 \end{array}\right] = \left[\begin{array}{c} P_1 + P_3 \\ P_1 + P_2 \end{array}\right]$$

$$\text{where} \qquad P_1 = (X_0 + X_1)Y_0, \qquad P_2 = X_0(Y_1 - Y_0)$$
$$P_3 = X_1(Y_1 + (Y_1 - Y_0)).$$

The (Toeplitz) sub-matrices X_i for $i = 0, 1$ are of size 9×9 and contain (some) common elements, whereas, the vectors Y_i are of size 9×1. The application of (2.2) at level-2 seems to be smooth for the computation of P_2 and P_3 because there is no danger of overflow. So, the arithmetic cost of P_2 and P_3 at level-1 is each $9\mathbf{A}$. On the other hand, the computation of P_1 is tricky because the size of the (Toeplitz) matrix entries of $X_0 + X_1$ is 30- and 31-bit unsigned. So, our two techniques differ in how

we compute P_1 and suppose at level-1 the P_1 is

$$P_1 = \begin{bmatrix} S_1 & S_2 & S_3 & S_4 & S_5 & S_6 & S_7 & S_8 & S_9 \\ S_{10} & & & & & & & & \\ S_{11} & & & & & & & & \\ S_{12} & & & & & & & & \\ S_{13} & & & & & & & & \\ S_{14} & & & & & & & & \\ S_{15} & & & & & & & & \\ S_{16} & & & & & & & & \\ S_{17} & & & & & & & & \end{bmatrix} \times \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \\ y_8 \end{bmatrix}$$

where
$$\begin{aligned} S_1 &= x_0 + x_9, & S_2 &= 2x_{17} + x_8, & S_3 &= 2x_{16} + x_7, \\ S_4 &= 2x_{15} + x_6, & S_5 &= 2x_{14} + x_5, & S_6 &= 2x_{13} + x_4, \\ S_7 &= 2x_{12} + x_3, & S_8 &= 2x_{11} + x_2, & S_9 &= 2x_{10} + x_1, \\ S_{10} &= x_1 + x_{10}, & S_{11} &= x_2 + x_{11}, & S_{12} &= x_3 + x_{12}, \\ S_{13} &= x_4 + x_{13}, & S_{14} &= x_5 + x_{14}, & S_{15} &= x_6 + x_{15}, \\ S_{16} &= x_7 + x_{16}, & S_{17} &= x_8 + x_{17}. \end{aligned}$$

The total cost of computing S_1 to S_{17} is $17\mathbf{A} + 8$**-shift**. The size of the matrix entries S_1 and S_{10} to S_{17} are 30-bit unsigned while the size of S_2 to S_9 is 31-bit unsigned at the worst-case. In our residue representation x_0 is at most 30-bit unsigned which implies that we take the size of S_1 to S_9 as 31-bit unsigned. Now, it's time to discuss the two techniques of computing P_1.

4.3.1 Technique-2

Let's compute P_1 using (2.2) at level-2 and handle the overflow where it arises. So, we have

$$\begin{bmatrix} X_0 & X_1 & X_2 \\ X_3 & X_0 & X_1 \\ X_4 & X_3 & X_0 \end{bmatrix} \times \begin{bmatrix} Y_0 \\ Y_1 \\ Y_2 \end{bmatrix} = \begin{bmatrix} P_{1,3} + P_{1,4} + P_{1,6} \\ P_{1,2} - P_{1,4} + P_{1,5} \\ P_{1,1} - P_{1,2} - P_{1,3} \end{bmatrix}$$

$$\text{where} \quad P_{1,1} = (X_4 + X_3 + X_0)Y_0, \quad P_{1,2} = X_3(Y_0 - Y_1),$$
$$P_{1,3} = X_0(Y_0 - Y_2), \quad P_{1,4} = X_1(Y_1 - Y_2),$$
$$P_{1,5} = (X_0 + X_3 + X_1)Y_1, \quad P_{1,6} = (X_2 + X_0 + X_1)Y_2.$$

The sub-matrices X_i for $i \in \{0, 1, 2\}$ are of size 3×3 and contain (some) common elements, whereas, the vectors Y_i are of size 3×1. Like our 64-bit residue multiplication we use the schoolbook for TMVP of size 3 which implies that the total arithmetic cost of $P_{1,2}, P_{1,3}$ and $P_{1,4}$ is $3\big(9\mathbf{M} + 3\mathbf{A} + 6\mathbf{A}_d\big)$.

Computing $P_{1,1}$:

$$X_3 + X_0 = \begin{bmatrix} S_{12} & S_{11} & S_{10} \\ S_{13} & & \\ S_{14} & & \end{bmatrix} + \begin{bmatrix} S_1 & S_2 & S_3 \\ S_{10} & & \\ S_{11} & & \end{bmatrix} = \begin{bmatrix} S_{18} & S_{19} & S_{20} \\ S_{21} & & \\ S_{22} & & \end{bmatrix}$$

$$X_4 + (X_3 + X_0) = \begin{bmatrix} S_{15} & S_{14} & S_{13} \\ S_{16} & & \\ S_{17} & & \end{bmatrix} + \begin{bmatrix} S_{18} & S_{19} & S_{20} \\ S_{21} & & \\ S_{22} & & \end{bmatrix} = \begin{bmatrix} S_{23} & S_{24} & S_{25} \\ S_{26} & & \\ S_{27} & & \end{bmatrix}$$

$$(X_2 + X_1 + X_0)Y_0 = \underbrace{\begin{bmatrix} S_{23} & S_{24} & S_{25} \\ S_{26} & & \\ S_{27} & & \end{bmatrix}}_{\texttt{32-bit unsigned}} \times \begin{bmatrix} y_0 \\ y_1 \\ y_2 \end{bmatrix}$$

The value of S_{23}, S_{24} and S_{25} can be at most $7(2^{29} - 1)$ and the value of S_{26} and S_{27} can be at most $6(2^{29} - 1)$. In other words, the size of the matrix entries is at most 32-bit unsigned. Hence, the total arithmetic cost of $P_{1,1}$ is $9\mathbf{M} + 10\mathbf{A} + 6\mathbf{A}_d$.

Computing $P_{1,5}$:

$$(X_3 + X_0) + X_1 = \begin{bmatrix} S_{18} & S_{19} & S_{20} \\ S_{21} & & \\ S_{22} & & \end{bmatrix} + \begin{bmatrix} S_4 & S_5 & S_6 \\ S_3 & & \\ S_2 & & \end{bmatrix} = \begin{bmatrix} S_{28} & S_{29} & S_{30} \\ S_{31} & & \\ S_{32} & & \end{bmatrix}$$

$$(X_3 + X_0 + X_1)Y_1 = \underbrace{\begin{bmatrix} S_{28} & S_{29} & S_{30} \\ S_{31} & & \\ S_{32} & & \end{bmatrix}}_{\texttt{32-bit unsigned}} \times \begin{bmatrix} y_3 \\ y_4 \\ y_5 \end{bmatrix}$$

The value of S_{28}, S_{29} and S_{30} can be at most $8(2^{29} - 1)$ and the value of S_{31} and S_{32}

can be at most $7(2^{29}-1)$. In other words, the size of the matrix entries is at most 32-bit unsigned. Hence, the total arithmetic cost of $P_{1,5}$ is $9\mathbf{M}+5\mathbf{A}+6\mathbf{A}_d$.

Computing $P_{1,6}$: The upper triangular matrix entries of the result of $X_0+X_1+X_2$ causes overflow in 32-bit word. Therefore, to avoid the overflow we compute $P_{1,6}$ as follows

$$= \begin{bmatrix} (S_1+S_4)y_6+S_7y_6 & +(S_2+S_5)y_7+S_8y_7 & +(S_3+S_6)y_8+S_9y_8 \\ S_{30}y_6 & +(S_1+S_4)y_7+S_7y_7 & +(S_2+S_5)y_8+S_8y_8 \\ S_{29}y_6 & +S_{30}y_7 & +(S_1+S_4)y_8+S_7y_8 \end{bmatrix}$$

Hence, the total arithmetic cost of $P_{1,6}$ is $15\mathbf{M}+3\mathbf{A}+12\mathbf{A}_d$. Note that we are not recomputing the common expression just like before.

Final Computation: At last, we have to compute

$$\begin{bmatrix} P_{1,3}+P_{1,4}+P_{1,6} \\ P_{1,2}-P_{1,4}+P_{1,5} \\ P_{1,1}-P_{1,2}-P_{1,3} \end{bmatrix}$$

where each $P_{1,i}$ for $i=1,\ldots,6$ is a 3×1 vector and the elements are of double-word size so the total cost is $18\mathbf{A}_d$. Hence, the total arithmetic cost of P_1 at level-2 is $3\big(9\mathbf{M}+3\mathbf{A}+6\mathbf{A}_d\big)+9\mathbf{M}+10\mathbf{A}+6\mathbf{A}_d+9\mathbf{M}+5\mathbf{A}+6\mathbf{A}_d+15\mathbf{M}+3\mathbf{A}+12\mathbf{A}_d+18\mathbf{A}_d = 60\mathbf{M}+27\mathbf{A}+60\mathbf{A}_d$.

4.3.1.1 Overall Cost

From the computation of P_1 using (2.2) at level-2, it is easy to conceive that the total arithmetic cost of P_2 and P_3 at level-2 is $2\big(54\mathbf{M}+30\mathbf{A}+54\mathbf{A}_d\big) = 108\mathbf{M}+60\mathbf{A}+108\mathbf{A}_d$. Then the total arithmetic cost of the computation at level-2 and level-1 is $168\mathbf{M}+87\mathbf{A}+168\mathbf{A}_d$ and $35\mathbf{A}+18\mathbf{A}_d$, respectively, precluding the negligible cost of **shift** operation. Thus, the overall cost is $168\mathbf{M}+122\mathbf{A}+186\mathbf{A}_d = 168\mathbf{M}+494\mathbf{A}$.

4.3.2 Technique-3

From Technique-2, it is evident that the use of (2.2) to compute P_1 results in many (inefficient) mixed-sign multiplications e.g. the computation of $P_{1,1}, P_{1,5}$ and $P_{1,6}$.

