AN ABSTRACT OF THE THESIS OF

Alper Halbutoğulları for the degree of Doctor of Philosophy
in Electrical & Computer Engineering presented on November 2, 1998.

Title: Fast Bit-Level, Word-Level and Parallel Arithmetic in Finite Fields for Elliptic Curve Cryptosystems

Redacted for privacy

Abstract approved: _____

Çetin K. Koç

Computer and network security has recently become a popular subject due to the explosive growth of the Internet and the migration of commerce practices to the electronic medium. Thus the authenticity and privacy of the information transmitted and the data stored on networked computers is of utmost importance.

The deployment of network security procedures requires the implementation of cryptographic functions. More specifically, these include encryption, decryption, authentication, digital signature algorithms and message-digest functions. Performance has always been the most critical characteristic of a cryptographic function, which determines its effectiveness.

In this thesis, we concentrate on developing high-speed algorithms and architectures for number theoretic cryptosystems. Our work is mainly focused on implementing elliptic curve cryptosystems efficiently, which requires space- and time-efficient implementations of arithmetic operations over finite fields.

We introduce new methods for arithmetic operations over finite fields. Methodologies such as precomputation, residue number system representation, and parallel computation are adopted to obtain efficient algorithms that are applicable on a variety of cryptographic systems and subsystems.

Since arithmetic operations in finite fields also have applications in coding theory and computer algebra, the methods proposed in this thesis are applicable to these applications as well.

# Fast Bit-Level, Word-Level and Parallel Arithmetic in Finite Fields for Elliptic Curve Cryptosystems

by

Alper Halbutoğulları

A THESIS submitted

to

Oregon State University

in partial fulfillment of the

requirements for the degree of

Doctor of Philosophy

Completed November 2, 1998

Commencement June 1999

Doctor of Philosophy thesis of Alper Halbutoğulları presented on November 2, 1998

APPROVED:

Redacted for privacy

Major Professor, representing Electrical & Computer Engineering

Redacted for privacy

Chair of Electrical & Computer Engineering Department

Redacted for privacy

Dean of Graduate School

I understand that my thesis will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my thesis to any reader upon request.

Redacted for privacy

Alper Halbutoğulları, Author

# ACKNOWLEDGMENTS

I thank my advisor Çetin K. Koç for his reviews and help in producing the papers we published, which formed a basis for this thesis.

I also thank my committee members for their efforts and patience in going through my work in a very short time.

Finally, I acknowledge financial support for this work from Intel Corporation.

Alper Halbutoğulları

Corvallis, Oregon

# TABLE OF CONTENTS

## TABLE OF CONTENTS (CONTINUED)

## TABLE OF CONTENTS (CONTINUED)

## TABLE OF CONTENTS (CONTINUED)

## TABLE OF CONTENTS (CONTINUED)

## LIST OF FIGURES

## LIST OF TABLES

To my family...

for their endless love and support

# Fast Bit-Level, Word-Level and Parallel Arithmetic in Finite Fields for Elliptic Curve Cryptosystems

## Chapter 1
## INTRODUCTION

Computer and network security has recently become a popular subject due to the explosive growth of the Internet and the migration of commerce practices to the electronic medium. We started to buy goods online, bank online, conduct business and financial transactions and send messages world-wide. Thus, the authenticity and privacy of the information transmitted and the data stored on networked computers is of utmost importance.

The deployment of network security procedures requires the implementation of cryptographic functions. More specifically, these include encryption, decryption, authentication, digital signature algorithms and message-digest functions. Software implementations of these algorithms are often desired because of their flexibility and cost effectiveness. However, performance has always been the most critical characteristic of a cryptographic function that determines its effectiveness, requiring the algorithm engineer to invent and develop new methods for high-speed implementations.

In this thesis, we concentrate on developing high-speed algorithms and architectures for number theoretic cryptosystems. Our work is mainly focused on implementing elliptic curve cryptosystems efficiently, which requires space- and time-efficient implementations of arithmetic operations over finite fields. Since longer bit lengths are required for better security, efficient arithmetic over finite fields with high order will serve our purpose.

We introduce new methods for arithmetic operations over finite fields. Methodologies such as precomputation, residue number system representation, and parallel

computation are adopted to obtain efficient algorithms that are applicable on a variety of cryptographic systems and subsystems.

We choose to work on finite fields of characteristic two that has high order, mainly because they are easier to implement in hardware and software, and longer bit lengths provide better security.

We basically try to invent and improve algorithms to manage fast arithmetic operations in finite fields of characteristic two, i.e., Galois field $GF(2^k)$. Some of the algorithms can also be used in finite fields with different characteristics. We prefer polynomial representation, because combined with characteristic two, addition and subtraction then become equivalent and can easily be performed on the corresponding terms of polynomials using bit-wise `XOR` operations. Since the efficiency of exponentiation inherently depends on the efficiency of multiplication, the main issue is then to manage multiplication efficiently, which is usually followed by a reduction. Therefore, in this thesis, we mainly focused on multiplication algorithms and architectures.

Since arithmetic operations in finite fields also have applications in coding theory and computer algebra, the methods proposed in this thesis are applicable to these applications as well.

Chapters 2 through 8 provides the background necessary to understand the motivation and the contents of the contributions made, which are presented in Chapters 9 through 12. A brief description of each chapter is as follows:

Chapter 2 provides a brief introduction to cryptography and to the terminology commonly used. It also explores different types of cryptographic systems commonly used, and summarizes the basic ideas on which these systems are built. Strong and weak points of the systems are also presented to able to compare the level of security they provide.

Chapter 3 includes the definitions related to the elliptic curves. It mainly describes how a set of points on an elliptic curve form an Abelian group, and how to manage the group operations on this set. The general group structure, the procedures to find the order of the curve, and the theorems about the upper and lower bounds

of the order are also presented in this chapter. Finally elliptic curves constructed over rings, along with the cryptosystems they yield, are discussed.

Chapter 4 explains how the elliptic curves are classified according to the characteristics of the underlying finite field and the security they provide. Simplified addition formulas are derived for each case. Using the relations between the order and the structure of the Abelian groups, it was possible to classify all the elliptic curves into a small number of isomorphism classes, which are then listed in a table, with the structure and the order found for each class.

In Chapter 5, we present the discrete logarithm problem in finite fields and described the algorithms found to date, to solve it. Then the elliptic curve logarithm problem is presented and it is compared to the discrete logarithm problem. It is also shown that for some classes of elliptic curves, the elliptic curve logarithm problem reduces to the discrete logarithm problem in a finite field, and hence these curves provide no extra security. The results and the reduction parameters for each class are summarized in a table.

In Chapter 6, the implementation issues such as which curve to select, how to count the order of a curve and how to implement an elliptic curve cryptosystem are discussed. Results of some previous implementations are summarized. We also present an algorithm to find the order, which we built using Schoof's original paper.

Chapter 7 summarizes other uses of the elliptic curves, such as primality testing and factorization. The known primality testing methods are presented and compared to that of elliptic curves'.

Chapter 8 provides an introduction to arithmetic operations in finite fields. It presents different notations and technics used to represent field elements and manage field arithmetic. The algorithms and methods presented in later chapters assume the notations and technics presented in this chapter.

In Chapter 9, a new table lookup based reduction method for performing the modular reduction operation is proposed. This method can be used to obtain fast software implementations of the finite field multiplication and squaring operations. The reduction algorithm has both left-to-right and right-to-left versions, which re-

spectively improve the standard and Montgomery multiplication methods. Furthermore, it is shown that the right-to-left version of the proposed reduction method also works in the integer case, if the modulus $n$ is an odd number.

In Chapter 10, a novel method of parallelization of the multiplication operation in $GF(2^k)$ is presented for an arbitrary value of $k$ and arbitrary irreducible polynomial $n(x)$ generating the field. The parallel algorithm is based on the polynomial residue number system. The parallel algorithm receives the residue representations of the input operands (elements of the field) and produces the result in its residue form, however, it is guaranteed that the degree of this polynomial is less than $k$, i.e., it is an element of the field, properly reduced by the irreducible polynomial $n(x)$.

In Chapter 11, we propose a new formulation of the multiplication matrix and an architecture for the multiplication operation in $GF(2^m)$. The proposed architecture generalizes the Mastrovito multiplication, and is particularly efficient when applied to a specific class of polynomials that is known in advance, as it uses all possible optimizations. We have studied all such cases in detail, and obtained space and time complexities, and furthermore, provided actual design examples.

Chapter 12 concludes the thesis with the summary of the results, contributions and discussions.

Appendix A includes the list of notations and definitions used in the thesis.

In Appendix B and Appendix C, we provide some of the algorithms and theorems referred or used during the presentation of methods throughout the thesis.

Appendix D includes some short notes on finite fields that can be used to refresh the knowledge on finite fields, before reading Chapter 8.

Finally, we also provide a rich index for the convenience of the reader.

# Chapter 2
# CRYPTOGRAPHIC SYSTEMS

## 2.1 Introduction

The fundamental goal of cryptography has historically been to achieve privacy, i.e., to enable two people to send each other messages over an insecure channel in such a way that only the intended recipient can read (or understand) the message. This objective has traditionally been met by using private key cryptosystems. As will be explained in the following sections this system has some disadvantages that make it unsuitable for use in certain applications. Public key cryptosystems overcome the key distribution and management problems inherent with private key systems. And elliptic curves offer more security using smaller bandwidth, which make them important.

## 2.2 Private Key Cryptography

Let $\mathcal{M}$ denote the set of all possible plaintext messages, $\mathcal{C}$ the set of all possible ciphertext messages and $\mathcal{K}$ the set of all possible keys.

A *private key cryptosystem* consists of a family of pairs of functions

$$E_k : \mathcal{M} \longrightarrow \mathcal{C} \quad \text{and} \quad D_k : \mathcal{C} \longrightarrow \mathcal{M} \qquad k \in \mathcal{K} \ ,$$

such that

$$D_k(E_k(m)) = m \qquad \text{for all } m \in \mathcal{M} \text{ and } k \in \mathcal{K} \ .$$

To use such a system, the two users must initially agree upon a secret key $k \in \mathcal{K}$. They can do this by physically meeting, but this can be impractical or sometimes even impossible, or they might use the services of a trusted courier. Besides the problem of availability, the secureness of the courier is still questionable.

The main disadvantages of the private key cryptosystems are:

1. Key distribution problem (a secure channel may not be available)
2. Key management problem (if the number of pairs (of users) is large then the number of keys becomes unmanageable)
3. No signatures possible

A *digital signature* is an electronic analogue of a hand-written signature that allows a receiver to convince a third party that the message is in fact originated from the sender.

In *one-time pad* method, the keys are random binary strings and a message is encrypted by `XOR`'ing the key to it, one bit at a time. This system is secure but requires a key as long as the message itself.

## 2.3 Public Key Cryptography

In 1976, W. Diffie and M. Hellman invented public key cryptography to address the deficiencies in private key cryptography, stated in the previous section [10]. Their protocol is known as *Diffie-Hellman key exchange.* And in terms of an arbitrary group it can be described as:

1. (Setup) $A$ and $B$ publicly select a (multiplicatively written) finite group $G$ and an element $\alpha \in G$.
2. $A$ generates a random integer $a$, computes $\alpha^a$ in $G$, and transmits $\alpha^a$ to $B$ over a public communications channel.
3. $B$ generates a random integer $b$, computes $\alpha^b$ in $G$, and transmits $\alpha^b$ to $A$ over a public communications channel.
4. $A$ receives $\alpha^b$ and computes $(\alpha^b)^a$.
5. $B$ receives $\alpha^a$ and computes $(\alpha^a)^b$.

$A$ and $B$ now share the common group element $\alpha^{ab}$. Note that an eavesdropper knows $G$,$\alpha$,$\alpha^a$ and $\alpha^b$, and his task is to use this information to reconstruct $\alpha^{ab}$. This problem is commonly referred to as *Diffie-Hellman problem.*

The problem of computing $a$, given $G$, $\alpha$ and $\alpha^a$ is called the *discrete logarithm problem.* It has not been proven but widely believed that the discrete logarithm problem and the Diffie-Hellman problem are computationally equivalent [46].

### 2.3.1 Trapdoor One-way Functions

*Easy-hard* : We will use the term *hard* to mean computationally infeasible, i.e., infeasible using the best known algorithms and best available technology. In terms of software engineering only, *easy* and *hard* will mean requiring polynomial and exponential time, respectively.

*One-way function* : An invertible function $f : \mathcal{M} \longrightarrow \mathcal{C}$, such that $\forall m \in \mathcal{M}$ it is *easy* to compute $f(m)$, but for most $c \in \mathcal{C}$ it is *hard* to compute $f^{-1}(m)$. There is no function that is proven to be a one-way function, but there are some candidates.