Therefore, our technique-2 is based on the idea of reducing the number of mixed-sign multiplications and have minimum increase in the arithmetic cost. Definitely, the number of multiplications will be more than the technique-1. From Section 4.5.1, it is clear that we have size 6 as the cross-over point between the schoolbook and a TMVP variant. Therefore, we compute P_1 by decomposing it, shown by dotted lines, in such a way that we have a TMVP of size 6 as shown below.

$$P_1 = \left[\begin{array}{cccccc:ccc} S_1 & S_2 & S_3 & S_4 & S_5 & S_6 & S_7 & S_8 & S_9 \\ S_{10} & S_1 & S_2 & S_3 & S_4 & S_5 & S_6 & S_7 & S_8 \\ S_{11} & S_{10} & S_1 & S_2 & S_3 & S_4 & S_5 & S_6 & S_7 \\ \hdashline S_{12} & S_{11} & S_{10} & S_1 & S_2 & S_3 & S_4 & S_5 & S_6 \\ S_{13} & & & & & & S_3 & S_4 & S_5 \\ S_{14} & & & & & & S_2 & S_3 & S_4 \\ S_{15} & & & & & & S_1 & S_2 & S_3 \\ S_{16} & & & & & & S_{10} & S_1 & S_2 \\ S_{17} & & & & & & S_{11} & S_{10} & S_1 \end{array}\right] \times \left[\begin{array}{c} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \\ y_8 \end{array}\right]$$

The first, second and third row of P_1 are (fully) computed by sum of products while the rest are computed as sum of the result of the TMVP of size 6 and sum of the products of the other entries in that respective row. So, the total arithmetic cost of the first, second and third row is $27\mathbf{M}+24\mathbf{A}_d$. On the other hand, the total arithmetic cost of the partial results of the fourth to ninth row is $18\mathbf{M} + 12\mathbf{A}_d$ precluding the cost of the TMVP of size 6. Finally, the TMVP of size 6 is computed using (2.1) followed by the schoolbook because it results in the cross-over point between the schoolbook and TMVP. For further details, see Section 4.5.1. Hence, the TMVP of size 6 is computed as follows.

$$P'_1 = \left[\begin{array}{ccc:ccc} S_{12} & S_{11} & S_{10} & S_1 & S_2 & S_3 \\ S_{13} & & & & & \\ S_{14} & & & & & \\ \hdashline S_{15} & & & & & \\ S_{16} & & & & & \\ S_{17} & & & & & \end{array}\right] \times \left[\begin{array}{c} y_0 \\ y_1 \\ y_2 \\ \hdashline y_3 \\ y_4 \\ y_5 \end{array}\right]$$

$$\left[\begin{array}{cc} X_0 & X_1 \\ X_2 & X_0 \end{array}\right] \times \left[\begin{array}{c} Y_0 \\ Y_1 \end{array}\right] = \left[\begin{array}{c} P'_{1,2} + P'_{1,1} \\ P'_{1,3} - P'_{1,1} \end{array}\right]$$

$$\text{where} \quad P'_{1,1} = X_0(Y_0 - Y_1), \qquad P'_{1,2} = (X_1 + X_0)Y_1,$$
$$P'_{1,3} = (X_2 + X_0)Y_0.$$

The (Toeplitz) sub-matrices X_i for $i \in \{0, 1, 2\}$ are of size 3×3 and contain (some) common elements, whereas, the vectors Y_i are of size 3×1. Note that, at the worst-case, the entries of X_0 and X_2 are 30-bit unsigned while the upper triangular matrix entries of X_1 are 31-bit unsigned.

Computing $P'_{1,1}$:

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \end{bmatrix} - \begin{bmatrix} y_3 \\ y_4 \\ y_5 \end{bmatrix} = \begin{bmatrix} U_0 \\ U_1 \\ U_2 \end{bmatrix}$$

$$\underbrace{\begin{bmatrix} S_{12} & S_{11} & S_{10} \\ S_{13} & & \\ S_{14} & & \end{bmatrix}}_{\texttt{31-bit signed}} \times \underbrace{\begin{bmatrix} U_0 \\ U_1 \\ U_2 \end{bmatrix}}_{\texttt{30-bit signed}}$$

Hence, the total arithmetic cost of $P'_{1,1}$ is $9\mathbf{M} + 3\mathbf{A} + 6\mathbf{A}_d$.

Computing $P'_{1,2}$:

$$\begin{bmatrix} S_1 & S_2 & S_3 \\ S_{10} & & \\ S_{11} & & \end{bmatrix} + \begin{bmatrix} S_{12} & S_{11} & S_{10} \\ S_{13} & & \\ S_{14} & & \end{bmatrix} = \begin{bmatrix} T_1 & T_2 & T_3 \\ T_4 & & \\ T_5 & & \end{bmatrix}$$

By the worst-case, T_1, T_2 and T_3 are 32-bit unsigned while T_4 and T_5 are 31-bit unsigned.

$$\begin{bmatrix} T_1 & T_2 & T_3 \\ T_4 & & \\ T_5 & & \end{bmatrix} \times \begin{bmatrix} y_3 \\ y_4 \\ y_5 \end{bmatrix}$$

Hence, the total arithmetic cost of $P'_{1,2}$ is $9\mathbf{M} + 5\mathbf{A} + 6\mathbf{A}_d$.

Computing $P'_{1,3}$:

$$\begin{bmatrix} S_{15} & S_{14} & S_{13} \\ S_{16} & & \\ S_{17} & & \end{bmatrix} + \begin{bmatrix} S_{12} & S_{11} & S_{10} \\ S_{13} & & \\ S_{14} & & \end{bmatrix} = \begin{bmatrix} T_6 & T_5 & T_4 \\ T_7 & & \\ T_8 & & \end{bmatrix}$$

$$\underbrace{\begin{bmatrix} T_6 & T_5 & T_4 \\ T_7 & & \\ T_8 & & \end{bmatrix}}_{\texttt{32-bit signed}} \times \begin{bmatrix} y_0 \\ y_1 \\ y_2 \end{bmatrix}$$

Note that the entries T_4 and T_5 are already computed by $P'_{1,2}$. Hence, the total arithmetic cost of $P_{4,3}$ is $9\mathbf{M} + 3\mathbf{A} + 6\mathbf{A}_d$.

Final Computation: Finally, we have to compute $P'_{1,2} + P'_{1,1}$ and $P'_{1,3} - P'_{1,1}$ where each $P'_{1,i}$ is a vector of size 3 for $i = 1, 2, 3$. So, the arithmetic cost of these operations is $6\mathbf{A}_d$. The result of P'_1 needs to be added to the partial result of the fourth to ninth row of (original) P_1 and the arithmetic cost of this operation is $6\mathbf{A}_d$.

Hence, the total arithmetic cost of computing P_1 using technique-3 is $27\mathbf{M} + 24\mathbf{A}_d + 18\mathbf{M} + 12\mathbf{A}_d + 9\mathbf{M} + 3\mathbf{A} + 6\mathbf{A}_d + 9\mathbf{M} + 5\mathbf{A} + 6\mathbf{A}_d + 9\mathbf{M} + 3\mathbf{A} + 6\mathbf{A}_d + 12\mathbf{A}_d = 72\mathbf{M} + 11\mathbf{A} + 66\mathbf{A}_d$.

4.3.2.1 Overall Cost

Like before, P_2 and P_3 are computed using (2.2) at level-2 and the total arithmetic cost is $2\big(54\mathbf{M} + 30\mathbf{A} + 54\mathbf{A}_d\big) = 108\mathbf{M} + 60\mathbf{A} + 108\mathbf{A}_d$. Then the total arithmetic cost of the computation at level-2 and level-1 is $180\mathbf{M}+71\mathbf{A}+174\mathbf{A}_d$ and $35\mathbf{A}+18\mathbf{A}_d$, respectively, precluding the negligible cost of **shift** operation. Thus, the overall cost is $180\mathbf{M} + 106\mathbf{A} + 192\mathbf{A}_d = 180\mathbf{M} + 490\mathbf{A}$.

4.4 Residue Representation

From the timing results of both the residue and scalar multiplication in [1], it can be deduced that modulus p is efficient than $2p$. Therefore, we use modulus p and the same strategy of residue representation as the one employed in [1, 23]. In case of 32-bit implementation, the bitlength of the limbs is half of that 64-bit implementation. Therefore, for 32-bit residue multiplication and squaring, the input and output range of the limbs are $[0, 2^{29}+2^7-2] \times [0, 2^{29}-1]^{16} \times [0, 2^{28}-1]$ where the least significant limb is in $[0, 2^{29} + 2^7 - 2]$ and the most significant limb is in $[0, 2^{28} - 1]$. Hence, the

size of the least significant limbs x_0 and y_0 are at most 30-bit.

In case of 64-bit residue multiplication, overflow can be disregarded in spite of using different residue representation e.g. [1, 23]. Because there are 9-limb and enough bit space in single- and double-word to avoid overflow comfortably. But the situation is more challenging for our 32-bit residue multiplication techniques. Instead of proving that overflow does not occur in single- and double-word using all our techniques, we show the proof for our most efficient technique, minimum clock cycles count, on our machine. It is not a restriction that proof should be provided for the most efficient technique but we are doing it for the sake of simplicity. Definitely, the most efficient technique is determined by timing test.

4.5 Comparisons, Implementation and Testing Environment

In this section, to the best of our knowledge, we compare our 32-bit residue multiplication techniques to some of the well-known algorithms mainly in terms of the arithmetic cost and also provide the published timing results. For our timing results, we have to verify whether a TMVP variant takes less number of clock cycles than the schoolbook to compute a Toeplitz matrix-vector product of size 6. In other words, can size 6 be the cross-over point between the schoolbook and TMVP variant? So, we have to describe our testing environment for timing results.

The implementation/timing results are obtained on Intel(R) Core i$5-6402$P CPU @ 2.80GHz with Turbo Boost disabled, Hyper-threading not supported and the machine is running on one core. All our programs are written in C programming language and as pointed out in [23], this gives the freedom of portability unlike the other platform dependent (assembly) implementation. As recommended by Paoloni in his white paper [32], we use *rdtsc* and *rdtscp* for clock cycles counts. For all the C programs in this paper we use GCC $5.4.0$ compiler on Ubuntu 16.04 LTS and the clock cycles of our residue multiplication algorithms are counted at optimization level 2 i.e. -O2. Note that the testing environment is different from our earlier work [1] for 64-bit residue multiplication i.e. Chapter-3.

Table 4.1: SB = Schoolbook, 3wD = Three-way Decomposition, 2wD = Two-way Decomposition and *m.line* = manually inlined. Clock Cycles counts for SB and the TMVP variants of size 6

SB	3**wD**		2**wD**	
	m.line **SB**	*m.line* 2**wD**	*m.line* **SB**	*m.line* 3**wD**
83	84	83	76	90

4.5.1 Schoolbook vs. TMVP variants of size 6

We perform timing tests to determine whether size 6 is the cross-over point between the schoolbook and TMVP. Also, 6 is a multiple of both 2 and 3, therefore, it is necessary to decide to apply either (2.1) or (2.2). To answer these questions we executed our C programs with the command *gcc -Wall -m32 -O2 program.c -o program.exe*. We have tested the programs on 100 random values such that for each value the function is called 10^7 times in order to get consistent cycle count. The optimization argument -O3 returns 0 cycle. The results of our timing tests are provided in Table 4.1.

By *m.line*, we mean that the callee function is manually inlined within the body of the caller function. Because we found the compiler's inlining feature slower than our manual inlining. Our timing results show that the application of (2.1) followed by the schoolbook is the most efficient approach such that the schoolbook is manually inlined within the caller function. On the other hand, the application of (2.1) followed by (2.2) is the worst technique. The application of the schoolbook results in almost the same cycle count as the application of (2.2) followed by either (2.1) or the schoolbook. Hence, we find size 6 as the cross-over point where a TMVP variant is more efficient than the straight application of the schoolbook.

4.5.2 Residue Multiplication and Squaring

In this section, we provide the arithmetic cost and clock cycles count of our residue multiplication techniques and squaring for 32- and 64-bit platforms. Since the testing environment has changed therefore, we also provide the new timing result of our 64-bit residue multipicaiton. In case of 64-bit, we don't reproduce the Table 1 in [1] and

only include the one that we require i.e. Hybrid version. Because in terms of the timing results, Hybrid version was found the most efficient in [1] and the source code is freely available on GitHub.