*Trapdoor one-way function* : A one-way function that can be efficiently inverted by using some extra information. The extra information is called the *trapdoor.*

A public key cryptosystem is constructed using a family $f_k : \mathcal{M} \longrightarrow \mathcal{C}$, $k \in \mathcal{K}$, of trapdoor one-way functions. The trapdoors $t(k)$ should be easy to obtain for all $k \in \mathcal{K}$. Also an efficient algorithm is needed to find $f_k$, but knowing it, should not lead one to any information that will make it feasible to recover $k$ (and thus $t(k)$). If these conditions are satisfied then each user $U$ selects a random key $u \in \mathcal{K}$ and publishes the algorithm $A_u$, his *public key*, for computing $f_u$. Then the trapdoor $t(u)$, which is used to invert $f_u$, will be the user's *private key.* Any message $m$ is encrypted to $f_u(m)$ using $A_u$, and only the user $U$ can find out $m$ from $f_u(m)$, as only *he* knows $t(u)$ to invert $f_u$. To use digital signatures we assume $\mathcal{M} = \mathcal{C}$ [46].

The group order and exponentiation are two commonly used trapdoor one-way functions. RSA and elliptic curves over the ring $\mathcal{Z}_n$ uses group order as a trapdoor one-way function, and ElGamal Cryptosystem uses exponentiation. Now we will further explain these systems.

Suppose that for the group $G$, multiplication is easy and finding its order is hard. Then we can construct a public key cryptosystem whose trapdoor one-way function is based on the difficulty of finding group order. Each user chooses a group $G$ (with order n) and a random integer $e$ such that $\gcd(e, n) = 1$ and computes (using the extended Euclidean Algorithm) an integer $d$, $1 \leq d \leq n-1$, such that

$$ed \cong 1 \pmod{n} .$$

$A$'s public key consists of the group $G$ and the integer $e$. To send the message $m$ to $A$, one computes and sends $m^e$. $A$ can recover $m$ using $d$ since $(m^e)^d = m$.

Exponentiation in a multiplicatively written finite group can be performed efficiently by repeated square-and-multiply method, if an efficient way to compute the product is known. Similarly, multiplication in an additively written finite group can be performed efficiently by repeated double-and-add method, if an efficient way to compute the addition is known.

### 2.3.2 RSA Cryptosystem

The RSA cryptosystem was invented in 1977 by Rivest, Shamir and Adleman [60] and was the first realization of Diffie-Hellman's abstract model for public key cryptography.

To set up this system each user picks two large primes $p$ and $q$ and computes their product $n = pq$. The group used is $G = \mathcal{Z}_n^*$. It is well known that the order of $G$ is

$$\phi(n) = (p-1)(q-1) .$$

The public key is the pair of integers $(n, e)$ and the private key is $d$. The problem of computing $\phi(n)$ using only $n$ is computationally equivalent to the problem of

factoring $n$, which is believed to be hard (but not proven). Thus the security of the RSA is based on the factoring problem.

### 2.3.3 ElGamal Cryptosystem

In 1985, T. ElGamal [11] proposed a public key scheme based on discrete exponentiation, which exhibits the properties of a trapdoor one-way function.

1. (Setup) A finite group $G$ and element $\alpha \in G$ are chosen. Each user picks a random integer $l$ (the private key) and makes public $\alpha^l$ (the public key). We suppose that messages are elements of $G$ and that user $A$ wishes to send a message $m$ to user $B$.
2. $A$ generates a random integer $k$ and computes $\alpha^k$.
3. $A$ looks up $B$'s public key $\alpha^b$, and computes $(\alpha^b)^k$ and $m\alpha^{bk}$.
4. $A$ sends to $B$ the pair of group elements $(\alpha^k, m\alpha^{bk})$.
5. $B$ computes $(\alpha^k)^b$ and uses this to recover m.

Clearly, the security of the ElGamal cryptosystem and the Diffie-Hellman key exchange are equivalent, and hence the security of the ElGamal cryptosystem is based on the difficulty of the discrete logarithm problem. ElGamal [11] also designed a signature scheme, which makes use of the group $G$ [46].

### 2.3.4 Elliptic Curve Cryptosystems

When an elliptic curve is defined over a finite field, the points on the curve form an Abelian group. The addition operation in this group is "easy" to implement, both in hardware and software. The discrete logarithm problem in this group is believed (but not proven) to be very "hard", in particular harder than the one defined in finite fields of the same size. It was for this reason that the elliptic curves were first suggested in 1985 by N. Koblitz [27] and V. Miller [48] for implementing public key cryptosystems.

Elliptic curves over finite fields can be used to implement the Diffie-Hellman key exchange, and the ElGamal cryptosystem. Also they can replace trapdoor one-way functions in any system. These systems potentially provide the same security as the existing public key cryptosystems, but uses shorter key lengths. Having shorter key lengths mean smaller bandwidth and memory requirements, which is a crucial factor in some applications such as design of smart cards, where both memory and processing power is limited. Another advantage of using the elliptic curves is that each user may select a different curve, even though the underlying field is the same for all. That means each user can change his curve periodically (for extra security) without changing the hardware [46].

# Chapter 3
# ELLIPTIC CURVES

## 3.1 Basic Definitions

Elliptic curves can be defined over any field (e.g. real, complex, finite), but for cryptographic purposes we will only be concerned with those over finite fields [29]. Let $H$ be a field and $\overline{H}$ be its *algebraic closure* (*see* Appendix A). Then an *Elliptic Curve* $E$ is the set of solutions of the (Weierstrass) equation

$$y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6 \ , \tag{3.1}$$

in the affine plane $\overline{H} \times \overline{H}$, together with the *point at infinity* $\mathcal{O}$.

If $a_1, a_2, a_3, a_4, a_6 \in H$ , then $E$ is said to be defined over $H$ and denoted by $E/H$. The set of *H-rational points of* $E$, denoted $E(H)$, is the set of points both of whose coordinates lie in $H$, including the point $\mathcal{O}$.

## 3.2 Group Law

The points on an elliptic curve form an Abelian group under a certain addition, with identity $\mathcal{O}$. Negative of a point $P$ is denoted by $-P$ and is defined to be the point with the same x-coordinate. First we will define the operations intuitively: If a line intersects an elliptic curve at more than one point, then it intersects it at exactly 3 points, counting the multiplicities (When there are less than 3 finite points, it is assumed that the rest of the intersections are at infinity, i.e., at the point $\mathcal{O}$) [26]. Then the sum of any two points is defined to be the negative of the third intersection of the elliptic curve with the line through these two points. For a nice geometrical introduction to elliptic curves see [66].

Now we can explicitly state the addition rules for all $P$,$Q \in E$ as follows:

1. $\mathcal{O} + P = P$ and $P + \mathcal{O} = P$ ($\mathcal{O}$ is the identity element).

2. $-\mathcal{O} = \mathcal{O}$ .

3. If $P = (x_1, y_1) \neq \mathcal{O}$, then $-P = (x_1, -y_1 - a_1 x_1 - a_3)$.

4. If $Q = -P$, then $P + Q = \mathcal{O}$.

5. If $P \neq \mathcal{O}$, $Q \neq \mathcal{O}$, $Q \neq -P$, then let $R$ be the third point of intersection (counting multiplicities) of either the line $\overline{PQ}$ if $P \neq Q$, or the tangent line to the curve at $P$ if $P = Q$, with the curve. Then $P + Q = -R$.

   If $P = (x_1, y_1)$, $Q = (x_2, y_2)$ and $R = (x_3, y_3)$ then :

$$x_3 = \lambda^2 + a_1\lambda - a_2 - x_1 - x_2$$

   and

$$y_3 = -(\lambda + a_1)x_3 - \beta - a_3 \ ,$$

   where

$$\lambda \ = \ \begin{cases} \frac{y_2 - y_1}{x_2 - x_1} & \text{if } P \neq Q \ , \\ \frac{3x_1^2 + 2a_2x_1 + a_4 - a_1y_1}{2y_1 + a_1x_1 + a_3} & \text{if } P = Q \ . \end{cases}$$

**Theorem 3.1 ([46])** *$(E, +)$ is an Abelian group with the identity element $\mathcal{O}$. If $E$ is defined over $H$, then $E(H)$ is a subgroup of $E$.*

## 3.3 Group Structure

There are many analogies between the group of $F_q$-rational points on an elliptic curve and the multiplicative group $F_q^*$: approximately same number of elements, etc. But for a single (large) $q$ there are many different elliptic curves and many different $N$ to choose from. Hence elliptic curves offer a rich source of "naturally occurring" finite Abelian groups. It is this major advantage that makes elliptic curves attractive for cryptography [26].

Let $E$ be an elliptic curve defined over $F_q$. Let $q = p^k$, where $p$ (a prime) is the characteristic of $F_q$. The number of points in $E(F_q)$ is called the *order of the curve* $E(F_q)$ and denoted by $\#E(F_q)$. Then we have:

**Theorem 3.2 (Hasse,[65])** *Let* $\#E(F_q) = q + 1 - t$. *Then* $|t| \leq 2\sqrt{q}$.

An important consequence of Hasse's Theorem is that we can pick points $P$ uniformly and randomly on an elliptic curve $E(F_q)$ in probabilistic polynomial time.

The next result, proven by Waterhouse, determines the possible values for the order $\#E(F_q)$ as $E$ varies over all elliptic curves defined over $F_q$, where $q = p^k$.

**Theorem 3.3 ([46])** *There exists an elliptic curve* $E/F_q$ *such that* $E(F_q)$ *has an order* $q + 1 - t$ *over* $F_q$ *if and only if one of the following conditions holds:*

*(i)* $t \neq 0 \pmod{p}$ *and* $t^2 \leq 4q$.

*(ii)* $k$ *is odd and one of the followings holds:*

1. $t = 0$.
2. $t^2 = 2q$ *and* $p = 2$.
3. $t^2 = 3q$ *and* $p = 3$.

*(iii)* $k$ *even and one of the followings holds:*

1. $t^2 = 4q$.
2. $t^2 = q$ *and* $p \not\equiv 1 \pmod{3}$.
3. $t = 0$ *and* $p \not\equiv 1 \pmod{4}$.

The following theorem is due to Deuring:

**Theorem 3.4 ([59])** *Every integer in the interval* $(p + 1 - 2\sqrt{p}, p + 1 + 2\sqrt{p})$ *is attained as* $\#E(F_p)$ *for some choice of* $(a, b) \in F_p \times F_p$ *with* $4a^3 + 27b^2 \neq 0$.

As, $E$ varies over all elliptic curves over $F_q$, the values $\#E(F_q)$ are nearly uniformly distributed in the interval of size $\sqrt{p}$ centered at $p+1$. This statement is made precise by the following theorem, which was a key ingredient in Lenstra's elliptic curve algorithm for factoring integers (*see* Appendix B.5) [36].

**Theorem 3.5 ([46, 35])** *There exist positive effectively computable constants $c_1$ and $c_2$ such that for each prime $p \geq 5$ and for any subset $S$ of integers in the interval $[p+1-\sqrt{p}, p+1+\sqrt{p}]$, the probability $r_s$ of a random pair $(a,b) \in F_p \times F_p$ defining an elliptic curve $E : y^2 = x^3 + a_4x + a_5$ with $\#E(F_q) \in S$ is bounded as follows:*

$$\frac{\#S-2}{2\lfloor\sqrt{p}\rfloor+1} \cdot c_1(\log p)^{-1} \leq r_s \leq \frac{\#S}{2\lfloor\sqrt{p}\rfloor+1} \cdot c_2(\log p)(\log\log p)^2 \ .$$

For the next theorem we will need some standard results from Abelian group theory. Every finite Abelian group $G$ can be decomposed into a direct sum of cyclic groups

$$G = \mathcal{Z}_{n_1} \oplus \mathcal{Z}_{n_2} \oplus \cdots \oplus \mathcal{Z}_{n_s} \ ,$$

where $n_{i+1}|n_i$ for all $i = 1, 2, \ldots, s-1$, and $n_s \geq 2$. Furthermore this decomposition is unique. We say that $G$ is an Abelian group of *type* $(n_1, n_2, \ldots, n_s)$ and *rank* $s$.

**Theorem 3.6 ([46])** *$E(F_q)$ is an Abelian group of rank 1 or 2. The type of the group is $(n_1, n_2)$, i.e., $E(F_q) \cong \mathcal{Z}_{n_1} \oplus \mathcal{Z}_{n_2}$, where $n_2|n_1$, and furthermore $n_2|q-1$.*

The curve $E$ can also be viewed as an elliptic curve over any extension field $L = F_{q^m}$ of $F_q$; $E(F_q)$ is a subgroup of $E(L)$. The Weil Theorem enables one to compute $\#E(F_{q^m})$, for $m \geq 2$, from $\#E(F_q)$ as follows:

**Theorem 3.7 (Weil,[46])** *Let $E$ be an elliptic curve defined over $F_q$, and let $t = q+1-\#E(F_q)$. Then*

$$\#E(F_{q^m}) = q^m + 1 - \alpha^m - \beta^m \ ,$$

*where $\alpha, \beta$ are complex numbers determined from the factorization of*

$$1 - tT + qT^2 = (1-\alpha T)(1-\beta T) \ .$$

Now we state a few results on the group structure of $E = E(\overline{F_q})$. $E$ is a torsion group, i.e., for each point $P \in E$ there is a positive integer $k$ such that $kP = \mathcal{O}$. The smallest such integer is called the *order of the point* $P$. An *n-torsion* point is a point $P \in E(\overline{F_q})$ satisfying $nP = \mathcal{O}$. Let $E(F_q)[n]$ denote the subgroup of $n$-torsion points in $E(F_q)$, where $n \neq 0$. We will write $E[n]$ for $E(\overline{F_q})[n]$. In the algebraic number theory of elliptic curves, one finds a deep analogy between the coordinates of the $n$-torsion points on an elliptic curve and the $n$-division points on the unit circle (which are the $n$th roots of unity in the complex plane [29]). If $n$ and $q$ are relatively prime, then $E[n] \cong \mathcal{Z}_n \oplus \mathcal{Z}_n$. *Division polynomials* are recursively defined set of polynomials that helps one to identify the $n$-torsion points or to compute $nP$ using $P$ [46]. As will be seen later in Section 6.4, they are basically used in determining the order of a curve.

## 3.4 Elliptic Curves Over Rings

Elliptic curves over the ring $\mathcal{Z}_n$ are used in Lenstra's integer factoring algorithm (*see* Appendix B.5) [36] and the Goldwasser-Kilian primality proving algorithm [15].

Let $n$ be a positive integer with $\gcd(n, 6) = 1$. An elliptic curve over $\mathcal{Z}_n$ is given by an equation

$$E_{a,b} : y^2 = x^3 + ax + b \ ,$$

where $a, b \in \mathcal{Z}$ and $\gcd(4a^3 + 27b^2) = 1$. If $n$ is a product of 2 primes $p$ and $q$, then points on

$$\tilde{E}_{a,b}(\mathcal{Z}_n) = E_{a,b}(F_p) \times E_{a,b}(F_q)$$

can be computed using $E_{a,b}(\mathcal{Z}_n)$, i.e., without knowing $p$ & $q$, and any failure leads to the factorization $n = pq$ [46].

A cryptosystem can be proposed based on the above ideas (See [46] for a description). Like the RSA the security of that system is based on the difficulty of factoring $n$, however it is not known whether breaking the system is equivalent to factoring $n$. Although the system is not as efficient as RSA, it has the advantage that it appears to be resistant to some of the known attacks on RSA. See [33] for more

details. In [52], Okamoto, Fujioka and Fujisaki propose a practical digital signature scheme based on elliptic curves over $\mathcal{Z}_n$, where $n = p^2q$. The scheme appears to be several times faster than the RSA signature scheme.

# Chapter 4
# CLASSIFICATION & ANALYSIS OF ELLIPTIC CURVES

## 4.1 Isomorphic Curves

Two elliptic curves are said to be *isomorphic* if they are isomorphic as projective varieties. Two projective varieties $V_1$ and $V_2$ defined over a field $H$, are isomorphic over $H$, if there exists morphisms $\phi$ and $\psi$ defined over $H$, such that

$$\phi : V_1 \longrightarrow V_2 \quad , \qquad \psi : V_2 \longrightarrow V_1 \ ,$$

and both $\phi \circ \psi$ and $\psi \circ \phi$ are identity maps on $V_1$ and $V_2$, respectively.

**Theorem 4.1 ([46])** *Two elliptic curves, $E_1/H$ and $E_2/H$, given by the equations*

$$\begin{aligned} E_1 : y^2 + a_1xy + a_3y &= x^3 + a_2x^2 + a_4x + a_6 \\ E_2 : y^2 + \bar{a}_1xy + \bar{a}_3y &= x^3 + \bar{2}_3x^2 + \bar{a}_4x + \bar{a}_6 \end{aligned}$$

*are isomorphic over $H$, denoted by $E_1/H \cong E_2/H$, if and only if there exist $u, r, s, t \in H$, $u \neq 0$, such that the change of variables*

$$(x, y) \longrightarrow (u^2x + r, u^3y + u^2sx + t)$$

*transforms equation $E_1$ into equation $E_2$. Furthermore when $E_1/H \cong E_2/H$ holds, $E_1/H$ and $E_2/H$ are also isomorphic as Abelian groups. The relationship of isomorphism is an equivalence relation.*

The change of variables in above theorem is referred to as an *admissible change of variables.*

**Table 4.1.** EC Equations and Addition Formulas for Char$(H) = 2$

| Elliptic Curve Equation | $P \stackrel{?}{=} Q$ | Coordinates |
|---|---|---|
| $y^2 + xy = x^3 + a_2x^2 + a_6$ $(a_1 \neq 0)$ | $P \neq Q$ | $x_3 = \left(\frac{y_1+y_2}{x_1+x_2}\right)^2 + \frac{y_1+y_2}{x_1+x_2} + x_1 + x_2 + a_2$ |
| | | $y_3 = \left(\frac{y_1+y_2}{x_1+x_2}\right)(x_1 + x_3) + x_3 + y_1$ |
| | $P = Q$ | $x_3 = x_1^2 + \frac{a_6}{x_1^2}$ |
| | | $y_3 = x_1^2 + (x_1 + 1 + \frac{y_1}{x_1})x_3$ |
| $y^2 + a_3y = x^3 + a_4x + a_6$ $(a_1 = 0)$ | $P \neq Q$ | $x_3 = \left(\frac{y_1+y_2}{x_1+x_2}\right)^2 + x_1 + x_2$ |
| | | $y_3 = \left(\frac{y_1+y_2}{x_1+x_2}\right)(x_1 + x_3) + a_3 + y_1$ |
| | $P = Q$ | $x_3 = \frac{x_1^4+a_4^2}{a_3^2}$ |
| | | $y_3 = \left(\frac{x_1^2+a_4}{a_3}\right)(x_1 + x_3) + a_3 + y_1$ |

## 4.2 Curves over Fields with Different Characteristics

Let

$$E_2 : y^2 + \bar{a}_1xy + \bar{a}_3y = x^3 + \bar{a}_2x^2 + \bar{a}_4x + \bar{a}_6$$

be the general elliptic curve equation. When we specify the characteristic of the underlying field, this equation transforms into simpler forms that are isomorphic to it. For example for Char$(H) = 3$, it simplifies to:

$$y^2 = x^3 + a_2x^2 + a_4x + a_6$$

Table 4.1 and Table 4.2 summarize the cases for Char$(H) = 2$ and Char$(H) \neq 2, 3$ respectively, along with the coordinates of the sum $P + Q = (x_3, y_3)$ of two points $P = (x_1, y_1)$ and $Q = (x_2, y_2)$ on the given curve [46].

Elliptic curves over finite fields with Char$(H) = 2$, i.e., GF$(2^k)$, is particularly interesting. The arithmetic operations for the underlying field are easy to construct and relatively simple to implement. These systems have shown the potential to provide small and low-cost public key cryptosystems. Therefore, we will concentrate on these fields.

**Table 4.2.** EC Equations and Addition Formulas for Char$(H) \neq 2, 3$

| Elliptic Curve Equation | $P \stackrel{?}{=} Q$ | $x_3$ | $y_3$ |
|---|---|---|---|
| $y^2 = x^3 + a_4x + a_6$ | $P \neq Q$ | $\left(\frac{y_2-y_1}{x_2-x_1}\right)^2 - x_1 - x_2$ | $\left(\frac{y_2-y_1}{x_2-x_1}\right)(x_1 - x_3) - y_1$ |
| | $P = Q$ | $\left(\frac{3x_1^2+a_4}{2y_1}\right)^2 - x_1 - x_2$ | $\left(\frac{3x_1^2+a_4}{2y_1}\right)(x_1 - x_3) - y_1$ |

## 4.3 Singular vs. Non-singular Elliptic Curves

Let $F(x, y) = 0$ be the implicit equation for $y$ as a function of $x$ in Equation (3.1). A point on the curve is said to be *non-singular* (or a *smooth* point) if at least one of the partial derivatives $\partial F/\partial x, \partial F/\partial y$ is nonzero at that point. If both of the partial derivatives are zero, then the point is called *singular*. If a curve has no singular points, then it is called *non-singular elliptic curve*, otherwise it is called a *singular elliptic curve*. The condition for the cubic on the right of the elliptic curve equations in Table 4.1 & 4.2, not to have multiple roots, is equivalent to require all points on the curve to be non-singular [29].

## 4.4 The Discriminant and $j$-Invariant

Let $E$ be the elliptic curve given by the Equation (3.1). Define the following quantities:

$$\begin{aligned}
d_2 &= a_1^2 + 4a_2 \\
d_4 &= 2a_4 + a_1a_3 \\
d_6 &= a_3^2 + 4a_6 \\
d_8 &= a_1^2a_6 + 4a_2a_6 - a_1a_3a_4 + a_2a_3^2 - a_4^2 \\
c_4 &= d_2^2 - 24d_4 \\
\Delta &= -d_2^2d_8 - 8d_4^3 - 27d_6^2 + 9d_2d_4d_6 \\
j(E) &= c_4^3/\Delta
\end{aligned}$$

The quantity $\Delta$ is called the *discriminant* of the elliptic curve equation, while $j(E)$ is called the *j-invariant* of $E$ if $\Delta \neq 0$.

**Theorem 4.2 ([46])** *$E$ is an elliptic curve, i.e., the Elliptic Curve Equation (3.1) is non-singular, if and only if $\Delta \neq 0$.*

**Theorem 4.3 ([46])** *If two elliptic curves $E_1/H$ and $E_2/H$ are isomorphic over $H$, then $j(E_1) = j(E_2)$. The converse is also true if $H$ is an algebraically closed field.*

## 4.5 Supersingular vs. Non-supersingular Elliptic Curves

The elliptic curve $E$ is said to be *supersingular* if $p$ divides $t$, where $\#E(F_q) = q+1-t$. Otherwise it is called *non-supersingular*. It is well-known that if $p = 2$ or if $p = 3$, then $E$ is supersingular if and only if $j(E) = 0$. For Char$(H) = 2$, $j$-invariant simplifies to $j(E) = a_1^{12}/\Delta$, so $j(E) = 0$ if and only if $a_1 = 0$. Thus curves in the first row of the Table 4.1 are non-supersingular and the ones in the second row are supersingular. In Theorem 3.3, case (i) shows the non-supersingular curves and cases (ii) & (iii) show the supersingular ones. From that theorem we can easily deduce the following:

**Corollary 4.4 ([46])** *Let $E$ be defined over $F_q$. Then $E$ is supersingular if and only if $t^2 = 0, q, 2q, 3q, 4q$.*

If $E$ is a supersingular curve, then the group structure of $E(F_q)$ is determined by the next result.

**Lemma 4.5 ([63])** *Let $\#E(F_q) = q + 1 - t$.*

*(i) If $t^2 = q, 2q, 3q$, then $E(F_q)$ is cyclic.*

*(ii) If $t^2 = 4q$, then* $$E(F_q) \cong \begin{cases} \mathcal{Z}_{\sqrt{q}-1} \oplus \mathcal{Z}_{\sqrt{q}-1} & \text{if } t = 2\sqrt{q} \\ \mathcal{Z}_{\sqrt{q}+1} \oplus \mathcal{Z}_{\sqrt{q}+1} & \text{if } t = -2\sqrt{q} \end{cases}$$

*(iii) If $t = 0$, then* $$E(F_q) \cong \begin{cases} \text{cyclic} & \text{if } q \not\equiv 3 \pmod 4 \\ \mathcal{Z}_{(q+1)/2} \oplus \mathcal{Z}_2 \text{ or cyclic} & \text{if } q \equiv 3 \pmod 4 \end{cases}$$

We can deduce the following Theorem immediately from Theorem 3.6, which determines all possible groups $E(F_q)$ that occur as $E$ varies over all non-supersingular curves defined over $F_q$.

**Theorem 4.6 ([61, 70])** *If $r$ is a prime, then let $v_r(n)$ be the largest integer with $r^{v_r(n)}|n$. Let $N = \#E(F_q) = q + 1 - t$, where $t \neq 0 \pmod p$ and $t^2 \leq 4q$. If $a_r, b_r$ are integers which satisfy $a_r \geq b_r, a_r + b_r = v_r(N)$, and $b_r \leq v_r(q-1)$ for each prime $r \neq p$, then there exists a non-supersingular curve $E$ defined over $F_q$ such that $E(F_q)$ has group structure*

$$\mathcal{Z}/p^{v_p(N)} \oplus \bigoplus_{r \neq p} (\mathcal{Z}/r^{a_r} \oplus \mathcal{Z}/r^{a_r}) \quad .$$

*Also, if $N = \#E(F_q)$ factors as a product of distinct primes, then $E(F_q)$ is cyclic.*

If $n = p^e$, then either $E[p^e] \cong \{\mathcal{O}\}$ if $E$ is supersingular, or else $E[p^e] \cong \mathcal{Z}_{p^e}$ if $E$ is non-supersingular.

## 4.6 Isomorphism Classes of Elliptic Curves

In this section, the isomorphism classes of elliptic curves over finite fields $H$ will be counted. For the case $H = F_{2^k}$, a representative from each class will be listed. $\#E(F_{2^k})$ for each supersingular curve $E$ defined over $F_{2^k}$, will be determined using Weil's Theorem (Theorem 3.7). The group type of the curves may subsequently be determined using Lemma 4.5.