In the earlier sections of this chapter, you may have noticed that we presented the total arithmetic cost in terms of the single-word multiplication and addition by taking $1\mathbf{A}_d = 2\mathbf{A}$. Henceforth, we follow the same arithmetic cost presentation. Let's apply the Granger and Scott observation [23] on 18-limb and we find the arithmetic cost as $171\mathbf{M} + 664\mathbf{A}$. Similarly, applying the Karatsuba algorithm and using the Scott's work in [33] to decide about the cross-over point between the schoolbook and Karatsuba, we have two choices. First choice, apply the Karatsuba 3-way identity in [36] followed by the schoolbook when the size of multiplicands become 6. So, using this technique we find the total arithmetic cost as $216\mathbf{M} + 564\mathbf{A}$ including the cost of reduction. Second choice, using the technique of Bernstein et al. in [5], we apply the reduced refined Karatsuba identity at level-1 to yield multiplicands of size 9. Then we apply the Karatsuba 3-way identity because of the cross-over point between the schoolbook and Karatsuba, as shown in [33]. Thus, we find the total arithmetic cost of the second choice as $162\mathbf{M} + 576\mathbf{A}$ including the cost of reduction. Note that the total arithmetic cost of our residue multiplication techniques are $168\mathbf{M} + 494\mathbf{A}, 180\mathbf{M} + 490\mathbf{A}$ and $189\mathbf{M} + 481\mathbf{A}$. Clearly, the total arithmetic cost of all our techniques for 32-bit platforms is less than the aforementioned techniques even if one takes $1\mathbf{M} = 3\mathbf{A}$. Note that we have not included the cost of shift operation in all the aforementioned techniques because it is not significant and in our case all the shift operations are performed on single-word.

Now that the arithmetic cost of different techniques is provided, it's time to perform timing tests. In our timing tests, we run the residue multiplication functions for 10^7 iterations over 100 random values generated at run-time and the results are provided in Table 4.2. The Arbitrary Degree Karatsuba (ADK) is more efficient than the schoolbook for 9-limb multiplication, as shown in [33]. But [1] and [23] present even more efficient modular multiplication algorithm than ADK for modulus p with 9-limb. In [23], 155 clock cycles are reported for the modular multiplication algorithm and it is also mentioned that they tried different compilers and options. On the other hand, we got 165-cycle for the same modular multiplication

code in our test and the result is almost same as the one reported in [1]. Surprisingly, we find 136-cycle for the Hybrid version of our residue multiplication in [1] which is less than the reported 179-cycle in our earlier tests. Note that we have used the same residue multiplication function `gmul` and `TMV_product` as provided at `http://indigo.ie/~mscott/ws521.cpp` and `https://github.com/Shoukat-Ali/521-bit-Mersenne-Prime/blob/master/hybrid.c`, respectively. Similarly, for squaring, we use the same code provided at `https://github.com/Shoukat-Ali/521-bit-Mersenne-Prime/blob/master/ws521.cpp`.

The Table 4.2 shows that our cycles count for 64-bit squaring is 110 as compared to 105 in [23]. For 32-bit squaring, the number of cycles is 4 times of the 64-bit version and the increase in the number of multiplication and addition from 64- to 32-bit squaring is 3.8 and 3.77 times, respectively. The 550-cycle of the Technique-1 is 4.04 times of the Hybrid version cycle. While, the increase in the number of multiplication and addition from 64- to 32-bit residue multiplication are 3.5 and 3.62 times, respectively. The 574-cycle of the Technique-2 is around 4.22 times of the Hybrid version cycle. Whereas, the increase in the number of multiplication and addition from 64- to 32-bit residue multiplication are 3.11 and 3.71 times, respectively. Lastly, the 572-cycle of the Technique-3 is about 4.21 times of the Hybrid version cycle. While, the increase in the number of multiplication and addition from 64- to 32-bit residue multiplication are aroudn 3.33 and 3.68 times, respectively. It is to mention that Seo et al. [34] reported 350- and 708-cycle on Cortex-A15 and Cortex-A9, respectively, for their parallel multiplication modulo 521-bit Mersenne prime. As compared to the 20-limb used by Seo et al. [34], our techniques use 18-limb which also implies that the parallel implementation of our residue multiplication techniques might have way better result than [34] on Cortex-A15.

From Table 4.2, it is evident that Technique-1 is our most efficient residue multiplication. So, Technique-1 is our selection for which we show that the technique does not cause overflow in single- and double-word. In spite of their less arithmetic cost, both Technique-2 and Technique-3 take more clock cycles than Technique-1. The most evident factor is the inefficient mixed-sign multiplication performed the GCC compiler. Actually, all our codes are written in C programming language with no use

Table 4.2: Number of operations for 32-bit and 64-bit residue multiplication techniques and their respective clock cycles counts. The cycles count were obtained by generating 100 random integers at run-time and executing each respective function for 10^7 times on each value.

Technique	**Arithmetic cost**	**Clock Cycles**
Squaring (64-bit)	$45\mathbf{M} + 91\mathbf{A}$	110
Granger-Scott (64-bit)	$45\mathbf{M} + 176\mathbf{A}$	165
Hybrid version (64-bit)	$54\mathbf{M} + 133\mathbf{A}$	136
Squaring (32-bit)	$171\mathbf{M} + 343\mathbf{A}$	440
Technique-1 (this chapter)	$189\mathbf{M} + 481\mathbf{A}$	550
Technique-2 (this chapter)	$168\mathbf{M} + 494\mathbf{A}$	574
Technique-3 (this chapter)	$180\mathbf{M} + 490\mathbf{A}$	572

of intrinsics and SIMD/assembly instructions. Techniques-2 has more mixed-sign multiplication than Technique-3. In order to obtain better timing results, we did different type casting and mixed-sign addition tricks, that we know, for Technique-2 and Technique-3.

4.6 Worst-case bitlength analysis of Technique-1

We are using reduced-radix representation for 521-bit residues such that the bitlength of the limbs on 32-bit platforms is more than the recommended range in [33]. In other words, overflow is a big challenge to ensure correctness and efficiency on 32-bit platforms. Moreover, the number of limbs and our residue representation aggravate the situation further. As aforementioned in earlier section, overflow is not a threat for 64-bit residue multiplication.

In Section 4.1, it is shown that overflow is avoided in single-word on 32-bit platforms. Now, in this section, we prove that the Technique-1 does not cause overflow in double-word (64-bit) in spite of using our residue representation. For the sake of simplicity and the worst-case analysis, the (non-negative) residues X and Y are considered the largest 522-bit value. In other words, the maximum values of all the limbs of X and Y are $2^{29} - 1$. Before starting our proof, it is reminded that based on (2.1) and (2.2) we restrict addition and subtraction to Toeplitz matrix and vector, respectively. This implies that the result of P_1, P_5 and P_6 is always unsigned/non-negative while the

result of P_2, P_3 and P_4 is signed (negative/non-negative). Therefore, our focus is on the vectors Y_0, Y_1 and Y_2 which determine the value of P_2, P_3 and P_4.

Case-1:

$$Y_0 = \begin{bmatrix} 2^{29}-1 \\ 2^{29}-1 \\ 2^{29}-1 \\ 2^{29}-1 \\ 2^{29}-1 \\ 2^{29}-1 \end{bmatrix}, \quad Y_1 = \begin{bmatrix} 2^{29}-1 \\ 2^{29}-1 \\ 2^{29}-1 \\ 2^{29}-1 \\ 2^{29}-1 \\ 2^{29}-1 \end{bmatrix}, \quad Y_2 = \begin{bmatrix} 2^{29}-1 \\ 2^{29}-1 \\ 2^{29}-1 \\ 2^{29}-1 \\ 2^{29}-1 \\ 2^{29}-1 \end{bmatrix}$$

These vector values give the maximum value of P_1, P_5 and P_6 as shown below

$$\begin{aligned} P_1 &= (2^2(2^{29}-1))(2^{29}-1)6 = 2^{62}+2^{61}-2^{34}-2^{33}+2^4+2^3 \\ P_5 &= (5(2^{29}-1))(2^{29}-1)6 \\ &= 2^{62}+2^{61}+2^{60}+2^{59}-2^{34}-2^{33}-2^{32}-2^{31}+2^4+2^3+2^2+2 \\ P_6 &= (6(2^{29}-1))(2^{29}-1)6 \\ &= 2^{63}+2^{60}-2^{35}-2^{32}+2^5+2^2 \end{aligned}$$

Since $Y_0 = Y_1 = Y_2$ then the resultant vector of P_2, P_3 and P_4 will be $[0, \ldots, 0]$. Which implies that the result of P_2, P_3 and P_4 will be zero. Hence, we have

$$\begin{aligned} P_3 + P_4 + P_6 &= P_6 \\ P_2 - P_4 + P_5 &= P_5 \\ P_1 - P_2 - P_3 &= P_1 \end{aligned}$$

So, none of the result is more than 64-bit unsigned. Thus, overflow is avoided in double-word by taking the summation as 64-bit unsigned integer.

Case-2:

$$Y_0 = \begin{bmatrix} 2^{29}-1 \\ 2^{29}-1 \\ 2^{29}-1 \\ 2^{29}-1 \\ 2^{29}-1 \\ 2^{29}-1 \end{bmatrix}, \quad Y_1 = \begin{bmatrix} 2^{28}-1 \\ 2^{28}-1 \\ 2^{28}-1 \\ 2^{28}-1 \\ 2^{28}-1 \\ 2^{28}-1 \end{bmatrix}, \quad Y_2 = \begin{bmatrix} 2^{29}-1 \\ 2^{29}-1 \\ 2^{29}-1 \\ 2^{29}-1 \\ 2^{29}-1 \\ 2^{29}-1 \end{bmatrix}$$

In this case, the computation of P_1, P_5 and P_6 result in the following values

$$
\begin{aligned}
P_1 &= (2^2(2^{29}-1))(2^{29}-1)6 = 2^{62}+2^{61}-2^{34}-2^{33}+2^4+2^3 \\
P_5 &= (5(2^{29}-1))(2^{28}-1)6 \\
&= 2^{61}+2^{60}+2^{59}+2^{58}-2^{34}-2^{32}-2^{31}-2^{29}+2^4+2^3+2^2+2 \\
P_6 &= (6(2^{29}-1))(2^{29}-1)6 \\
&= 2^{63}+2^{60}-2^{35}-2^{32}+2^5+2^2
\end{aligned}
$$

At level-1, the resultant vector of P_2, P_3 and P_4 will be $[2^{28},\ldots,2^{28}], [0,\ldots,0]$ and $[-2^{28},\ldots,-2^{28}]$, respectively. So, the result of P_3 will be zero and the computation of P_2 and P_4 at level-2 are shown below

$$
\begin{aligned}
P_{2,1} &= 0 \\
P_{2,2} &= P_{2,3} = (2(2^{29}-1))(2^{28})3 = 2^{59}+2^{58}-2^{30}-2^{29} \\
P_{4,1} &= 0 \\
P_{4,2} &= P_{4,3} = (2(2^{30}-2))(-2^{28})3 = -(2^{60}+2^{59}-2^{31}-2^{30})
\end{aligned}
$$

Finally, we have to compute

$$
\begin{aligned}
P_3 + P_4 + P_6 &= 2^{63}-2^{58}-2^{35}-2^{31}+2^{29}+2^5+2^2 \\
P_2 - P_4 + P_5 &= 2^{63}+2^{60}+2^{59}-2^{35}-2^{31}+2^4+2^3+2^2+2 \\
P_1 - P_2 - P_3 &= 2^{62}+2^{61}-2^{59}-2^{58}-2^{34}-2^{33}+2^{30}+2^{29}+2^4+2^3
\end{aligned}
$$

Here, none of the result is more than 64-bit unsigned. Thus, overflow is avoided in double-word by taking the summation as 64-bit unsigned integer.