**Theorem 4.7 ([46])** *The number of isomorphism classes of elliptic curves over $F_q$, with Char($F_q$) $> 3$, is $2q+6, 2q+2, 2q+4, 2q$, for $q \cong 1, 5, 7, 11 \pmod{12}$ respectively.*

**Theorem 4.8 ([46])** *There are $2(2^k - 1)$ isomorphism classes of non-supersingular elliptic curves over $F_{2^k}$. Let $\gamma$ be an element of $F_{2^k}$, such that $Tr(\gamma) = 1$ (if $k$ is odd, we can take $\gamma = 1$). A set of representatives of the isomorphism classes is:*

$$\left\{ y^2 + xy = x^3 + a_2x^2 + a_6 \quad | \quad a_6 \in F_{2^k}^*, a_2 \in \{0, \gamma\} \right\} \quad .$$

**Theorem 4.9 ([46])** *There are 3 and 7 isomorphism classes of supersingular curves over $F_{2^k}$, where $k$ is odd and even respectively. For the $k$-odd case, let $\gamma$ be a non-cube in $F_{2^k}$, and let $\alpha, \beta, \delta, \omega \in F_{2^k}$ be such that*

$$Tr(\gamma^{-2}\alpha) = 1, \quad Tr(\gamma^{-4}\beta) = 1, \quad Te(\delta) \neq 0 \quad and \quad Tr(\omega) = 1 \ .$$

*Then a representative from each class is as shown in Table 4.3, along with the number of points and group types. (The $k$ values will be explained later)*

Given an arbitrary supersingular elliptic curve $E$ over $F_{2^k}$, we can compute $\#E(F_{2^k})$ by first determining to which isomorphism class $E$ belongs. Then we can use the results in the Table 4.3 to determine $\#E(F_{2^k})$. Several efficient polynomial time algorithms to find the isomorphism class of a curve can be found in [3].

**Table 4.3.** Isomorphism classes of supersingular ECs over the field $F_{2^k}$

| Curve $E$ | $k$ | $\#E(F_{2^k})$ | Group Type | $m$ |
|---|---|---|---|---|
| $y^2+y=x^3$ | odd | $q+1$ | cyclic | 2 |
| $y^2+y=x^3+x$ | $k\equiv 1,7 \pmod 8$ | $q+1+\sqrt{2q}$ | cyclic | 4 |
| | $k\equiv 3,5 \pmod 8$ | $q+1-\sqrt{2q}$ | cyclic | 4 |
| $y^2+y=x^3+x+1$ | $k\equiv 1,7 \pmod 8$ | $q+1-\sqrt{2q}$ | cyclic | 4 |
| | $k\equiv 3,5 \pmod 8$ | $q+1+\sqrt{2q}$ | cyclic | 4 |
| $y^2+\gamma y=x^3$ | $k\equiv 0 \pmod 4$ | $q+1+\sqrt{q}$ | cyclic | 3 |
| | $k\equiv 2 \pmod 4$ | $q+1-\sqrt{q}$ | cyclic | 3 |
| $y^2+\gamma y=x^3+\alpha$ | $k\equiv 0 \pmod 4$ | $q+1-\sqrt{q}$ | cyclic | 3 |
| | $k\equiv 2 \pmod 4$ | $q+1+\sqrt{q}$ | cyclic | 3 |
| $y^2+\gamma^2 y=x^3$ | $k\equiv 0 \pmod 4$ | $q+1+\sqrt{q}$ | cyclic | 3 |
| | $k\equiv 2 \pmod 4$ | $q+1-\sqrt{q}$ | cyclic | 3 |
| $y^2+\gamma^2 y=x^3+\beta$ | $k\equiv 0 \pmod 4$ | $q+1-\sqrt{q}$ | cyclic | 3 |
| | $k\equiv 2 \pmod 4$ | $q+1+\sqrt{q}$ | cyclic | 3 |
| $y^2+y=x^3+\delta x$ | $k$ even | $q+1$ | cyclic | 2 |
| $y^2+y=x^3$ | $k\equiv 0 \pmod 4$ | $q+1-2\sqrt{q}$ | $\mathcal{Z}_{\sqrt{q}-1}\oplus\mathcal{Z}_{\sqrt{q}-1}$ | 1 |
| | $k\equiv 2 \pmod 4$ | $q+1+2\sqrt{q}$ | $\mathcal{Z}_{\sqrt{q}+1}\oplus\mathcal{Z}_{\sqrt{q}+1}$ | 1 |
| $y^2+y=x^3+\omega$ | $k\equiv 0 \pmod 4$ | $q+1+2\sqrt{q}$ | $\mathcal{Z}_{\sqrt{q}+1}\oplus\mathcal{Z}_{\sqrt{q}+1}$ | 1 |
| | $k\equiv 2 \pmod 4$ | $q+1-2\sqrt{q}$ | $\mathcal{Z}_{\sqrt{q}-1}\oplus\mathcal{Z}_{\sqrt{q}-1}$ | 1 |

# Chapter 5
# THE ELLIPTIC CURVE LOGARITHM PROBLEM

## 5.1 Discrete Logarithm Problem

Let $G$ be a (multiplicatively written) finite cyclic group of order $n$, $\alpha$ be a generator for $G$ and $\beta \in G$. The *discrete logarithm* of $\beta$ to the base $\alpha$, denoted by $\log_\alpha \beta$, is the unique integer $v$, $0 \leq v < n$, such that $\beta = \alpha^v$. The *discrete logarithm problem* (DLP), is to find an 'easy' (i.e., computationally feasible) method for computing logarithms in a given group $G$. It is also referred to as the discrete-log problem. Brief descriptions of the known methods and algorithms are as follows:

### 5.1.1 Linear Search Method

Computing the successive powers of $\alpha$ until $\beta$ is found.

### 5.1.2 Square Root Methods

#### 5.1.2.1 Baby-Step Giant-Step

Let $u = \lceil \sqrt{n} \rceil$. Observe that if $v = \log_\alpha \beta$, then we can uniquely write $v = ju + i$, where $0 \leq i < u$. Precompute a list of pairs $(i, \alpha^i)$ for $0 \leq i < u$ and sort this list by second component. For each $j$, $0 \leq j < u$, compute $\beta\alpha^{-ju}$ and check (by binary search) if this element is equal to the second component of some pair in the list. If $\beta\alpha^{-ju} = \alpha^i$ for some $i$, $0 \leq i < u$, then $\beta = \alpha^{ju+i}$ and hence $\log_\alpha \beta = ju + i$. Note that this method requires a table with $u$ entries. [46].

#### 5.1.2.2 Pollard $\rho$-method

J. Pollard [57] gave a method to find logarithms which is probabilistic but removes the necessity of precomputing a list of logarithms.

Partition the group $G$ into three sets $S_1$, $S_2$ and $S_3$ of roughly equal size. (Some restrictions apply, such as $1 \notin S_2$.) Define a sequence of group elements $v_0, v_1, v_2, \ldots$ by

$$v_i = \begin{cases} 1 & i = 0 \ , \\ \beta v_{i-1} & v_i \in S_1 \ , \\ v_{i-1}^2 & v_i \in S_2 \ , \\ \alpha v_{i-1} & v_i \in S_3 \ , \end{cases}$$

for $i \geq 0$. It easily follows that the sequence of group elements defines a sequence of integers $\{a_i\}$ and $\{b_i\}$, where

$$\begin{aligned} v_i &= \beta^{a_i}\alpha^{b_i} \qquad i \geq 0 \\ a_0 &= b_0 = 0 \\ a_{i+1} &\cong a_i + 1, 2a_i \text{ or } a_i \pmod{n} \\ b_{i+1} &\cong b_i, 2b_i \text{ or } b_i + 1 \pmod{n} \end{aligned}$$

depending on which set ($S_1$, $S_2$ or $S_3$) contains $v_{i-1}$. Making use of Floyd's algorithm, Pollard computes the six tuple $(v_i, a_i, b_i, v_{2i}, a_{2i}, b_{2i})$, $i = 1, 2, \ldots$ until $v_i = v_{2i}$. At this stage, we have

$$\beta^r = \alpha^s$$

for $r \cong a_i - a_{2i}$ and $s \cong b_{2i} - b_i \pmod{n}$ . This gives

$$r \log_\alpha \beta \cong s \pmod{n} \ .$$

There are only $d = \gcd(r, n)$ possible values for $\log_\alpha \beta$. If $d$ is small then each of these possibilities can be enumerated to find the correct value.

### 5.1.3 Pohlig-Hellman Method

This method for computing logarithms in a cyclic group [56], takes advantage of the factorization of the order of the group. Let

$$n = \prod_{i=1}^{t} p_i^{\lambda_i} \ ,$$

where $p_i$ is a prime and $\lambda_i$ is a positive integer for each $1 \leq i \leq t$. If $v = \log_\alpha \beta$, then the approach is to determine $v$ modulo $p_i^{\lambda_i}$ for each $i$, $1 \leq i \leq t$, and then use the Chinese Remainder Theorem (*see* Appendix C) to compute $v$ modulo $n$. We begin by determining $z \cong v \pmod{p_1^{\lambda_1}}$. Suppose that

$$z = \sum_{i=0}^{\lambda_1 - 1} z_i p_1^i \ ,$$

where $0 \leq z_i \leq p_1 - 1$. Let $\gamma = \alpha^{n/p_1}$ be a $p_1$th root of unity in $G$. Then

$$\beta^{n/p_1} = \alpha^{vn/p_1} = \gamma^v = \gamma^{z_0} \ .$$

Using one of the square root methods described in the previous section we determine the logarithm of $\gamma^{z_0}$ to the base $\gamma$ in the cyclic group of order $p_1$ in $G$. This gives us $z_0$. If $\lambda_1 > 1$, then to determine $z_1$ we consider

$$(\beta\alpha^{-z_0})^{n/p_1^2} = \left(\alpha\sigma_{i=1}^{\lambda_1 - 1} z_i p_1^i\right)^{n/p_1^2} = \lambda^{z_1} \ .$$

Again $z_1$ can be found by a square root method. In this manner we can determine all $z_i$, $0 \leq i < \lambda_1$, and thus $v$ modulo $p_1^{\lambda_1}$ [46].

### 5.1.4 Index Calculus Method

First we attempt to find the logarithms of the elements of a fixed subset $\Gamma = \{\gamma_1, \gamma_2, \ldots, \gamma_t\}$ of $G$, called the *factor base*, as follows. We pick a random integer $s$ and attempt to write $\alpha^s$ as a product of elements in $\Gamma$:

$$\alpha^s = \prod_{i=1}^{t} \gamma_i^{a_i} \ .$$

If we succeed, then taking the logarithms of both sides of this equality we get the linear congruence

$$s \cong \sum_{i=1}^{t} a_i \log_\alpha \gamma_i \pmod{n} \ .$$

After collecting a sufficient number of relations in that form, one can then hopefully solve for the indeterminates $\log_\alpha \gamma_i$, $1 \leq i \leq t$.

In the second stage, we find $\log_\alpha \beta$ as follows. Repeatedly pick random integers $s$ until $\alpha^s \beta$ can be written as a product of elements in $\Gamma$:

$$\alpha^s \beta = \prod_{i=1}^{t} \gamma_i^{b_i} \ .$$

Taking the logarithm of both sides, we get

$$\log_\alpha \beta = \sum_{i=1}^{t} b_i \log_\alpha \gamma_i - s \pmod{n} \ .$$

The choice of the factor base determines the running time. An appropriate choice of $\Gamma$ will be a small set and at the same time the proportion of elements of $G$ that factor in $\Gamma$ is large [41, 46].

For the field $F_p$, a practical method is the Gaussian integer method [7] and the fastest one is the number field sieve [34] (although it appears to be impractical at present). For the field $F_{2^k}$ (or in general $F_{p^k}$, where $p$ is fixed [22]), we represent the elements of $F_{2^k}$ as polynomials in $F_2[x]$ of degree at most $(k-1)$, where the multiplication is performed modulo a fixed irreducible polynomial of degree $k$ in $F_2[x]$. The set $\Gamma$ is then taken to be the set of all irreducible polynomials of degree at most some prescribed bound $b$. To write $\alpha^s$ as a product of elements in $\Gamma$ we express it as a polynomial of degree at most $(k-1)$, and attempt to factor it in $F_2[x]$ as a product of polynomials in $\Gamma$. The running time of this method (after some improvements) is the subexponential time (*see* Appendix A) $L[2^k, c, 1/3]$, where $1.3507 \leq c \leq 1.4047$ [6]. The best algorithms for $F_p$ and $F_{2^k}$ with rigorously proved running times are due to Pomerance [58]. For fields $F_{p^k}$, the number field sieve is the best algorithm known. Unfortunately it is still unknown whether there exist subexponential algorithms (with either heuristically or rigorously proven running times) for the DLP in fields where both $q$ and $k$ tend to infinity. The running times of the above algorithms are compared at the Table 5.1 [46].

**Table 5.1.** Comparison of methods that solve the discrete-log problem

| Method | | Time |
|---|---|---|
| Linear Search | | $O(n)$ |
| Square Root Methods | Baby-Step Giant-Step | $O(\lceil\sqrt{n}\rceil \log n)$ |
| | Pollard $\rho$ | $O(\lceil\sqrt{n}\rceil)$ |
| Pohlig-Hellman | | $O(\sum_{i=1}^{t} \lambda_i(\log n + \sqrt{p_i} \log p_i))$ |
| Index Calculus Methods | Gaussian integer (for $F_p$) | $L[p, 1, 1/2]$ |
| | Pomerance (for $F_p$) | $L[p, \sqrt{2}, 1/2]$ |
| | Number Field Sieve (for $F_p$) | $L[p, 3^{2/3}, 1/3]$ |
| | Number Field Sieve (for $F_{p^k}$) | $L[p^k, c, 1/3]$ ($c$ is const) |
| | Pomerance (for $F_{2^k}$) | $L[2^k, \sqrt{2}, 1/2]$ |

## 5.2 Elliptic Curve Logarithm Problem

Let $P \in E(F_q)$ be a point of order $n$ and

$$E(F_q) \cong \mathcal{Z}_{n_1} \oplus \mathcal{Z}_{n_2} \ .$$

Also assume that $n$ is known and $\gcd(\#E(F_q), q-1) = 1$. Then the *elliptic curve logarithm problem* (ECLP), is to determine the unique integer $l$, such that $R = lP$, provided that such an integer exist.

In the paper [43], A. Menezes, T. Okamoto and S. Vanstone showed that the ECLP in $E(F_q)$ can be reduced to the DLP in a finite field $F_{q^m}$, where $E(F_q)$ has type $(n_1, n_2)$ and $m$ is the smallest integer such that $E(n_1) \subseteq E(F_{q^m})$, and hence $E(n) \subseteq E(F_{q^m})$. Furthermore $E(F_{q^m})$ has type $(cn_1, cn_2)$. In general the reduction algorithm takes exponential time (in $\ln q$), but they proved that it takes probabilistic polynomial time for supersingular curves. When combined with a subexponential algorithms for the DLP in a finite field, this yields a probabilistic subexponential time algorithm to compute the ECLPs for supersingular curves. This algorithm is

**Table 5.2.** Parameters used in reducing supersingular ECLP to DLP

| Class of curve | t | Group Structure | Comment | $n_1$ | $m$ |
|---|---|---|---|---|---|
| I | 0 | cyclic | | $q+1$ | 2 |
| II | 0 | $\mathcal{Z}_{(q+1)/2} \oplus \mathcal{Z}_2$ | $q \equiv 3 \pmod 4$ | $(q+1)/2$ | 2 |
| III | $\pm\sqrt{q}$ | cyclic | $k$ is even | $q+1 \mp \sqrt{q}$ | 3 |
| IV | $\pm\sqrt{2q}$ | cyclic | $p=2$, $k$ is odd | $q+1 \mp \sqrt{2q}$ | 4 |
| V | $\pm\sqrt{3q}$ | cyclic | $p=3$, $k$ is odd | $q+1 \mp \sqrt{3q}$ | 6 |
| VI | $\pm 2\sqrt{q}$ | $\mathcal{Z}_{\sqrt{q}\mp 1} \oplus \mathcal{Z}_{\sqrt{q}\mp 1}$ | $k$ is even | $\sqrt{q} \mp 1$ | 1 |

| Class of Curve | Type of $E(F_{q^m})$ | c |
|---|---|---|
| I | $(q+1, q+1)$ | 1 |
| II | $(q+1, q+1)$ | 2 |
| III | $(\sqrt{q^3} \pm 1, \sqrt{q^3} \pm 1)$ | $\sqrt{q} \pm 1$ |
| IV | $(q^2+1, q^2+1)$ | $q \pm \sqrt{2q} + 1$ |
| V | $(q^3+1, q^3+1)$ | $(q+1)(q \pm \sqrt{3q} + 1)$ |
| VI | $(\sqrt{q} \pm 1, \sqrt{q} \pm 1)$ | 1 |

called the *MOV attack*. Thus, for supersingular curves, the ECLP is more tractable than was previously believed. Table 5.2 summarizes the results [46]. The $m$ values for all isomorphism classes of supersingular elliptic curves over the finite field $F_{2^k}$ were indicated on Table 4.3 before.

If a non-supersingular curve is desired, then the curve must be chosen so that the corresponding $m$ value is sufficiently large. (By sufficiently large we mean $m > c$ values for which the DLP in $F_{q^c}$ is considered intractable)

Frey and Rück showed how to use a variant of the Tate pairing for Abelian varieties over local fields, to reduce the ECLP in the $n$-torsion part of the divisor class group of a projective irreducible non-singular curve over $F_q$ (with $\text{char}(F_q)$

coprime with $n$), to the DLP in $F_{q^m}$, where $m$ is the smallest integer such that $n|(q^m - 1)$. For elliptic curves, this method is advantageous over the above method as the condition $n|(q^m - 1)$ is usually weaker than the condition $E(n) \subseteq E(F_{q^m})$ [46].

## 5.3 Group Structure

Miller [49] suggested an algorithm to find the type $(n_1, n_2)$ of an elliptic curve $E$ defined over $F_q$ in probabilistic polynomial time, given the factorization of $N$ or $\gcd(N, q-1)$, where $N = \#E(F_q)$.

# Chapter 6
# IMPLEMENTATION OF ELLIPTIC CURVE CRYPTOSYSTEMS

## 6.1 Advantages of Elliptic Curves

There are many analogies between the group of $F_q$-rational points on an elliptic curve and the multiplicative group $F_q^*$: approximately same number of elements, etc. But for a single (large) $q$ there are many different elliptic curves and many different $N$ to choose from. Hence elliptic curves offer a rich source of "naturally occurring" finite Abelian groups. It is this major advantage that makes elliptic curves attractive for cryptography.

## 6.2 Selecting a Curve

One of the most crucial steps in developing an elliptic curve cryptosystem, is selecting the appropriate curve to use. Some elliptic curves are susceptible to attacks that make them no more secure than existing systems today. Although not proven the curves that has the following characteristics are believed to provide more security than all the other systems in use [46]:

- Non-supersingular (against MOV attack; *see* Section 5.2)
- Large order (against square root methods; *see* Section 5.1.2)
- The order is divisible by a large prime factor (against Pohlig-Hellman method; *see* Section 5.1.3)
- Optimal normal basis in the underlying field (for efficient finite field arithmetic; *see* Section 8.2)

In addition to above, we will prefer a curve whose group is cyclic.

## 6.3 Counting Points on Elliptic Curves

We have seen in Section 6.2 that security of the Elliptic Curve Cryptosystem depends on the curve selected, and some properties that its order should satisfy. So it is crucial to know the order or else to find it.

One method of selecting curves is to choose a curve $E$ defined over $F_q$, where $q$ is small enough so that $\#E(F_q)$ can be computed directly, and then using the group $E(F_{q^m})$ for suitable $m$. Note that $\#E(F_{q^m})$ can be computed from $\#E(F_q)$ by the Weil Theorem (Theorem 3.7). Observe also that if $n$ divides $m$, then $\#E(F_{q^n})$ divides $\#E(F_{q^m})$, and so we should select $n$ such that it is prime, or else a product of a small factor and a large prime. But clearly the curve selected that way is not a random one [46].

If a random elliptic curve is required, then the order of the curve should be computed. In 1985, Schoof [62] presented a polynomial time algorithm for computing $\#E(F_q)$, the number of $F_q$-rational points on an elliptic curve $E$ defined over the field $F_q$, where $q$ is an odd prime. A step by step presentation of this algorithm, which we constructed based on the above original paper, is presented in Section 6.4. The algorithm has a running time of $O(\log^8 q)$ bit operations, and is rather cumbersome in practice. The algorithm basically uses endomorphisms and division polynomials. Using fast multiplication technics it is possible to reduce the total running time to $O(\log^{5+\epsilon} q)$ bit operations, for any $\epsilon > 0$, however since these technics are only practical for very large $q$, there is still place for improvement. Buchmann and Muller combined Schoof's algorithm with Shanks' baby-step giant-step algorithm, and were able to compute orders of curves over $F_p$, where $p$ is a 27-decimal digit prime [46].

However, as explained before, from the point of view of practical cryptography curves over fields of characteristic 2 are more important. For this reason, in [28] Koblitz adapted Schoof's algorithm to curves over $F_{2^k}$ and studied the implementation and security of a random curve cryptosystem. Using heuristic arguments, Koblitz showed that if $E/F_q$ is a randomly chosen non-supersingular elliptic curve, then the probability that $N = \#E(F_q)$ is divisible by a prime factor $\geq N/B$, is about

$\frac{1}{k}\log_2(B/2)$. Thus, for example, the probability that the order of a randomly chosen non-supersingular curve over $F_{2^{155}}$ is divisible by a 40-digit prime is approximately 0.136, which means one can expect to try about 7 curves before a suitable one is found [46].

Menezes, Vanstone and Zuccherato implemented a version of Schoof's algorithm for curves over fields of characteristic 2, and used some heuristic arguments to improve the running time, such as using Schank's baby-step giant-step method to manage modular arithmetic efficiently [44]. In their implementation multiplying field elements roughly takes 90% of all the time taken by the algorithm. Thus building a special purpose chip or finding faster methods to reduce the time taken by field multiplications will dramatically improve the total running time. Other possible improvements that they did not implement, were to use Pollard's lambda method for catching kangaroos [57] instead of the baby-step giant-step method. Both has the same expected running time but the later requires very little storage [46].

In 1991, Atkin described a new algorithm for computing $\#E(K)$ which uses modular equations. The algorithm has not been rigorously analyzed but performs remarkably well in practice. He also implemented another algorithm inspired by Elkies' ideas, but it is only used for $q$ odd, and generalization to the case $q$ even does not appear to be straightforward [46].

## 6.4 Schoof's Algorithm

Schoof's algorithm is a deterministic algorithm to compute the number of points on an elliptic curve. The versions presented below are constructed based on Schoof's original paper [62]. An outline of the Schoof's algorithm is as follows:

By Hasse's Theorem (Theorem 3.2), the number of points on an elliptic curve $E$ over the field of $q = 2^k$ elements is of the form $N = q + 1 - t$, where $|t| \leq 2\sqrt{q}$. Schoof's Algorithm determines $N$ modulo $l$ for a bunch of small primes $l$. If we run through enough $l$ so that $\prod l > 4\sqrt{q}$, then $N$ can be uniquely determined by the Chinese Remainder Theorem (*see* Appendix C).

For $l > 2$, one determines $N$ modulo $l$ by looking at the points of order $l$, with coordinates in field extensions of $F_q$. It turns out that $N$ modulo $l$ is determined by the action of the map $(x, y) \mapsto (x^q, y^q)$ on the set of points of order $l$. For example, suppose that the map $(x, y) \mapsto (x^q, y^q)$ leaves some such point fixed. Then this means that there is a point of order $l$ whose coordinates are in $F_q$, i.e., our original group of points with $F_q$ coordinates has a nontrivial element of order $l$. In that case $N \equiv 0 \pmod{l}$. More generally, the value of $N$ modulo $l$ is determined by how the $q$-th power map permutes the points of order $l$. Thus, a basic role in Schoof's algorithm is played by the so-called "division polynomials", which characterize the point $P$ (with coordinates in extensions of $F_q$) for which $lP$ is the identity.

Below is the Schoof's algorithm, preceded with the definitions used while describing it.

### 6.4.1 Definitions

Let $E/F_q$ be the curve $y^2 = x^3 + ax + b$, where $char(F_q)neq2, 3$. The polynomials $\Psi_n(x, y) \in F_q[x, y]$ for $n \geq -1$ are defined as:

- $\Psi_{-1}(x, y) = -1$
- $\Psi_0(x, y) = 0$
- $\Psi_1(x, y) = 1$
- $\Psi_2(x, y) = 2y$
- $\Psi_3(x, y) = 3x^4 + 6ax^2 + 12bx - a^2$
- $\Psi_4(x, y) = 4y(x^6 + 5ax^4 + 20bx^3 - 5a^2x^2 - 4abx - 8b^2 - a^3)$
- $\Psi_{2n}(x, y) = \Psi_n(\Psi_{n+2}\Psi_{n-1}^2 - \Psi_{n-2}\Psi_{n+1}^2)/2y \qquad (n \geq 1)$
- $\Psi_{2n+1}(x, y) = \Psi_{n+2}\Psi_n^3 - \Psi_{n+1}^3\Psi_{n-1} \qquad (n \geq 1)$

The *division polynomials* $f_n(x) \in F_q[x]$ are defined as follows: First eliminate all $y^2$ terms from $\Psi_n$ using the elliptic curve relation, and call it $\Psi'_n$, then define:

$$f_n(x) = \begin{cases} \Psi'_n(x,y) & \text{if } n \text{ is odd,} \\ \Psi'_n(x,y)/y & \text{if } n \text{ is even.} \end{cases}$$

First version describes the algorithm using words and the latter is the more detailed and formal version. Parallelization is also introduced during the detailed description in the second version.