Case-3:

$$
Y_0 = \begin{bmatrix} 2^{28}-1 \\ 2^{28}-1 \\ 2^{28}-1 \\ 2^{28}-1 \\ 2^{28}-1 \\ 2^{28}-1 \end{bmatrix}, \quad
Y_1 = \begin{bmatrix} 2^{29}-1 \\ 2^{29}-1 \\ 2^{29}-1 \\ 2^{29}-1 \\ 2^{29}-1 \\ 2^{29}-1 \end{bmatrix}, \quad
Y_2 = \begin{bmatrix} 2^{29}-1 \\ 2^{29}-1 \\ 2^{29}-1 \\ 2^{29}-1 \\ 2^{29}-1 \\ 2^{29}-1 \end{bmatrix}
$$

This time, the computation of P_1, P_5 and P_6 result in the following values

$$\begin{aligned}
P_1 &= (2^2(2^{29}-1))(2^{28}-1)6 = 2^{61}+2^{60}-2^{34}-2^{31}+2^4+2^3 \\
P_5 &= (5(2^{29}-1))(2^{29}-1)6 \\
&= 2^{62}+2^{61}+2^{60}+2^{59}-2^{34}-2^{33}-2^{32}-2^{31}+2^4+2^3+2^2+2 \\
P_6 &= (6(2^{29}-1))(2^{29}-1)6 \\
&= 2^{63}+2^{60}-2^{35}-2^{32}+2^5+2^2
\end{aligned}$$

At level-1, the resultant vector of P_2, P_3 and P_4 will be $[-2^{28}, \ldots, -2^{28}], [-2^{28}, \ldots, -2^{28}]$ and $[0, \ldots, 0]$, respectively. This time, the result of P_4 will be zero. For the sake of simplicity and the worst-case analysis, we further assume that all the matrix entries of P_3 are $2^{30}-2$. So, the computation of P_2 and P_3 at level-2 are shown below

$$\begin{aligned}
P_{2,1} &= 0 \\
P_{2,2} &= P_{2,3} = (2(2^{29}-1))(-2^{28})3 = -(2^{59}+2^{58}-2^{30}-2^{29}) \\
P_{3,1} &= 0 \\
P_{3,2} &= P_{3,3} = (2(2^{30}-2))(-2^{28})3 = -(2^{60}+2^{59}-2^{31}-2^{30})
\end{aligned}$$

Finally, we have to compute

$$\begin{aligned}
P_3 + P_4 + P_6 &= 2^{63}-2^{59}-2^{35}-2^{30}+2^5+2^2 \\
P_2 - P_4 + P_5 &= 2^{61}+2^{60}-2^{34}-2^{32}-2^{30}+2^4+2^3+2^2+2 \\
P_1 - P_2 - P_3 &= 2^{62}+2^{60}+2^{58}-2^{34}-2^{32}-2^{31}-2^{29}+2^4+2^3
\end{aligned}$$

So, none of the result is more than 63-bit unsigned. Thus, there is no overflow in double-word.

CHAPTER 5

SCALAR MULTIPLICATION

In this chapter we discuss the vital elliptic curve operation of scalar/point multiplication kP where k is an integer and P is a point on elliptic curve defined over a prime field. Actually, the efficiency of elliptic curve cryptography (ECC) is directly proportional to the efficiency of scalar multiplication. The single-scalar multiplication kP is used in: public-key generation, singing messages, and shared-secret computation/Diffie-Hellman. The base point P is a standard/fixed point in the operations of public-key generation and singing messages. If P is fixed/known, then kP is called fixed-base scalar multiplication. On the other hand, the point P is unknown in the shared-secret computation. So, if P is unknown, then kP is called variable-base scalar multiplication. The scalar both in fixed- and variable-base is secret and must be protected from any kind of leakage in computation. For signature verification in ECC, one needs to compute $kP + lQ$, called double-scalar multiplication, where l is an integer and Q is a point. Note that our work is focused on the public-key generation and shared-secret computation.

In fixed-base scalar multiplication, one can perform precomputations using P to speedup the scalar multiplication kP. On the other hand, no computations can be performed in advance using P in variable-base scalar multiplication. Which implies that the computation of variable-base scalar multiplication takes longer than fixed-base scalar multiplication. That's why the efficient computation of variable-base scalar multiplication has received more attention. In Section 5.1, we briefly present some of the different techniques to compute the variable-base scalar multiplication. In Section 5.2, we discuss our first implementation of the constant-time variable-base scalar mul-

tiplication using our 64-bit residue multiplication modulo p and $2p$ where $p = 2^{521}-1$. Next, we discuss some of the algorithms to compute fixed-base scalar multiplication in Section 5.3. Finally, we discuss our implementation — both constant-time fixed- and variable-base scalar multiplication — in detail using state-of-the-art algorithms proposed by Bos et al. [14] in Section 5.4.

5.1 Variable-base Scalar Multiplication

Recall that $E(\mathbb{F}_p)$ is the set of points on elliptic curve E defined over prime field $\mathbb{F}_p$. Now, let's assume that the order of $E(\mathbb{F}_p)$ is nh where n is prime and h is a small integer. Furthermore, $P, Q \in E(\mathbb{F}_p)$ where both have order n and the secret integer k is selected randomly from the interval $[2, n-1]$. Also, suppose that the bitlength of the integer k is m where $m = \lceil log_2 n \rceil$ so, the binary representation of k is $(k_{m-1}, \ldots, k_1, k_0)_2$.

The most basic technique of computing kP is the use of repeated-double-and-add method which corresponds to the repeated-square-and-multiply method for exponentiation. To make things more clear, let's do a simple example, $21 \cdot P = (((((P \cdot 2 + 0)2 + P)2 + 0)2) + P$. The Algorithm 13 represents the left-to-right repeated-double-and-add method for scalar multiplication. The expected running time of the

Algorithm 13 Left-to-right repeated-double-and-add method for scalar multiplication

Input: $k = (k_{m-1}, \ldots, k_1, k_0)$ and $P \in E(\mathbb{F}_p)$

Output: kP

1: $Q \leftarrow \infty$
2: **for** i from $m-1$ downto 0 **do**
3: $\quad Q \leftarrow 2Q$
4: $\quad$ **if** $k_i = 1$ **then**
5: $\quad\quad Q \leftarrow Q + P$
6: Return Q

Algorithm 13 is approximately

$$m \cdot D + (m/2) \cdot A$$

where D and A stands for point doubling and point addition, respectively. Clearly, the number of point doubling cannot be altered but the number of relatively expensive point addition. For that purpose, we introduce a signed digit representation called *width-w Non-Adjacent Form (NAF)*. Actually, the *width-w NAF* is the extension of NAF.

Definition For a positive integer $w \geq 2$, a *width-w NAF* of a positive integer k is an expression $k = \sum_{i=0}^{l-1} 2^i k_i$ where each nonzero coefficient k_i is odd, $|k_i| < 2^{w-1}$ and $k_{l-1} \neq 0$. For any w consecutive digits, at most one of them is nonzero and l is called the length of the width-w NAF.

For a positive integer k, the properties of width-w NAF are given below:

- k has a unique width-w NAF representation and denoted by $\text{NAF}_w(k)$.
- The length of $\text{NAF}_w(k)$ is at most one more than the bitlength of k.
- Among all width-w NAFs of length l, the average density of nonzero digits is approximately $1/(w+1)$.

The Algorithm 14 represents the left-to-right repeated-double-and-add method for scalar multiplication using $\text{NAF}_2(k)$. Note that the $\text{NAF}_2(k)$ is also called NAF and denoted by $\text{NAF}(k)$. From the properties of the $\text{NAF}(k)$, one can expect the running

Algorithm 14 $\text{NAF}(k)$ method for scalar multiplication

Input: $k = \sum_{i=0}^{l-1} 2^i k_i$ where $k_i \in \{0, \pm 1\}$ and $P \in E(\mathbb{F}_p)$

Output: kP

1: $Q \leftarrow \infty$
2: **for** i from $l-1$ downto 0 **do**
3: $\quad Q \leftarrow 2Q$
4: $\quad$ **if** $k_i = 1$ **then** $\quad Q \leftarrow Q + P$
5: $\quad$ **if** $k_i = -1$ **then** $\quad Q \leftarrow Q - P$
6: Return Q

time of the Algorithm 14 to be approximately

$$m \cdot D + (m/3) \cdot A.$$

Remember that l at most one more than m, therefore, we have expressed the expected running time in terms of m.

From Algorithm 13, Algorithm 14 and width-w NAF, one can deduce that instead of process one digit of k, why not to process w digits of k at a time and this technique is called window method. Definitely, the window method requires some extra memory to store the possible points iP for $i = 1, 3, \ldots 2^{w-1} - 1$ where the odd values are due to the definition of width-w NAF. The Algorithm 15 represents the width-w NAF method for scalar multiplication. So, the expected running time of the Algorithm 15

Algorithm 15 $\text{NAF}_w(k)$ method for scalar multiplication

Input: $P \in E(\mathbb{F}_p)$, $\text{NAF}_w(k) = \sum_{i=0}^{l-1} 2^i k_i$ and widow width w

Output: kP

1: Compute $P_i = iP$ for $i = 1, 3, \ldots, 2^{w-1} - 1$
2: $Q \leftarrow \infty$
3: **for** i from $l - 1$ downto 0 **do**
4: $\quad Q \leftarrow 2Q$
5: $\quad$ **if** $k_i \neq 0$ **then**
6: $\quad\quad$ **if** $k_i > 0$ **then** $\quad Q \leftarrow Q + P_i$
7: $\quad\quad$ **else** $\quad Q \leftarrow Q - P_i$
8: Return Q

is approximately

$$\underbrace{1 \cdot D + \left(2^{w-2} - 1\right) \cdot A}_{P_i \text{computation}} + \underbrace{m \cdot D + \left(\frac{m}{w+1}\right) \cdot A}_{\text{loop computation}}.$$

The next algorithm is based on the idea of skipping the consecutive zero entries rather than processing them one at a time. The technique is called *Sliding Window* and the Algorithm 16 represents the sliding window method for scalar multiplication using $\text{NAF}(k)$. The digits of $\text{NAF}(k)$ are processed from left-to-right for window of width w such that the value obtained in the window is odd. So, the length of consecutive zeros between windows in sliding window method affects the running time. On average, the length of a run of zeros between windows is given below.

$$v(w) = \frac{4}{3} - \frac{(-1)^w}{3 \cdot 2^{w-2}}$$

Algorithm 16 Sliding window method for scalar multiplication using NAF(k)

Input: $P \in E(\mathbb{F}_p)$, NAF$(k) = \sum_{i=0}^{l-1} 2^i k_i$ and widow width w

Output: kP

1: Compute $P_i = iP$ for $i = 1, 3, \ldots, \big(2(2^w - (-1)^w)/3\big) - 1$
2: $Q \leftarrow \infty$
3: $i = l - 1$
4: **while** $i \geq 0$ **do**
5: **if** $k_i = 0$ **then**
6: $t \leftarrow 1, u \leftarrow 0$
7: **else** find the largest $t \leq w$ such that $u \leftarrow (k_i, \ldots, k_{i-t+1})$ is odd
8: $Q \leftarrow 2^t Q$
9: **if** $u > 0$ **then**
10: $Q \leftarrow Q + P_u$
11: **else if** $u < 0$ **then**
12: $Q \leftarrow Q - P_u$
13: $i \leftarrow i - t$
14: Return Q

Therefore, the running time of Algorithm 16 is expected to be

$$\underbrace{1 \cdot D + \left(\frac{2^w - (-1)^w}{3} - 1\right) \cdot A}_{P_i\text{computation}} + \underbrace{m \cdot D + \left(\frac{m}{w + v(w)}\right) \cdot A}_{\text{loop computation}}.$$

The differences between Algorithm 15 and Algorithm 16 are the number and storage of P_i, and the cost of loop computation. Clearly, for a window width w, the Algorithm 15 computes lesser number of P_i to store than Algorithm 16. But the cost of the loop computation in Algorithm 15 is more than the Algorithm 16.