### 6.4.2 Version 1

**Input:** Underlying field $F_q$, the list of primes up to $\mathcal{O}(\sqrt{q})$,
polynomials $f_n$ for $-1 \leq n \leq 4$

**Output:** $\#E(F_q)$, i.e., the number of points on the elliptic curve $E(F_q)$

1. Find the smallest $L$ such that product of all primes up to $L$ (except 2 and p) is greater than $4\sqrt{q}$ ; call this set of primes as S.
2. Calculate the functions $f_n$ for $5 \leq n \leq L$
3. For each number $l$ in the set $S$ do the followings:
   (a) $v \equiv q \pmod{l}$
   (b) If there is a non-zero point $P$ in $E[l]$ for which $\phi_l^2 P = \pm vP$ holds then
      i. if $\phi_l^2 P = -vP$ holds
         then $t \equiv 0 \pmod{l}$
         else (i.e., if $\phi_l^2 P = vP$ holds)
         A. find a squareroot $w$ of $q \pmod{l}$
         B. if $\phi_l P \neq \pm wP$
            then $t \equiv 0 \pmod{l}$
            else if $\phi_l P = wP$

then $t \equiv 2w \pmod{l}$

else (i.e., $\phi_l P = -wP$) $t \equiv -2w \pmod{l}$

else (i.e., if there is no point satisfying $\phi_l^2 P = \pm vP$)

check for every $\tau \in \{1, 2, \ldots, l\}$ to see if there is a non-zero point $P$ in $E[l]$ for which $\phi_l^2 P + qP = \tau\phi_l$ holds. Store the solution as $t \equiv \tau \pmod{l}$.

4. Compute the value of $t$ using the values $(t \bmod l)$ for $(l \in S)$ in the Chinese Remainder Theorem (*see* Appendix C).

5. The number of points on the elliptic curve is equal to $(q + 1 - t)$.

### 6.4.3 Version 2

**Input:** Underlying field $F_q$, the list of primes up to $\mathcal{O}(\sqrt{q})$,
polynomials $f_n$ for $-1 \leq n \leq 4$

**Output:** Integer $\#E(F_q)$ which is the number of points on the elliptic curve $E(F_q)$

1. Find the smallest $L$ such that product of all primes up to $L$ (except 2 and p) is greater than $4\sqrt{q}$ ; call this set of primes as S.

2. For $u$ from 1 up to $logL$ do the followings in parallel: Calculate the function $f_n$ for $4 + 2^{u-1} \leq n \leq 3 + 2^u$

3. In parallel for each number $l$ in the set $S$, do the followings:

   (a) $v \equiv q \pmod{l}$

   (b) if $v$ is even then

   $\mathrm{GCD}_1 = \gcd((x^{q^2} - x) f_v^2(x)(x^3 + Ax + B) + f_{v-1}(x) f_{v+1}(x), f_l(x))$

   else

   $\mathrm{GCD}_1 = \gcd((x^{q^2} - x) f_v^2(x) + f_{v-1}(x) f_{v+1}(x)(x^3 + Ax + B), f_l(x))$

   (c) if $\mathrm{GCD}_1 \neq 1$ then

   i. if $\left(\frac{q}{l}\right) = -1$ then $t \equiv 0 \pmod{l}$

   else

A. Find a square root $w$ of $q$ by checking in parallel for
all $w \in \{1, 2, \ldots, l-1\}$ if $w^2 \equiv q \pmod{l}$

B. if $w$ is even then
$\mathrm{GCD}_2 = \gcd((x^q - x)f_w^2(x)(x^3 + Ax + B) + f_{w-1}(x)f_{w+1}(x),\ f_l(x))$
else
$\mathrm{GCD}_2 = \gcd((x^q - x)f_w^2(x) + f_{w-1}(x)f_{w+1}(x)(x^3 + Ax + B),\ f_l(x))$

C. if $\mathrm{GCD}_2 = 1$ then $t \equiv 0 \pmod{l}$
else

I. if $w$ is even then
$$\mathrm{GCD}_3 = \gcd(4(x^3 + Ax + B)^{(q-1)/2} f_w^3(x) - f_{w+2}^2(x)f_{w-1}(x) + f_{w-2}^2(x)f_{w+1}(x), f_l(x))$$
else
$$\mathrm{GCD}_3 = \gcd(4(x^3 + Ax + B)^{(q+3)/2} f_w^3(x) - f_{w+2}^2(x)f_{w-1}(x) + f_{w-2}^2(x)f_{w+1}(x), f_l(x))$$

II. if $\mathrm{GCD}_3 = 1$ then $t \equiv -2w \pmod{l}$
else $t \equiv 2w \pmod{l}$

else find the values of $\tau \in \{1, 2, \ldots, l\}$ for which the polynomials $\mathrm{POL}_1$ and $\mathrm{POL}_2$ are zero $\pmod{f_l(x)}$. (can be done in parallel for every $\tau \in \{1, 2, \ldots, l\}$) Store the solution as $t \equiv \tau \pmod{l}$.

i. $\alpha = \Psi_{v+2}\Psi_{v-1}^2 - \Psi_{v-2}\Psi_{v+1}^2 - 4Y^{q^2+1}\Psi_v^3$

ii. $\beta = ((x - x^{q^2})\Psi_v^2 - \Psi_{v-1}\Psi_{v+1}) - 4Y\Psi_v$

iii. $$\mathrm{POL}_1 = ((\Psi_{v-1}\Psi_{v+1} - \Psi_v(x^{q^2} + x^q + x))\beta^2 + \Psi_v^2\alpha^2) + \Psi_\tau^{2q} + \Psi_{\tau-1}^q\Psi_{\tau+1}^q\beta^2\Psi_v^{2q}$$

iv. $$\mathrm{POL}_2 = 4^q Y^q \Psi_\tau^{3q}(\alpha((2x^{q^2} + x)\Psi_v^2\Psi_{v-1}\Psi_{v+1}) - Y^{q^2}\beta\Psi_v^2) - \beta\Psi_v^2(\Psi_{\tau+2}\Psi_{\tau-1}^2 - \Psi_{\tau-2}\Psi_{\tau+1}^2)^q$$

4. Compute the value of $t$ using the values $(t \bmod l)$ for $(l \in S)$ in the Chinese Remainder Theorem (*see* Appendix C).

5. $\#E(F_q) = (q + 1 - t)$.

Note that for characteristic 2, the general form of the curve $E/F_q$ is

$$y^2 + xy = x^3 + a_2x^2 + a_6 \ ,$$

where $a_2 \in \{0, \omega\}$, $\omega \in F_q$ being a fixed element of trace 1, and $a_6 \in F_q^*$. Then the *division polynomials* given above simplifies as shown below:

$$\begin{aligned}
f_0 &= 0 \\
f_1 &= 1 \\
f_2 &= x \\
f_3 &= x^4 + x^3 + a_6 \\
f_4 &= x^6 + a_6x^2 \\
f_{2n+1} &= f_n^3 f_{n+2} + f_{n-1}f_{n+1}^3 \qquad n \geq 2 \\
xf_{2n} &= f_{n-1}^2 f_n f_{n+2} + f_{n-2}f_n f_{n+1}^2 \qquad n \geq 3
\end{aligned}$$

The algorithm will also be simplified ( *see* [28, 46] or [44]).

## 6.5 A Cryptographically Useful Subclass of Elliptic Curves

The following proposition gives the orders of some special elliptic curves:

**Proposition 6.1 ([4])** *The order of the group of the elliptic curve defined by* $y^2 = x^3 + a_6$ *over* $F_p$ *with* $p \cong -1 \pmod 3$ *is* $p+1$. *The order of the group of the elliptic curve defined by* $y^2 + y = x^3 + a_6$ *over* $F_{2^k}$, *where* $k$ *is odd, is* $2^k + 1$.

In the implementation, the cryptographically useful curves are found, if the number $p^*$ in the factorization of the group order

$$p + 1 = 2 \cdot 3 \cdot p^* \qquad \text{or} \qquad 2^k + 1 = 3 \cdot p^*$$

is a large prime. In [4], it is shown that it is indeed possible to find curves with this property. On these curves, finding an element with high order is achieved by calculating the $6^{th}$ and the $3^{rd}$ order of some arbitrary element, respectively, and then checking if the result is not the identity. If it is not, then the order of this element is a multiple of $p^*$.

**Table 6.1.** Operations required by different methods

| Method | Adding two distinct points | Doubling a point |
|---|---|---|
| Addition formulas | I + 2M | I + 3M |
| Projective coordinates | 13M | 7M |

Notation: I=Inversion M=Multiplication

## 6.6 Improving Speed

Instead of applying the addition formulas directly one can change the method, i.e., restore to projective coordinates to gain speed by avoiding the costly inversion operation. Table 6.1 summarizes the number of operations needed by different methods [46].

## 6.7 ElGamal Cryptosystem Using Elliptic Curves

Elliptic curves can be used to implement the ElGamal Cryptosystem [47]. The protocol is as follows:

1. (Setup) A non-supersingular curve $E$ of the form $y^2 + xy = x^3 + a_2x^2 + a_6$ defined over $F_{2^k}$ is chosen. A point $P$ (preferably a generator) on $E$ is made public. A normal basis is used to represent the elements of $F_{2^k}$. User $B$ wants to send the message $(M_1, M_2)$ to user $A$.

2. $A$ chooses an integer $a$ and makes public $aP$, while keeping $a$ itself secret.

3. $B$ selects a random integer $v$ and computes the points $vP$ and $avP = (\bar{x}, \bar{y})$, using the public point $P$ and $A$'s public key $aP$.

4. Assuming that $\bar{x}, \bar{y} \neq 0$, $B$ sends to $A$ the point $vP$, and the field elements $M_1\bar{x}$ and $M_2\bar{y}$.

5. To read the message, $A$ multiplies the point $vP$ by his secret key $a$ to obtain $(\bar{x}, \bar{y})$, from which he can recover $M_1$ and $M_2$ in two divisions.

A multiple of an elliptic curve point can be found using the repeated double-and-add algorithm (*see* Appendix B.2). Morain [50] presents some addition-subtraction chains for integers $k$ which lead to faster algorithms than that method. Koyama and Tsuruoka do the same for elliptic curves over the ring $\mathcal{Z}_n$ [46].

In this system four field elements are transmitted in order to convey a message consisting of two field elements. Thus there is *message expansion* by a factor of 2. This can be reduced to 3/2 by only sending $x_1$ and a single bit of $y_1/x_1$ (if $x_1 \neq 0$), instead of sending the point $P = (x_1, y_1)$. This is called *point compression technic.* Details can be found in [46] and [23]. For a software implementation of the ElGamal cryptosystem over the finite field $F_{2^{104}}$ see the paper [20].

In [8], R. Candall describes an implementation of the elliptic curve analogue of the Diffie-Hellman key exchange. Crandall presents a method for performing arithmetic modulo $p$ using only shift and add operations, eliminating the need for costly divisions. This technic, together with an inversionless parameterization of the elliptic curve, results in a very efficient implementation of elliptic curve arithmetic. The system is called Fast Elliptic Encryption (FEE).

## 6.8 Implementations

Table 6.2 summarizes the results of some of the implementations [46].

For some other hardware designed to perform calculations in finite fields, see [9, 12, 14, 71] and also consult the books [37, 42, 45, 24].

**Table 6.2.** Speed comparison of some implementations (in clock cycles)

| Year | Implementer/Device | Clock Rating | Inversion | Multiplication |
|---|---|---|---|---|
| 1988 | Newbridge Microsystems Inc. & Cryptech Systems Inc. | 20 MHz | 50,000 | 1,300 |
| 1993 | VLSI device | 40 MHz | 3,800 | 165 |

# Chapter 7
# OTHER APPLICATIONS OF ELLIPTIC CURVES

## 7.1 Primality Testing Using Elliptic Curves

The classical methods of primality testing are as follows [35]:

**Trial Division:** Divisibility of the given number is tested with primes up a certain bound. Clearly the bound should be less than $\sqrt{n}$, but it is infeasible if it is very big.

**Fermat's Little Theorem:** Finding an integer $a$ for which $a^n \not\equiv a \pmod{n}$ will prove that the number $n$ is composite. The number $a$ is called a it witness. However finding a witness might be difficult or even impossible. There exist composite numbers, the so-called *Carmichael numbers*, for which there exist no witnesses of $n$.

**Probabilistic Compositeness Test:** A witness of the compositeness of the number $n$ is found if an integer $a \in \{1, 2, \ldots, n-1\}$ is found that fails to satisfy

$$a \equiv 1 \pmod{n} \qquad \text{or} \qquad a^{r \cdot 2^i} \equiv -1 \pmod{n} ,$$

for some $i$ with $0 \leq i < k$, where $n - 1 = r \cdot 2^k$. If the generalized Riemann hypothesis were known to be true, it would suffice to verify the above condition for $a$ in $\{2, 3, \ldots, \lceil 2(\log n)^2 \rceil\}$, because the interval would contain a witness if $n$ were composite.

**Pocklington's Theorem:** Let $s$ be a positive divisor of $(n-1)$ with $s > (\sqrt{n} - 1)$. Randomly select integers $a$, until one satisfies:

$$a^{n-1} \equiv 1 \pmod{n} \qquad \text{and} \qquad \gcd(a^{(n-1)/q} - 1, n) = 1$$

for each prime $q$ dividing $s$. Then every prime dividing $n$ is congruent to 1 modulo $s$. If that doesn't work, then $n$ is probably not a prime.

**Downrun:** Simultaneously factor $n+1$ and $n-1$. If the unfactored part of any of them is found to be a probable prime, then apply the strategy recursively to prove the primality of this newly found probable prime. In this way a chain of primes $n = n_0, n_1, \ldots, n_t$ is built, such that $n_i$ divides $n_{i-1}+1$ or $n_{i-1}-1$ and such that the primality of $n_i$ implies the primality of $n_{i-1}$.

**11 Divisors:** The algorithm based on the following theorem is not only polynomial in time; but it is even efficient in practice.

**Theorem 7.1** *Let $r$, $s$ and $n$ be integers satisfying*

$$0 \leq r < s < n, \qquad s > \sqrt[3]{n}, \qquad \gcd(r,s) = 1 \ .$$

*Then there exist at most 11 positive divisors of $n$ that are congruent to $r$ modulo $s$, and there is a polynomial algorithm for determining all these divisors.*

A combination of this theorem with Pocklington's Theorem yields the following primality test: First find an integer $a$ satisfying both conditions in the previous method. This shows that the prime divisors of $n$ are all congruent to 1 modulo $s$. Next apply the algorithm in the above theorem to find at most 11 positive divisors of $n$ that are congruent to 1 modulo $s$. If no nontrivial factor of $n$ has been found in this way, then $n$ is prime.

**Jacobi Sum Test:** If, for positive integers $n$ and $s$, certain 'Fermat-like' tests hold for $n$, $s$, and the prime divisors $q$ of $s$, then any prime divisor of $n$ is congruent to a power of $n$ modulo $s$. This test is similar to the previous method but here $s$ can be taken as it any product $> \sqrt{n}$. The tests involving Gauss sums can be replaced by tests involving Jacobi sums. The Jacobi sum test consists of various 'Fermat-like' tests.

The main problem with the classical primality tests is that they are tied to groups that are fixed as soon as $n$ is fixed. If those groups turn out not to have the

favorable properties that are needed to complete the primality proof, then nothing can be done about it, since changing the group would change $n$ and consequently change the problem. Elliptic curves provide the choice of groups such that their relevant properties are randomized in the proper way, so that if the group chosen does not have the right properties, then another group will be chosen [35].

For a randomly chosen elliptic curve over $F_n$, with $n$ prime, the group order $\#E(F_n)$ will behave as a random integer near $(n+1)$, by Hasse's Theorem (Theorem 3.2). And the choice of the elliptic curve can be repeated until $E(F_n)$ has the favorable properties required. The following theorem can be formulated:

**Theorem 7.2 ([35])** *Let $n > 1$ be an integer with $\gcd(n, 6) = 1$. Let $E$ be an elliptic curve over $\mathcal{Z}/n\mathcal{Z}$, and let $u$ and $s$ be positive integers with $s$ dividing $u$. Suppose there is a point $P$ on the curve satisfying:*

$$u \cdot P = \mathcal{O} \qquad \textit{and} \qquad \frac{u}{q} \cdot P \neq \mathcal{O}$$

*for each prime $q$ dividing $s$. Then $\#E(F_p) \equiv 0 \pmod{s}$ for every prime $p$ dividing $n$, and if $s > (n^{1/4}+1)^2$ then $n$ is prime.*

This theorem leads to the following primality test based on elliptic curves [35]: Select an elliptic curve $E$ over $\mathcal{Z}/n\mathcal{Z}$ and an integer $u$ such that

$$u = \#E(F_n) \qquad \text{if } n \text{ is prime}$$

and

$$u = v \cdot q \ ,$$

for a small integer $v > 1$ and a probable prime $q > (n^{1/4}+1)^2$.

Given the pair $(E, u)$, select a point $P \in E(\mathcal{Z}/n\mathcal{Z})$ satisfying the requirements of the above theorem, with $s = q$ by performing the following steps:

1. Randomly select an $x \in \mathcal{Z}/n\mathcal{Z}$ and find $y$ such that the point $P = (x, y)$ is on the elliptic curve $E(\mathcal{Z}/n\mathcal{Z})$.

2. Compute $(u/q) \cdot P = v \cdot P$. If $v \cdot P$ is undefined, then a nontrivial divisor of $n$ has been found, which is exceedingly unlikely. If $v \cdot P = \mathcal{O}$, then go back to the previous step; this happens with probability $< 1/2$ if $n$ is prime. Otherwise if $v \cdot P \neq \mathcal{O}$, verify that $q \cdot (v \cdot P) = u \cdot P = \mathcal{O}$, which must be the case if $n$ is prime, because then $\#E(F_n) = u$.

3. Prove the primality of $q$ recursively using this algorithm, unless the primality of $q$ can be proved directly using some other method.

Using an efficient method for choosing $E$ and $u$ as explained in [35], the expected running time of the above test will be polynomial in $\log n$.

## 7.2 Factorization Based on Elliptic Curves

Lenstra's factorization method is based on the same idea as Pollard's $(p-1)$ method (for both algorithms *see* Appendix B.5). The order of the group of rational points on the elliptic curve is used in a similar way as $(p-1)$. The two methods are almost identical. The advantage of the elliptic curve is that there are a large number of different curves that can be tried each with a potentially different order. In the Pollard's method, we don't have the choice of changing $p$ [67, 59].

The following theorem is due to Lenstra:

**Theorem 7.3 ([67])** *The elliptic curve method with $v = L[p, 1, 1/2]^{1/\sqrt{2}}$ splits any integer $n$ in expected time $O(L[p, 1, 1/2]^{\sqrt{2}+O(1)}(logn)^2)$ where $p$ is the smallest prime divisor of $n$.*

**Corollary 7.4 ([67])** *The elliptic curve method can be used to factor completely any $n$ in expected time $O(L[n, 1, 1/2]^{1+O(1)})$.*

The above results show that asymptotically, Lenstra's method is as good as any previously known method to factor $n$. It has ,however, in addition several advantages. It is easy to program and ideally suited to implementation in parallel; many elliptic curves can be tried simultaneously and independently. It requires very little storage

and can be conveniently run as a background job. And if $n$ has a small prime factor, this can be expected to be found sooner than larger ones and helps to terminate the method in a shorter time. The running time of the previous methods was independent of the size of the factors [67].

# Chapter 8
# FINITE FIELDS

## 8.1 Choosing a Field

To efficiently implement the Elliptic Curve Cryptosystems, it is important to select a curve and a field so that the number of field operations involved in adding two points is minimized. Curves over the field $K = F_{2^k}$ are preferred because of the following reasons [46]:

- The arithmetic in $F_{2^k}$ is easier to implement in computer hardware than the arithmetic in finite fields of characteristic greater than 2, because of its binary nature.
- When using a normal basis representation for the elements of $F_{2^k}$, squaring a field element becomes a simple shift of the vector representation, and thus the multiplication count in adding two points is reduced.
- With curves over $F_{2^k}$, it is easy to use the point compression technic (*see* Section 6.7), i.e., to recover the y-coordinate of a point given its x-coordinate plus a bit of extra information. This is useful in reducing message expansion in the ElGamal cryptosystem as will be explained later in this chapter.

For these reasons and because the supersingular curves do not offer an advantage over finite field based cryptosystems, only the non-supersingular curves over $F_{2^k}$ will be considered in the rest of this thesis.

## 8.2 Representation of Finite Field Elements

For short notes on finite fields, please refer to Appendix D. The elements of the field $GF(2^k)$ can be represented in several different ways [42, 45, 38]. The polyno-

mial representation seems to be useful and suitable for both hardware and software implementations. According to this representation an element $a$ of $GF(2^k)$ is a polynomial of length $k$, that is of degree less than or equal to $(k-1)$, written as

$$a(x) = \sum_{i=0}^{k-1} a_i x^i \ ,$$

where the coefficients $a_i \in GF(2)$, i.e., $a_i \in \{0,1\}$.

As a shorthand to the polynomial representation, the element $a$ can be represented as the $k$-dimensional vector

$$a = (a_{k-1}a_{k-2}\cdots a_1 a_0) \ .$$

Software implementations are often based on the word-level representations of the field elements. So, assume that $k$ satisfies the inequality

$$sw \geq k > (s-1)w \ ,$$

where $w$ is the word-size which depends on the implementation details and the computer. In this case, partitioning the $k$-dimensional vector into $w$-bit blocks, the word-level representation of $a(x)$ becomes

$$(A_{s-1}A_{s-2}\cdots A_1 A_0) \ ,$$

where $A_i$ is of length $w$ for $i = 0, 1, \ldots, (s-1)$. These $A_i$ blocks are also represented as polynomials of degree $(w-1)$ using the notation $A_i(x)$. In hardware, a field element is stored in a shift register of length $k$.

In general, the field $F_{2^k}$ can be viewed as a vector space of dimension $k$ over $F_2$. Then, there exists a set of $k$ elements $\alpha_0, \alpha_1, \ldots, \alpha_{k-1}$ in $F_{2^k}$ such that each $\alpha \in F_{2^k}$ can be written uniquely in vector space representation as

$$\alpha = \sum_{i=0}^{k-1} a_i \alpha_i \qquad a_i \in \{0,1\} \ .$$

$\alpha$ can also be represented as the $0-1$ vector

$$(a_{k-1}, \ldots, a_1, a_0) \ .$$

A *normal basis* of $F_{2^k}$ over $F_2$ is a basis of the form

$$\{\beta, \beta^2, \beta^{2^2}, \ldots, \beta^{2^{k-1}}\} \ ,$$

where $\beta \in F_{2^k}$; it is well known that such a basis always exists [37]. Any element $\alpha \in F_{2^k}$ can be represented as

$$\alpha = \sum_{i=0}^{k-1} a_i \beta^{2^i}, \qquad a_i \in \{0, 1\} \ .$$

Unless otherwise noted, the first, i.e., the polynomial representation will be used. In order to simplify the analysis, we will assume $k = sw$ .

## 8.3 Arithmetic in Finite Fields

An irreducible polynomial $n(x)$ of order $k$ is needed to construct the finite field $F_{2^k}$, which is also called the Galois Field $GF(2^k)$.

The addition of two elements $a$ and $b$ in $GF(2^k)$ is performed by adding the polynomials $a(x)$ and $b(x)$, where the coefficients are added in $GF(2)$. This is equivalent to the bit-wise `XOR` operation on the vectors $a$ and $b$. The word-level addition is simply the bit-wise `XOR` operation on a pair of 1-word, i.e., 2 $w$-bit binary numbers, which is a readily available instruction on most general purpose microprocessors and signal processors. Assuming the processor can perform this word level `XOR` operation in one cycle, the computation of $c := a + b$ requires $s$ cycles. While analyzing the circuit delays in hardware implementations, however, $T_A$ and $T_X$ will be used to represent the delays of single 2-input `AND` and `XOR` gates, respectively.

During the multiplication of two elements $a$ and $b$ in $GF(2^k)$, reduction by the irreducible polynomial $n(x)$ is often required. The product $c = a \cdot b \in GF(2^k)$ is obtained by computing

$$c(x) = a(x)b(x) \pmod{n(x)} \ ,$$

where $c(x)$ is a polynomial of length $k$, representing the element $c \in GF(2^k)$. Thus, the multiplication operation in the field $GF(2^k)$ is accomplished by first multiplying

the input polynomials, and then performing a modular reduction using the generating polynomial $n(x)$.

The word-level multiplication operation receives two 1-word ($w$-bit) polynomials $A(x)$ and $B(x)$ defined over $GF(2)$, and computes the 2-word polynomial

$$C(x) = A(x)B(x) .$$

The degree of the product polynomial $C(x)$ is $2(w-1)$. For example, given $A = (1101)$ and $B = (1010)$, this operation computes $C$ as

$$A(x)B(x) = (x^3+x^2+1)(x^3+x) = x^6+x^5+x^4+x = (0111\ 0010) .$$

The implementation of this operation, which we call `MULGF2` as in [30], can be performed in three different ways:

1. An instruction implemented on the processor
2. The table lookup method
3. The emulation using the `XOR` and `SHIFT` operations

The details of the analysis of these methods can be found in [30]. The fastest is the first one, while the slowest is the last one. In our analysis, we will simply count the number of `MULGF2` operations, and assume that they are implemented using any of the above methods. A simple method for implementing the table lookup approach is to use two tables, one for computing the higher (`H`) and the other for computing the lower (`L`) bits of the product. The tables are addressed using the bits of the operands, and thus, the total size of these tables is of size $2 \times 2^w \times 2^w \times w$ bits. We store the values `H` and `L` in two table reads. Other approaches are also possible.

Since squaring is a linear operator in $F_{2^k}$, when using the normal basis representation, square of $\alpha$ becomes

$$\begin{aligned} \alpha^2 &= \sum_{i=0}^{k-1}\sum_{j=0}^{k-1} a_i a_j \beta^{2^i}\beta^{2^j} \\ &= \sum_{i=j=0}^{k-1} a_i a_j \beta^{2^i}\beta^{2^j} + 2 \sum_{k-1\geq i>j\geq 0} a_i a_j \beta^{2^i}\beta^{2^j} \end{aligned}$$

$$
\begin{aligned}
&= \sum_{i=0}^{k-1} a_i^2 \beta^{2^{i+1}} \\
&= \sum_{i=0}^{k-1} a_i \beta^{2^{i+1}} \\
&= \sum_{i=0}^{k-1} a_{i-1} \beta^{2^i} \\
&= (a_{k-2}, \ldots, a_0, a_{k-1}) \ ,
\end{aligned}
$$

with indices reduced modulo $m$. Hence a normal basis representation is advantageous because squaring a field element can be accomplished by a simple rotation of the vector representation, an operation that is easily implemented in hardware; thus squaring an element also takes one clock cycle.