From implementation point of view, the selection of window width w is tricky. The larger value of w implies larger computer memory requirement and access to large memory can lead to different problems. The first and foremost is the leakage of information in the side-channel attacks when large memory is used involving the secret integer k. Moreover, access to large memory is slow and can significantly affect the efficiency. Therefore, in practice we find $w \in \{4, 5, 6\}$. Note that among the other factors the selection of coordinates for point representation is very important for the efficient computation of scalar multiplication.

5.2 Our First implementation

This implementation is part of our initial work focused on 64-bit platforms. After finishing our work on 64-bit residue multiplication, we implemented constant-time variable-base scalar multiplication for the standard NIST curve P-521 and Edwards curve E-521 using the public codes of [23] available at `http://indigo.ie/~mscott/ws521.cpp` and `http://indigo.ie/~mscott/ed521.cpp`. Unlike the codes of [23], the clock cycles are counted using the proposed technique of Paoloni [32] i.e. using rdtsc() and rdtscp(). Note that the testing environment is same as that of the different versions of our 64-bit residue multiplication and for further details see Section 3.2.2.

The C++ codes of [23] are for modulus $2p$ and in our timing tests we have used both modulus $2p$ and p. For modulus $2p$, we have only replaced the modular multiplication of [23] by the different versions of our residue multiplication. On the other hand,

we have amended the functions *scr(), gsqr(), gsqr2()*, and *gmuli()* according to the requirements of modulus p. To obtain consistent clock cycles counts, we execute the cycle counting loop with different number of iterations and among those values we find 40-iteration as the most consistent. Hence, in this part of our work the cycles counts for constant-time variable-base scalar multiplication codes are provided in 40-iteration loop.

From the results of the clock cycles count of the different versions of our algorithm with respect to the multiplication algorithm in [23], one would also expect higher number of clock cycles for scalar multiplication using our residue multiplication. While measuring the clock cycles and playing with the GCC compiler we observed a strange behavior that when either the multiplication algorithm in [23] or our algorithm (Hybrid version) is called consecutively more than 2 times in a loop then our algorithm starts to take less and less number of cycles. Therefore, we find the cycles counts for scalar multiplication using each version of our algorithm to be less than the Granger-Scott algorithm. Since compiler optimization/behavior is not our domain so, we don't know why GCC compiler behaves like this. For each version of our multiplication algorithm we have used the optimal implementation on our machine. But we report the cycles counts using the Hybrid version of our algorithm [1] because it has the least number of clock cycles among the different versions. Using the command *openssl speed ecdh* on our machine where the installed version is $1.0.2$g. For the NIST P-521 it reports 1745.1 operations per second which is approximately $1,604,493$ cycles count. The clock cycles counts are given in Table 5.1 and each program was executed at optimization level-3 i.e. $-O3$. Although the number of clock cycles counts with CACHE_SAFE (defined for cache safety) is more than without CACHE_SAFE option but we prefer the former choice to ensure the constant-time implementation requirement. So, through testing with CACHE_SAFE on our machine we find the fixed window of width 4 as the optimal choice both for Granger-Scott and our algorithm. Hence, the cycles counts are for windows of width 4 with CACHE_SAFE. We have executed the scalar multiplication programs both for P-521 and E-521 multiple times in order to obtain the least cycles counts. In case of P-521 for Granger-Scott algorithm we find $1,332,165$ as the minimum mean cycles

[1] `https://github.com/Shoukat-Ali/521-bit-Mersenne-Prime/blob/master/ed521.cpp`
`https://github.com/Shoukat-Ali/521-bit-Mersenne-Prime/blob/master/ws521.cpp`

Table 5.1: Clock Cycles counts of constant-time variable-base scalar multiplication; GS stands for Granger-Scott algorithm, p and $2p$ stand for modulus p and $2p$ implementation of the Hybrid version of our residue multiplication, respectively

openSSL	**P-521**	**E-521**
$\approx 1,604,493$	GS $= 1,332,165$	GS $= 1,148,871$
	$2p = 1,270,130$	$2p = 1,073,127$
	$p = 1,251,502$	$p = 1,055,105$

count. While for both modulus $2p$ and p of the Hybrid version of our algorithm we find $1,270,130$ and $1,251,502$ cycles, respectively. Similarly, in E-521 for Granger-Scott algorithm we find $1,148,871$ as the minimum mean cycles count. However, for the modulus $2p$ and p of the Hybrid version of our algorithm we find $1,073,127$ and $1,055,105$ cycles respectively. Hence, the experimental results support our observation and intuition for multiple calls (in thousands or more) and modulus choice.

In our new timing test, not recommended for code benchmarking on eBACS web site [8], we use Intel Core i7 $-$ 2670QM CPU @ 2.20GHz with Turbo Boost and Hyper-Threading enabled. We compile and execute the same earlier codes of constant-time variable-base scalar multiplication with the same command. Like before, we find the timing results of constant-time variable-base scalar multiplication using the Hybrid version of our algorithm to be better than the Granger-Scott algorithm [23]. Still the minimum cycles counts are found for modulus p. For the curve E-521, the new tests show the minimum cycles counts as GS $= 893,371$ and $p = 847,145$ for the window width of size 4. On the other hand, for the curve P-521, we find the minimum cycles counts as GS $= 1,063,370$ and $p = 1,001,180$ for window width of size 5 and 4, respectively.

5.3 Fixed-base Scalar Multiplication

By now, we know that in fixed-base scalar multiplication the base-point P is public. Therefore, in order to speed up the computation of kP, both the communicating entities can perform computation based on P in advance. Indeed, anyone can perform computation based on P including the malicious attacker. The most basic way of

speeding up the computation of kP is to compute all the point doubling D in advance so that only point addition A is left to be performed.

In practice, large prime numbers are used and in our case it is 521-bit. Let's assume that $m = 521$, the points 2^iP for $i = 0, \ldots, 520$ are represented in affine coordinates (x, y) and the underlying platform is 64-bit. In our case, we have 9-limb coordinates where each limb is stored in a 64-bit word which implies that for storing all the points 2^iP, one needs approximately $9 \cdot 64 \cdot 2 \cdot 2^{521}$-bit or $9 \cdot 8 \cdot 2 \cdot 2^{521}$-byte of memory. Which is insanely very very large amount of memory.

A practical way of precomputing the points 2^iP in order to accelerate the computation of kP is presented in Algorithm 17. For a window width w we have $k = (K_{d-1}, \ldots, K_1, K_0)_{2^w}$ where $d = \lceil m/w \rceil$ and $P \in E(\mathbb{F}_p)$. First we compute and store $P_i = 2^{wi}P$ for $i = 0, \ldots, d-1$ in advance. Clearly, the memory required for the storage of the points $2^{wi}P$ is realistic.

Algorithm 17 Windowing method for scalar multiplication

Input: Window width w, $k = (K_{d-1}, \ldots, K_1, K_0)_{2^w}$, $P_i = 2^{wi}P$ for $i = 0, \ldots, d-1$

Output: kP

1: $Q \leftarrow \infty$
2: $R \leftarrow \infty$
3: **for** j from $2^w - 1$ downto 1 **do**
4: **for** each i such that $K_i = j$ **do**
5: $R \leftarrow R + P_i$
6: $Q \leftarrow Q + R$
7: Return Q

Next, we discuss another algorithm called the *comb method*. In this method, the binary representation of k is first padded with $dw - m$ 0s as the most significant bits. Then the binary representation is divided into w bit strings such that each string is of the same length d so, we have $k = K^{w-1}|| \cdots ||K^1||K^0$ where $||$ represents the

concatenation of bit strings. Actually, the bit strings K^j are written follows:

$$\begin{bmatrix} K^0 \\ K^1 \\ \vdots \\ K^{w-1} \end{bmatrix} = \begin{bmatrix} K^0_{d-1} & \cdots & K^0_0 \\ K^1_{d-1} & \cdots & K^1_0 \\ \vdots & \vdots & \vdots \\ K^{w-1}_{d-1} & \cdots & K^{w-1}_0 \end{bmatrix} = \begin{bmatrix} k_{d-1} & \cdots & k_0 \\ k_{2d-1} & \cdots & k_d \\ \vdots & \vdots & \vdots \\ k_{wd-1} & \cdots & k_{(w-1)d} \end{bmatrix}$$

Each column is processed one at a time such that for all the possible bit strings $(a_{w-1}, \ldots, a_1, a_0)$, the following points are computed and stored in advance.

$$[a_{w-1}, \ldots, a_1, a_0]P = a_{w-1}2^{(w-1)d}P + \cdots + a_1 2^d P + a_0 P$$

The use of comb method to compute the scalar multiplication is shown as Algorithm 18.

Algorithm 18 Comb method for scalar multiplication

Input: Window width w, $d = \lceil m/w \rceil$, $[a_{w-1}, \ldots, a_1, a_0]P$

Output: kP

1: $Q \leftarrow \infty$
2: **for** i from $d-1$ downto 0 **do**
3: $\quad Q \leftarrow 2Q$
4: $\quad Q \leftarrow Q + [K_i^{w-1}, \ldots, K_i^1, K_i^0]P$
5: Return Q

5.4 Our Second implementation

The efficiency ECC is determined by the efficiency of scalar multiplication, therefore, one finds different techniques in literature to efficiently compute the operation. Our work is focused on faster residue multiplication in 521-bit Mersenne prime modulus to improve the efficiency of constant-time single-scalar multiplication. In literature, it is well-known that the computation of the scalar multiplication should be independent of the secret scalar. So that, an attacker cannot easily derive the scalar by (simple) side-channel attacks. Therefore, there should not be any conditional branching on the secret scalar and similarly, the lookups in the precomputed table should not reveal

the secret indexes. Using our 64- and 32-bit residue multiplication, we have implemented the constant-time fixed- and variable-base (single) scalar multiplication. We have tested our software using fixed-window of width 4, 5 and 6. The efficiency of ECC is directly proportional to the efficiency of variable-base scalar multiplication and that is why, it has received more attention. To show the efficiency of our residue multiplication techniques, we have implemented the scalar multiplication for the standard NIST curve P-521 and Edwards curve E-521. We use the same testing environment that we did for our 32-bit residue multiplication techniques. For further details see Section 4.5.

For constant-time variable-base scalar multiplication using fixed-window w, we have implemented the "*Algorithm* 1" [14] and instead of 2^{w-2}, the size of our pre-computed table is 2^{w-1}. Similarly, the constant-time fixed-base scalar multiplication is implemented using the modified LSB-set comb method i.e. "*Algorithm* 7" [14]. Note that, for the sake of efficiency we have selected the point addition and doubling formulas from the Explicit Formulas Database (EFD) [9], managed by Daniel J. Bernstein and Tanja Lange, instead of the given formulas in [14].

The constant-time inversion is computed by powering a field element x by $p - 2 = 2^{521} - 3$ (Fermat's Little Theorem) and it's the same method used in [1, 23, 34]. Hence, the multiplicative inverse of x is computed with $520\mathbf{S} + 13\mathbf{M}$ where **S** stands for single-word squaring. Like [1, 23], multiplication by curve parameter — curve E-521 — is performed component-wise and reduced in-place for 32- and 64-bit implementation of the scalar multiplication. To ensure the limbs satisfy our residue representation such that overflow doesn't occur, we have used the same idea of *Short Coefficient Reduction* (SCR) function as in [1, 23]. In case of 64-bit implementation, field addition, subtraction and multiplication by small (fixed) constant(s) — other than curve parameter — are computed component-wise and don't require to be reduced in order to use as input to multiplication and squaring function. Because there is enough free bit space both in single- and double-word to easily avoid overflow. Unfortunately, things are more challenging for 32-bit scalar multiplication due to small free bit space. So, field addition, subtraction and multiplication by small (fixed) constant(s) — other than curve parameter — can easily cause overflow. Therefore, we perform in-place reduction where necessary, for efficiency purpose.

5.4.1 Point Arithmetic Formulas

We have selected the fastest point doubling and addition formulas for short Weierstrass curve P-521 — with curve parameter $a = -3$ and $b = 10938490380737342$ $74511112390766805569936207598951683748994586394495953116150735016013708$ $73757375962324859213229670631330943845253159101291214232748847898598 4$ — from the EFD site [9] and the curve is defined in [27]. Suppose $P_1 = (X_1, Y_1, Z_1)$ is a Jacobian point then $2P_1 = (X_3, Y_3, Z_3)$ is computed as follows:

$$R_0 = Z_1^2, \quad R_1 = Y_1^2, \quad R_2 = X_1 \cdot R_1, \quad R_3 = 3(X_1 + R_0) \cdot (X_1 - R_0),$$
$$X_3 = R_3^2 - 8R_2, \quad Z_3 = (Y_1 + Z_1)^2 - R_0 - R_1,$$
$$Y_3 = R_3 \cdot (4R_2 - X_3) - 8R_1^2.$$

Suppose $P_2 = (X_2, Y_2, 1)$ is an affine point and not equal to P_1. Then $2P_2 = (X_3, Y_3, Z_3)$ is computed as follows:

$$R_0 = X_2^2, \quad R_1 = Y_2^2, \quad R_2 = R_1^2, \quad R_3 = 2((X_2 + R_1)^2 - R_0 - R_2),$$
$$R_4 = 3(R_0 - 1), \quad X_3 = R_4^2 - 2R_3, \quad Y_3 = R_4 \cdot (R_3 - X_3) - 8R_2,$$
$$Z_3 = 2Y_2.$$

The mixed addition of Jacobian-affine coordinate $P_3 = P_1 + P_2 = (X_3, Y_3, Z_3)$ is computed as follows:

$$R_0 = Z_1^2, \quad R_1 = X_2 \cdot R_0, \quad R_2 = Y_2 \cdot Z_1 \cdot R_0, \quad R_3 = R_1 - X_1,$$
$$R_4 = R_3^2, \quad R_5 = 4R_4, \quad R_6 = R_3 \cdot R_5, \quad R_7 = 2(R_2 - Y_1),$$
$$R_8 = X_1 \cdot R_5, \quad X_3 = R_7^2 - R_6 - 2R_8,$$
$$Y_3 = R_7 \cdot (R_8 - X_3) - 2Y_1 \cdot R_6, \quad Z_3 = (Z_1 + R_3)^2 - R_0 - R_4.$$

Suppose $P_2 = (X_2, Y_2, Z_2)$ and P_1 are both Jacobian points. Then we compute $P_3 = (X_3, Y_3, Z_3) = P_1 + P_2$ as follows:

$$R_0 = Z_1^2, \quad R_1 = Z_2^2, \quad R_2 = X_1 \cdot R_1, \quad R_3 = X_2 \cdot R_0,$$
$$R_4 = Y_1 \cdot Z_2 \cdot R_1, \quad R_5 = Y_2 \cdot Z_1 \cdot R_0, \quad R_6 = R_3 - R_2,$$
$$R_7 = (2R_6)^2, \quad R_8 = R_6 \cdot R_7, \quad R_9 = 2(R_5 - R_4), \quad R_{10} = R_2 \cdot R_7,$$
$$X_3 = R_9^2 - R_8 - 2R_{10}, \quad Y_3 = R_9 \cdot (R_{10} - X_3) - 2R_4 \cdot R_8,$$
$$Z_3 = \big((Z_1 + Z_2)^2 - R_0 - R_1\big) \cdot R_6.$$

We have selected the fastest point doubling and addition formulas for the Edwards curves E-521 — with curve paramter $a = 1$ and $d = -376014$ — from the EFD site [9] and the curve is defined on site [10]. Suppose $P_1 = (X_1, Y_1, Z_1, T_1)$ is an extended projective point then $2P_1 = (X_3, Y_3, Z_3, T_3)$ is computed as follows:

$$R_0 = X_1^2, \quad R_1 = Y_1^2, \quad R_2 = 2Z_1^2, \quad R_3 = (X_1 + Y_1)^2 - R_0 - R_1,$$
$$R_4 = R_0 + R_1, \quad R_5 = R_4 - R_2, \quad R_6 = R_0 - R_1, \quad X_3 = R_3 \cdot R_5,$$
$$Y_3 = R_4 \cdot R_6, \quad Z_3 = R_4 \cdot R_5, \quad T_3 = R_3 \cdot R_6.$$

As pointed out in [5], one multiplication operation is saved by using the same doubling formula for those cases where T coordinate is not required. Suppose $P_2 = (X_2, Y_2, 1, T_2)$ is an extended projective point then $2P_2 = (X_3, Y_3, Z_3, T_3)$ is computed as shown below. Note that this formula is taken from the Appendix A of [5] because it is efficient and we couldn't find it on EFD site [9].

$$R_0 = X_1^2, \quad R_1 = Y_1^2, \quad R_2 = 2T_1, \quad R_3 = R_0 + R_1, \quad R_4 = R_3 - 2,$$
$$R_5 = R_0 - R_1, \quad X_3 = R_2 \cdot R_5, \quad Y_3 = R_3 \cdot R_5, \quad Z_3 = R_3^2 - 2R_3,$$
$$T_3 = R_2 \cdot R_5.$$

Now, we compute $P_3 = (X_3, Y_3, Z_3, T_3) = P_1 + P_2$ as shown below and d is the curve parameter.

$$R_0 = X_1 \cdot X_2, \quad R_1 = Y_1 \cdot Y_2, \quad R_2 = T_1 \cdot dT_2,$$
$$R_3 = (X_1 + Y_1) \cdot (X_2 + Y_2) - R_0 - R_1, \quad R_4 = Z_1 - R_2,$$
$$R_5 = Z_1 + R_2, \quad R_6 = R_1 - R_0, \quad X_3 = R_3 \cdot R_4, \quad Y_3 = R_5 \cdot R_6,$$
$$Z_3 = R_4 \cdot R_5, \quad T_3 = R_3 \cdot R_6.$$

Suppose $P_2 = (X_2, Y_2, Z_2, T_2)$ then $P_3 = (X_3, Y_3, Z_3) = P_1 + P_2$ is computed as shown below. Again, note that this formula is taken from the Appendix A of [5] because it is an efficient approach.

$$R_0 = X_1 \cdot X_2, \quad R_1 = Y_1 \cdot Y_2, \quad R_2 = T_1 \cdot dT_2, \quad R_3 = Z_1 \cdot Z_2,$$
$$R_4 = (X_1 + Y_1) \cdot (X_2 + Y_2) - R_0 - R_1, \quad R_5 = R_3 - R_2,$$
$$R_6 = R_3 + R_2, \quad R_7 = R_1 - R_0, \quad X_3 = R_4 \cdot R_5, \quad Y_3 = R_6 \cdot R_7,$$
$$Z_3 = R_5 \cdot R_6.$$

Now, we compute $P_3 = (X_3, Y_3, Z_3, T_3) = P_1 + P_2$ as follows:

$$R_0 = X_1 \cdot X_2, \quad R_1 = Y_1 \cdot Y_2, \quad R_2 = T_1 \cdot dT_2, \quad R_3 = Z_1 \cdot Z_2,$$
$$R_4 = (X_1 + Y_1) \cdot (X_2 + Y_2) - R_0 - R_1, \quad R_5 = R_3 - R_2,$$
$$R_6 = R_3 + R_2, \quad R_7 = R_1 - R_0, \quad X_3 = R_4 \cdot R_5, \quad Y_3 = R_6 \cdot R_7,$$
$$Z_3 = R_5 \cdot R_6, \quad T_3 = R_4 \cdot R_7.$$

5.4.2 NIST Curve P-521

The 521-bit short Weierstrass NIST curve P-521 is standardized and defined in [27]. For this curve, we have implemented the constant-time variable- and fixed-base scalar multiplication algorithm proposed by J.W. Bos et al. in [14]. Jacobian and affine coordinates were chosen by [1, 23] for their constant-time variable-base scalar multiplication. Similarly, our choice is also Jacboian and affine coordinates and the formulas for point arithmetic are selected from EFD [9]. The point arithmetic formulas are provided in Appendix B.

In case of constant-time variable-base scalar multiplication, two point doublings and two point additions are selected from EFD [9]. In our implementation, one Jacobian and one affine (point) doubling are used. The affine doubling is used only once at the "*precomputation stage*" of "*Algorithm* 1". Out of the two point additions, one is mixed and one is Jacobian. For efficiency purpose, the mixed addition formula is used in two ways such that they differ by the destination/accumulator only. The Jacobian and a mixed addition are performed at the "*precomputation stage*". Lastly, the Jacobian doubling and mixed addition are used at the "*Evaluation stage*".

For constant-time fixed-base scalar multiplication [14], we use one Jacobian doubling, one Jacobian addition and one mixed addition. The formulas are same as ones that we used in variable-base. We perform the "*Offline computation*" by computing the lookup table points in Jacobian coordinate at first and then using Montgomery's trick of single inversion to convert all points to affine form. Hence, the lookup table points are in affine form and the addition in "*Online computation*" is performed using affine-Jacboian coordinate. We have used the table parameter $v = 3$ so, the size of the precomputed table is 3456, 6912 and 13824-byte for window width of 4, 5 and 6, re-

spectively. Although, using $v = 4$ will decrease the value of $e = \lceil t/(wv) \rceil$ where t is the bitlength of the base point (521-bit) it increases both the size of the precomputed table and the number of (relatively) expensive addition operation at the "*Evaluation Stage*". For example, for $w = 5$ and $v = 3$, we have $e = 35$, lookup table of 6912-byte and 105 additions. Then for $w = 5$ and $v = 4$, we have $e = 27$, lookup table of 9216-byte and 108 additions. By performing similar computation we observe that $v = 4$ is not useful for both $w = 4$ and $w = 6$. Similarly, things are not encouraging with other parameters too. Therefore, we didn't implement the case $v \geq 3$.

We have used C programming language to implement the constant-time variable- and fixed-base scalar multiplication for 64-bit platforms[2]. Unlike the 64-bit implementation, the 32-bit implementation is tricky because of the 2-bit space in single-word to hold the carries at the worst case. Therefore, it is expected to lose some efficiency in order to perform reduction on single-word. On the other hand, the point arithmetic formulas of P-521 also adds to inefficiency in our 32-bit implementation. So, to minimize the effect of these inefficient factors, we have used in-place reduction in (most of the) field operations. Like the 64-bit implementation, the constant-time variable- and fixed-base scalar multiplication for 32-bit platforms[3] are also implemented in C language.

5.4.3 Edwards curve E-521

The curve E-521 is taken from the SafeCurves site [10] where it is mentioned that the curve is independently recommended by Bernstein and Lange, Aranha et al. [2] and Hamburg [25]. As pointed in [5], faster point addition formulas for Edwards curve are in extended projective coordinate, therefore, we use the extended coordinate for variable- and fixed-base scalar multiplication. The point arithmetic formulas are provided in Appendix B. The advantage of the Edwards curves is that it is faster than NIST curves because the point arithmetic formulas are simpler. Moreover, due to the complete addition formulas, it is easier to securely implement the Edwards curves and there are no exceptional cases. Definitely, a few bits of security is sacrificed in select-

[2] https://github.com/Shoukat-Ali/Faster-Residue-Multiplication/tree/master/64-bit/P-521

[3] https://github.com/Shoukat-Ali/Faster-Residue-Multiplication/tree/master/32-bit/P-521

ing Edwards curve. Hence, unlike the implementation of the scalar multiplication of P-521, the scalar multiplication implementation of E-521 is efficient, constant-time and exceptional-less.

For constant-time variable-base scalar multiplication, we have used the same strategy and point arithmetic formulas that were used in [1, 5, 23] for efficiency. So, there are three versions of point doubling and two versions of point additon formulas based on the coordinates "Z" and "T". Unlike [14], where odd scalars are recoded only, we have used the same (secret) even scalar in the public code of [1, 23].

For the constant-time fixed-base scalar multiplication, we use two point doubling and two point addition formulas. Only, one of the point addition formulas is different from those that were used in variable-base while, the remaining formulas are same. To achieve efficient *Online computation*, all the points are in extended coordinate including the points in the precomputed table. Like the fixed-base scalar multiplication of P-521, we have taken the table parameter $v = 3$. So, the size of the precomputed table is 6912, 13824 and 27648-byte for window width of 4, 5 and 6, respectively. Although, using $v = 4$ will decrease the value of $e = \lceil t/(wv) \rceil$ where t is the bitlength of the base point (519-bit) it increases both the size of the precomputed table and the number of (relatively) expensive addition operation at the *Evaluation Stage*. For example, for $w = 6$ and $v = 3$, we have $e = 29$, lookup table 27648-byte and 87 additions. Then for $w = 6$ and $v = 4$, we have $e = 22$, lookup table 36864-byte and 88 additions. By performing similar computation we didn't find $v = 4$ useful for both $w = 4$ and $w = 5$. Similarly, the value of other parameters is also not encouraging. Therefore, we didn't implement the case $v \geq 3$.

Like the curve P-521, the C language is used to implement the constant-time variable- and fixed-base scalar multiplication for 64-bit platforms[4]. Although, the 32-bit implementation faces the same issues as in the case of P-521 the (simple) point arithmetic formulas of E-521 cause less overhead than P-521 in order to handle overflow in single-word. In this case, every field operation is implemented using in-place reduction. The constant-time variable- and fixed-base scalar multiplication for 32-bit

[4] https://github.com/Shoukat-Ali/Faster-Residue-Multiplication/tree/master/64-bit/E-521

Table 5.2: Clock Cycles count of the scalar multiplication for NIST curve P-521 and Edwards curve E-521 obtained at optimization level-3 of the GCC 5.4.0 compiler on Ubuntu 16.04 LTS. The timing tests were performed on Intel(R) Core i5 − 6402P CPU @ 2.80GHz with Turbo Boost disabled, Hyper-threading not supported and the machine was running on one core

Operation	**Window width**	**64-bit**		**32-bit**	
		E-521	P-521	E-521	P-521
Variable-base	w=4	802,110	936,160	3,078,998	3,754,095
	w=5	814,085	939,956	3,068,761	3,750,483
	w=6	872,790	1,008,740	3,203,760	3,913,668
Fixed-base	w=4	366,590	347,800	1,299,507	1,342,525
	w=5	335,027	317,652	1,214,847	1,183,615
	w=6	368,516	319,855	1,169,799	1,130,007

platforms[5] are implemented in C language too.

5.4.4 Timings

In this section, we present our timing results — clock cycles count — of the variable- and fixed-base scalar multiplication for 64- and 32-bit platforms using our residue multiplication algorithms. We executed our 64- and 32-bit C programs using the command *gcc -Wall -O3 program.c -o program.exe* and *gcc -Wall -m32 -O3 program.c -o program.exe*, respectively. Similarly, each program is tested for window width 4, 5 and 6 and in order to obtain the minimum cycle count, we performed 10^4 scalar multiplications for 10 iterations using one (secret) scalar. Note that the testing environment is same as aforementioned in Section 4.5 and the timing results are provided in Table 5.2. In case of fixed-base scalar multiplication, the results represent the timing of the *Evaluation Stage* of the "*Algorithm 7*" [14].

To obtain the timing result of OpenSSL 1.0.2g for curve P-521, we ran the command *openssl speed ecdhp521* which reports 2,298 operations per second. Which

[5] https://github.com/Shoukat-Ali/Faster-Residue-Multiplication/tree/master/32-bit/E-521

implies approximately $1,218,451$ clock cycles for one variable-base scalar multiplication. Then we tested the public code of Scott available at `http://indigo.ie/~mscott/ws521.cpp` and `http://indigo.ie/~mscott/ed521.cpp` and find the minimum clock cycles of $1,056,544$ and $895,372$, respectively, for window width-4. While the results of window width 5 and 6 are worse. Interestingly, the cycle counts of Scott's codes on our machine are less than what is reported in [23]. Note that Scott implemented the constant-time variable-base scalar multiplication only. Hamburg [25] has implemented E-521 on 64-bit Haswell processor using C with intrinsics and exploiting the tuned AVX2. He reports 803k and 234k cycles for constant-time variable- and fixed-base scalar multiplication, respectively. Even though 512-bit curves in Weierstrass and twisted Edwards form were used with different primes in [14], the timing results of variable- and fixed-base scalar multiplication are no match to Hamburg's results of E-521 in [25]. As shown in Table 5.2, window width 4 yields the optimal result for variable-base scalar multiplication using fixed-window for both P-521 and E-521. So, we have $936,160$ and $802,110$ cycles for P-521 and E-521, respectively. Hence, our 64-bit residue multiplication yields the best result — to the best of our knowledge — for variable-base scalar multiplication for P-521 and E-521. In Table 5.2, window width 5 has the optimal result for constant-time fixed-base scalar multiplication using the modified LSB-set comb method for P-521 and E-521. So, we have $317,652$ and $335,027$ cycles for P-521 and E-521, respectively. Note that our software are purely implemented in C without any use of intrinsics and SIMD/assembly instructions. Therefore, our software are independent of the platforms.

Clearly, the window width 4 has the largest (finite field) arithmetic cost among the width $4, 5$ and 6 but the smallest constant-time table-lookup cost. The 64-bit implementation result of constant-time variable-base scalar multiplication in Table 5.2 shows that for 9-limb reside multiplication the cost of constant-time table-lookup outweighs the lesser arithmetic cost for window width 5 and 6. On the other hand, in the case of constant-time fixed-base scalar multiplication, window width 5 balances well the trade-off between constant-time table-lookup and the arithmetic cost. That's why the optimal result of P-521 and E-521 is with width 5.

Now, we discuss the results of constant-time variable- and fixed-base scalar multi-

plication using our 32-bit residue multiplication. The optimal result of variable-base scalar multiplication is obtained with window width 5 for both P-521 and E-521 as shown in Table 5.2. The results are $3,750,483$ and $3,068,761$ cycles for P-521 and E-521, respectively. In case of fixed-base scalar multiplication, the optimal result of P-521 and E-521 are obtained with window width 6. The Table 5.2 shows the results are $1,130,007$ and $1,169,799$ cycles for P-521 and E-521, respectively. Using low-level programming language implementation for their parallel multiplication and squaring algorithm on Cortex-A15 of the curve P-521, with curve parameter $a \neq -3$, Seo et al. [34] has reported $2,970,976$ and $1,056,902$ cycles using window width-6 for constant-time variable- and fixed-base scalar multiplication, respectively. On the other hand, like 64-bit implementation, all our 32-bit software are purely written in C programming language. Therefore, it is not a fair comparison between our 32-bit results and the results of [34]. Note that the timing results of E-521 is better (less clock cycles) than P-521 and the difference is prominent in variable-base scalar multiplication.

Now, in the case of 32-bit implementation (18-limb), window width 5 yields the optimal result for constant-time variable-base scalar multiplication for P-521 and E-521. Clearly, the cost of constant-time table-lookup outweighs the less (finite field) arithmetic cost using window width 6. However, the result of constant-time fixed-base scalar multiplication shows that it is not always the case. So, this time the arithmetic cost is more significant than the cost of constant-time table-lookup. That's why the number of clock cycles are decreasing from width 4 to 6 for both P-521 and E-521.

CHAPTER 6

CONCLUSION

We have presented a novel way to perform residue multiplication modulo 521-bit Mersenne prime p on 32- and 64-bit platforms such that, to the best of our knowledge, the total arithmetic cost is lower than the existing well-known algorithms. The total arithmetic cost is computed by evaluating the arithmetic cost such that the cost of one single-word multiplication is equal to three single-word additions and one double-word addition is equal to two single-word additions. Our idea is based on Toeplitz Matrix-Vector Product (TMVP) and efficiency is achieved by exploiting the common expressions using the properties of TMVP. The two- and three-way TMVP decomposition are the cornerstones of our work. The reduced-radix representation is used to represent the elements of the field on the underlying computers. We have used our residue multiplication algorithms in the vital operation of single-scalar multiplication for the standard NIST curve P-521 and Edwards curve E-521.

In 64-bit residue multiplication, the field elements are 9-limb long such that each limb is at most 58-bit. The free bit spaces in 64-bit single-word and 128-bit double-word give the advantage of procrastinating the carry propagation, which in turn improves the efficiency especially in the computation of single-scalar multiplication. We have found the total arithmetic cost of our 64-bit residue multiplication to be less than the previously known best algorithm of Granger and Scott [23]. We have also presented three versions of our residue multiplication and found the total arithmetic cost of each version of our residue multiplication to be less than its counterparts. For implementation purpose, we have used both modulus $p = 2^{521} - 1$ and $2p$. We have found the timing results of p better than $2p$ and the difference is more evident especially in the

case of constant-time variable-base scalar multiplication. In the first part of our work, the different versions of our residue multiplication and constant-time variable-base scalar multiplication were implemented in C and C++ programming languages, respectively. In our first testing environment, we couldn't find better timing result than the modular multiplication algorithm in [23] but our scalar multiplication results were better. So, in the second part of our work using a different testing environment, we use the most efficient version — least timing result i.e. Hybrid version — of our 64-bit residue multiplication. In our new timing tests we find 136-cycle which is clearly better than the best timing result of 155-cycle in [23] for the sequential implementation of residue multiplication modulo 521-bit Mersenne prime.

The second part of our work is mainly focused on 32-bit residue multiplication and following our 64-bit residue multiplication, we get TMVP of size 18 on 32-bit platforms where the Toeplitz matrix entries above the diagonal are at most 30-bit. We have proposed three techniques to compute the residue multiplication on 32-bit platforms such the total arithmetic cost is less than the other well-known algorithms. Using the reduced-radix approach for limbs, 64-bit implementation is straigtforward but different challenges arise for 32-bit implementation. Above all, overflow is the main challenge which affects both correctness and efficiency. Therefore, we have shown that our most efficient technique — least timing result i.e. technique-1 — never causes overflow and the compromise on efficiency is kept minimal. In this testing environment, we have found size 6 as the cross-over point between the schoolbook and a variant of TMVP which is required to achieve efficient timing result. Our best timing result for 32-bit residue multiplication is 550-cycle while Seo et al. [34] have reported 350- and 708-cycle on Cortex-A15 and Cortex-A9, respectively, for their parallel multiplication modulo 521-bit Mersenne prime.

Note that all our codes, residue multiplications and single-scalar multiplication, are purely implemented in C programming language without any use of compiler intrinsics and SIMD/assembly instructions in the second part of our work. Using our residue multiplication for 32- and 64-bit platforms, we have implemented constant-time variable- and fixed-base scalar multiplication for the curve P-521 and E-521. To the best of our knowledge, for 64-bit platforms, we have the best timing result for variable-base scalar multiplication for both the curves. The timing results of both the

residue multiplication and the scalar multiplication can further be improved by using SIMD/assembly instructions on 32- and 64-bit platforms. Moreover, in terms of efficiency, we find E-521 to be a better choice than P-521 using our residue multiplication.

REFERENCES

[1] S. Ali and M. Cenk, A new algorithm for residue multiplication modulo $2^{521} - 1$, in S. Hong and J. H. Park, editors, *Information Security and Cryptology - ICISC 2016 - 19th International Conference, Seoul, South Korea, November 30 - December 2, 2016, Revised Selected Papers*, pp. 181–193, 2016.

[2] D. F. Aranha, P. S. L. M. Barreto, C. C. F. P. Geovandro, and J. E. Ricardini, A note on high-security general-purpose elliptic curves, IACR Cryptology ePrint Archive, 2013, p. 647, 2013.

[3] D. J. Bernstein, Curve25519: New diffie-hellman speed records, in M. Yung, Y. Dodis, A. Kiayias, and T. Malkin, editors, *Public Key Cryptography - PKC 2006, 9th International Conference on Theory and Practice of Public-Key Cryptography, New York, NY, USA, April 24-26, 2006, Proceedings*, pp. 207–228, 2006.

[4] D. J. Bernstein, Batch binary edwards, in S. Halevi, editor, *Advances in Cryptology - CRYPTO 2009, 29th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2009. Proceedings*, pp. 317–336, 2009.

[5] D. J. Bernstein, C. Chuengsatiansup, and T. Lange, Curve41417: Karatsuba revisited, in L. Batina and M. Robshaw, editors, *Cryptographic Hardware and Embedded Systems - CHES 2014 - 16th International Workshop, Busan, South Korea, September 23-26, 2014. Proceedings*, pp. 316–334, 2014.

[6] D. J. Bernstein, C. Chuengsatiansup, T. Lange, and P. Schwabe, Kummer strikes back: New DH speed records, in P. Sarkar and T. Iwata, editors, *Advances in Cryptology - ASIACRYPT 2014 - 20th International Conference on the Theory and Application of Cryptology and Information Security, Kaoshiung, Taiwan, R.O.C., December 7-11, 2014. Proceedings, Part I*, pp. 317–337, 2014.

[7] D. J. Bernstein, N. Duif, T. Lange, P. Schwabe, and B. Yang, High-speed high-security signatures, J. Cryptographic Engineering, 2(2), pp. 77–89, 2012.

[8] D. J. Bernstein and T. L. (editors), ebacs: Ecrypt benchmarking of cryptographic systems, accessed: 5 July 2016.

[9] D. J. Bernstein and T. Lange, Explicit formulas database, accessed: 10 March 2017.

[10] D. J. Bernstein and T. Lange, Safecurves: choosing safe curves for elliptic-curve cryptography, accessed: 10 March 2017.

[11] D. J. Bernstein and P. Schwabe, NEON crypto, in E. Prouff and P. Schaumont, editors, *Cryptographic Hardware and Embedded Systems - CHES 2012 - 14th International Workshop, Leuven, Belgium, September 9-12, 2012. Proceedings*, pp. 320–339, 2012.

[12] M. Bodrato, Towards optimal toom-cook multiplication for univariate and multivariate polynomials in characteristic 2 and 0, in *Arithmetic of Finite Fields, First International Workshop, WAIFI 2007, Madrid, Spain, June 21-22, 2007, Proceedings*, pp. 116–133, 2007.

[13] J. W. Bos, C. Costello, H. Hisil, and K. E. Lauter, Fast cryptography in genus 2, in T. Johansson and P. Q. Nguyen, editors, *Advances in Cryptology - EUROCRYPT 2013, 32nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Athens, Greece, May 26-30, 2013. Proceedings*, pp. 194–210, 2013.

[14] 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.

[15] D. R. L. Brown, Sec 2: Recommended elliptic curve domain parameters, certicom research, version 2.0, 27 January 2010.

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

[17] C. Costello, H. Hisil, and B. Smith, Faster compact diffie-hellman: Endomorphisms on the x-line, in P. Q. Nguyen and E. Oswald, editors, *Advances in Cryptology - EUROCRYPT 2014 - 33rd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Copenhagen, Denmark, May 11-15, 2014. Proceedings*, pp. 183–200, 2014.

[18] R. E. Crandall, Method and apparatus for public key exchange in a cryptographic system, October 27 1992, uS Patent 5,159,632.

[19] S. V. Darrel Hankerson, Alfred Menezes, *Guide to Elliptic Curve Cryptography*, Springer New York, 2004, ISBN 978-0-387-95273-4.

[20] W. Diffie and M. E. Hellman, New directions in cryptography, IEEE Trans. Information Theory, 22(6), pp. 644–654, 1976.

[21] H. Fan and M. A. Hasan, A new approach to subquadratic space complexity parallel multipliers for extended binary fields, IEEE Trans. Computers, 56(2), pp. 224–233, 2007.

[22] A. Faz-Hernández, P. Longa, and A. H. Sánchez, Efficient and secure algorithms for glv-based scalar multiplication and their implementation on GLV-GLS curves, in J. Benaloh, editor, *Topics in Cryptology - CT-RSA 2014 - The Cryptographer's Track at the RSA Conference 2014, San Francisco, CA, USA, February 25-28, 2014. Proceedings*, pp. 1–27, 2014.

[23] R. Granger and M. Scott, Faster ECC over $\mathbb{F}_{2^{521}-1}$, in J. Katz, editor, *Public-Key Cryptography - PKC 2015 - 18th IACR International Conference on Practice and Theory in Public-Key Cryptography, Gaithersburg, MD, USA, March 30 - April 1, 2015, Proceedings*, pp. 539–553, 2015.

[24] M. Hamburg, Fast and compact elliptic-curve cryptography, IACR Cryptology ePrint Archive, 2012, p. 309, 2012.

[25] M. Hamburg, Ed448-goldilocks, a new elliptic curve, IACR Cryptology ePrint Archive, 2015, p. 625, 2015.

[26] H. Hisil, K. K. Wong, G. Carter, and E. Dawson, Twisted edwards curves revisited, in J. Pieprzyk, editor, *Advances in Cryptology - ASIACRYPT 2008, 14th International Conference on the Theory and Application of Cryptology and Information Security, Melbourne, Australia, December 7-11, 2008. Proceedings*, pp. 326–343, 2008.

[27] N. I. o. S. Information Technology Laboratory and T. (NIST), Federal information processing standards publication (fips), digital signature standard (dss), july 2013.

[28] A. Karatsuba and Y. Ofman, Multiplication of multidigit numbers on automata, Soviet Physics Doklady, 7, pp. 595–596, January 1963.

[29] N. Koblitz, Elliptic curve cryptosystems, 48, pp. 203–209, 1987.

[30] P. Longa and F. Sica, Four-dimensional gallant-lambert-vanstone scalar multiplication, in X. Wang and K. Sako, editors, *Advances in Cryptology - ASIACRYPT 2012 - 18th International Conference on the Theory and Application of Cryptology and Information Security, Beijing, China, December 2-6, 2012. Proceedings*, pp. 718–739, 2012.

[31] V. S. Miller, Use of elliptic curves in cryptography, in H. C. Williams, editor, *Advances in Cryptology - CRYPTO '85, Santa Barbara, California, USA, August 18-22, 1985, Proceedings*, pp. 417–426, 1985.

[32] G. Paoloni, How to benchmark code execution times on intel ® ia-32 and ia-64 instruction set architectures, september 2010.

[33] M. Scott, Missing a trick: Karatsuba variations, Cryptology ePrint Archive, Report 2015/1247, 2015, `http://eprint.iacr.org/2015/1247`.

[34] H. Seo, Z. Liu, Y. Nogami, T. Park, J. Choi, L. Zhou, and H. Kim, Faster ECC over $\mathbb{F}_{2^{521}-1}$ (feat. NEON), in S. Kwon and A. Yun, editors, *Information Security and Cryptology - ICISC 2015 - 18th International Conference, Seoul, South Korea, November 25-27, 2015, Revised Selected Papers*, pp. 169–181, 2015.

[35] J. A. Solinas, Generalized mersenne number (gmn), 1999, technical Report, National Security Agency, Ft. Meade, MD, USA.

[36] A. Weimerskirch and C. Paar, Generalizations of the karatsuba algorithm for efficient implementations, Cryptology ePrint Archive, Report 2006/224, 2006.

CURRICULUM VITAE

PERSONAL INFORMATION

Surname, Name: Ali, Shoukat

Nationality: Pakistani

Date and Place of Birth: 20 June 1983, Quetta

Marital Status: Single

Phone: 0090 507 831 5617

EDUCATION

Degree	Institution	Year of Graduation
M.S.	University of Management and Technology (UMT), Lahore	2011
B.S.	National University of Computer and Emerging Sciences (NUCES), Karachi	2006
High School	Tameer-e-Nau Public College, Quetta	2001

PROFESSIONAL EXPERIENCE

Year	Place	Enrollment
05/2011-09/2012	University of Lahore	Lecturer
05/2007-07/2008	National Database and Registration Authority (NADRA)	Assistant Manager
08/2008-02/2009	Ministry of Quality Education, Government of Balochistan	Database Administrator

INVITED TALKS

20th Workshop on Elliptic Curve Cryptography (ECC 2016), September 5–7, Yasar University, Izmir, Turkey.

PUBLICATIONS

International Journal Publications

Shoukat Ali and Murat Cenk, “Faster Residue Multiplication Modulo 521-bit Mersenne Prime and an Application to ECC,” submitted to Journal.

International Conference Publications

Shoukat Ali and Murat Cenk, “A new algorithm for residue multiplication modulo $2^{521} - 1$,” in Information Security and Cryptology - ICISC 2016 - 19th International Conference, Seoul, South Korea, November 30 - December 2, 2016, Revised Selected Papers, S. Hong and J. H. Park, Eds., 2016, pp. 181–193.