Multiplication in a normal basis representation is more complicated. Let

$$A = (a_{k-1}, \ldots, a_1, a_0) \quad \text{and} \quad B = (b_{k-1}, \ldots, b_1, b_0)$$

be arbitrary elements in $F_{2^k}$, and let

$$C = A \cdot B = (c_{k-1}, \ldots, c_1, c_0) \ .$$

Then

$$c_l = \sum_{i=0}^{k-1} \sum_{j=0}^{k-1} a_i b_j \lambda_{i-l,j-l}^{(0)} = \sum_{i=0}^{k-1} \sum_{j=0}^{k-1} a_i b_j \lambda_{i,j}^{(l)} = \sum_{i=0}^{k-1} \sum_{j=0}^{k-1} a_{i+l} b_{j+l} \lambda_{i,j}^{(0)} \ ,$$

where $\lambda_{i,j}^{(l)} \in \{0, 1\}$ and is defined by

$$\beta^{2^i} \beta^{2^j} = \sum_{l=0}^{k-1} \lambda_{i,j}^{(l)} \beta^{2^l} \ .$$

They can be proven to satisfy

$$\lambda_{i,j}^{(l)} = \lambda_{i-l,j-l}^{(0)} \qquad \text{for all} \quad 0 \leq i, j, l \leq k-1 \ .$$

Hence if a logic circuit with inputs $A$ and $B$ is built to compute the product digit $c_0$, then the same circuit with inputs $A^{2^{-l}}$ and $B^{2^{-l}}$, which are simply the cyclic shifts of the vector representation of $A$ and $B$, yields the product digit $c_l$. In this way $C$ can be computed in $k$ clock cycles. Massey and Omura [53] constructed a serial-in serial-out multiplier to exploit this particular aspect of normal bases [46].

The complexity of such a circuit is determined by $C_N$, the number of non-zero terms $\lambda_{ij}^{(0)}$, since this quantity measures the number of interconnections between the registers containing $A$, $B$ and the product $C$. Clearly, $C_N \leq k^2$. A lower bound on $C_N$ is $C_N \geq 2k-1$ [51]. If $C_N = 2k-1$, then the normal basis is said to be *optimal*. *Optimal normal bases* were introduced and studied by Mullin, Onyszchuk, Vanstone and Wilson. The constructions together with a list of fields for which these bases exist, can be found in [51]. Gao and Lenstra proved in [13] that the optimal normal bases constructed in [51], are essentially all of the optimal normal bases. An associated architecture for a hardware implementation is given in [1]. Using this architecture a multiplication can be performed in $k$ clock cycles.

Finally, the most efficient technic, from the point of view of minimizing the number of multiplications, to compute an inverse of an element in $F_{2^k}$ was proposed by Itoh, Teechai and Tsujii [68]. Observe that if $\alpha \in F_{2^k}$, $\alpha \neq 0$, then

$$\alpha^{-1} = \alpha^{2^k-2} = (\alpha^{2^{k-1}-1})^2 \ .$$

If $k$ is odd, then

$$2^{k-1} - 1 = (2^{(k-1)/2} - 1)(2^{(k-1)/2} + 1)$$

yields

$$\alpha^{2^{k-1}-1} = (\alpha^{2^{(k-1)/2}-1})^{2^{(k-1)/2}+1} \ .$$

Hence it takes only one additional multiplication to evaluate $\alpha^{2^{k-1}-1}$ once the quantity $\alpha^{2^{(k-1)/2}-1}$ is known (ignoring the cost of squaring which is a shift operation). If $k$ is even, then

$$\alpha^{2^{k-1}-1} = \alpha^{2(2^{(k-1)/2}-1)(2^{(k-1)/2}+1)+1} \ ,$$

and consequently it takes two additional multiplications to evaluate $\alpha^{2^{k-1}-1}$ once the quantity $\alpha^{2^{(k-1)/2}-1}$ is known. The procedure is then repeated recursively. This method requires exactly

$$\lfloor \log_2(k-1) \rfloor + \omega(k-1) - 1$$

field multiplications, where $\omega(n)$ denotes the *Hamming weight* of $n$, i.e., the number of 1's in the binary representation of $n$ [46].

# Chapter 9
# A REDUCTION METHOD FOR MULTIPLICATION

In this chapter, a new table lookup based reduction method for performing the modular reduction operation is proposed. This method can be used to obtain fast software implementations of the finite field multiplication and squaring operations. The reduction algorithm has both left-to-right and right-to-left versions, which respectively improve the standard and Montgomery multiplication methods. Furthermore, it is shown that the right-to-left version of the proposed reduction method also works in the integer case, if the modulus $n$ is an odd number.

## 9.1 Introduction

The software implementation of the arithmetic operations in $GF(2^k)$ require that we design *word-level* algorithms for efficient implementation on current general purpose microprocessors. We propose a new table lookup based reduction (TLBR) method for performing the modular reduction operation, which can be used to obtain fast software implementations of the finite field multiplication and squaring operations. The proposed modular reduction method has three important properties:

- The method works for an arbitrary generating polynomial $n(x)$. It does not assume any special structure in $n(x)$, for example, the generating polynomial need not be a trinomial, a sparse polynomial, or an all-one-polynomial (AOP). However, in such cases, the method simplifies to certain modular reduction methods existing in the literature.

- The method provides word-level algorithms, enabling efficient software implementations of finite field multiplication and squaring operations.

- The method has both left-to-right and right-to-left versions, which are useful for performing both the standard and the Montgomery multiplications in finite fields.

Furthermore, it turns out that the right-to-left version of the TLBR method is also applicable to the integer case, giving a more efficient Montgomery multiplication algorithm. The left-to-right version, however, turns out to be inefficient in the integer case.

## 9.2 Table Lookup Based Reduction Algorithms

The proposed modular reduction methodology is a *table lookup based method*, which uses a table of the multiples of the generating polynomial $n(x)$, and performs a *word-level division*. We start with $n(x)$, which is a polynomial of degree $k$, and compute all multiples of $n(x)$ having degrees less than $k+w$. Consider the set of all polynomials over $GF(2)$ of length $w$ and the set of all $w$-bit integers as

$$\begin{aligned} Q_w &= \{1, x, x+1, x^2, x^2+1, x^2+x, \ldots, x^{w-1}+x^{w-2}+\cdots+1\} \ , \\ I_w &= \{0, 1, 2, \ldots, 2^w-1\} \ . \end{aligned}$$

Let $q_i(x)$ be the $i$th element of $Q_w$ and

$$v_i(x) = q_i(x)n(x)$$

for $i \in I_w$. The polynomial $v_i(x)$ is of degree less than $k+w$, which we represent as an $(s+1)$-word number

$$v_i(x) = (V_{i,s}V_{i,s-1}\cdots V_{i,1}V_{i,0}) \ .$$

We then construct the table $T_1$ containing $2^w$ rows, in which we store the polynomial $v_i(x)$ using its most significant word ($w$-bits) as the index, i.e.,

$$T_1(V_{i,s}) = (V_{i,s-1}\cdots V_{i,1}V_{i,0}) \ ,$$

for $i \in I_w$. An important observation is that the most significant words $V_{i,s}$ for $i \in I_w$ span the set $Q_w$, in other words, they are all unique.

**Theorem 9.1** *The most significant words $V_{i,s}$ are all unique for $i \in I_w$.*

**Proof** Assume $V_{i,s} = V_{j,s}$ for $i \neq j$. The polynomial

$$p(x) = v_i(x) + v_j(x)$$

has degree less than $k$, the degree of $p(x)$, since

$$\begin{aligned} p(x) &= (V_{i,s}V_{i,s-1} \cdots V_{i,1}V_{i,0}) + (V_{j,s}V_{j,s-1} \cdots V_{j,1}V_{j,0}) \\ &= (0P_{s-1} \cdots P_1P_0) \ . \end{aligned}$$

Furthermore, $p(x)$ is divisible by $n(x)$ since

$$p(x) = q_i(x)n(x) + q_j(x)n(x) = (q_i(x) + q_j(x))n(x) \ ,$$

which means $p(x)$ can only be the zero polynomial, i.e., $i = j$. □

The table $T_1$ will be used in the *left-to-right* TLBR algorithm for reducing polynomials modulo $n(x)$, i.e., to compute

$$p(x) \pmod{n(x)} \ .$$

In order to perform the Montgomery-type reduction, we use a *right-to-left* TLBR algorithm and obtain

$$x^{-k}p(x) \pmod{n(x)} \ .$$

This algorithm requires that we construct a table of multiples of $n(x)$ based on the least significant words. Similarly, we take the polynomial

$$v_i(x) = q_i(x)n(x)$$

for $i \in I_w$, and construct the table $T_2$ containing $2^w$ rows. The table $T_2$ keeps the polynomial $v_i(x)$, where we use the least significant word $V_{i,0}$ as the index, i.e.,

$$T_2(V_{i,0}) = (V_{i,s} \cdots V_{i,2}V_{i,1}) \ ,$$

The uniqueness of the least significant words $V_{i,0}$ depends on whether $n(x)$ is not divisible by $x$.

**Theorem 9.2** *The least significant words $V_{i,0}$ are all unique for $i \in I_w$ if and only if $n(x)$ is not divisible by $x$.*

**Proof** Assume $V_{i,0} = V_{j,0}$ for $i \neq j$. The polynomial

$$p(x) = x^{-w}(v_i(x) + v_j(x))$$

has degree less than $k$, the degree of $p(x)$, since

$$\begin{aligned} p(x) &= x^{-w}\left((V_{i,s}V_{i,s-1}\cdots V_{i,1}V_{i,0}) + (V_{j,s}V_{j,s-1}\cdots V_{j,1}V_{j,0})\right) \\ &= x^{-w}(P_s\cdots P_2P_10) \\ &= (P_s\cdots P_2P_1) \ . \end{aligned}$$

Furthermore, $p(x)$ is divisible by $n(x)$ if and only if

$$\gcd(x^w, n(x)) = 1$$

since

$$p(x) = x^{-w}(q_i(x)n(x) + q_j(x)n(x)) = x^{-w}(q_i(x) + q_j(x))n(x) \ .$$

Therefore, $p(x)$ can only be the zero polynomial, i.e., $i = j$. The condition

$$\gcd(x^w, n(x)) = 1$$

is satisfied if and only if $n(x)$ is not divisible by $x$. □

The tables $T_1$ and $T_2$ are used to reduce a polynomial of length $(sw + w)$ to a polynomial of length $sw$ using the irreducible polynomial $n(x)$. Let $p(x)$ be a polynomial of length $(sw + w)$, which is to be reduced, denoted as

$$p(x) = (P_sP_{s-1}\cdots P_1P_0) \ .$$

The left-to-right TLBR algorithm computes

$$p(x) \pmod{n(x)} \ ,$$

while the right-to-left TLBR algorithm computes

$$x^{-w}p(x) \pmod{n(x)} \ .$$

The resulting polynomial in both cases is of length $sw$.

**Left-To-Right TLBR Algorithm:** To reduce

$$p(x) = (P_s P_{s-1} \cdots P_1 P_0) \ ,$$

select the entry

$$(V_{s-1} V_{s-2} \cdots V_1 V_0)$$

from the table $T_1$ using the index $P_s = V_s$. Since $T_1$ was constructed so that the element

$$(P_s V_{s-1} V_{s-2} \cdots V_1 V_0)$$

resides in position $P_s$, we have

$$\begin{aligned} p(x) &:= (P_s P_{s-1} \cdots P_1 P_0) + (P_s V_{s-1} V_{s-2} \cdots V_1 V_0) \\ &:= (0 P'_{s-1} \cdots P'_1 P'_0) \ , \end{aligned}$$

where $P'_j = P_j + V_j$ for $j = 0, 1, \ldots, s-1$. We also discard the most significant $w$ bits of the new $p(x)$. Since we add a multiple of $n(x)$ to $p(x)$, and obtain a polynomial of length $sw$, we effectively compute

$$p(x) \pmod{n(x)} \ ,$$

as required. We denote the above computation as

$$p(x) := p(x) + T_1(P_s) \ .$$

**Right-To-Left TLBR Algorithm:** To reduce

$$p(x) = (P_s P_{s-1} \cdots P_1 P_0) \ ,$$

select the entry

$$(V_s V_{s-1} \cdots V_1)$$

from the table $T_2$ using the index $P_0 = V_0$. Since $T_2$ was constructed so that the element

$$(V_s V_{s-1} \cdots V_1 P_0)$$

resides in position $P_0$, we have

$$\begin{aligned} p(x) &:= (P_s P_{s-1} \cdots P_1 P_0) + (V_s V_{s-1} \cdots V_1 P_0) \\ &:= (P'_s P'_{s-1} \cdots P'_1 0) \ , \end{aligned}$$

where $P'_j = P_j + V_j$ for $j = 1, 2, \ldots, s$. We then shift the new $p(x)$, $w$ bits to the right, i.e., we multiply it by $x^{-w}$. Since we add a multiple of $n(x)$ to $p(x)$, and then, multiply it by $x^{-w}$ in order to obtain a polynomial of length $sw$, we effectively compute

$$x^{-w} p(x) \pmod{n(x)} \ ,$$

as required. We will denote the above computation as

$$p(x) := x^{-w}(p(x) + T_2(P_0)) \ .$$

## 9.3 Standard Multiplication Using TLBR Method

The standard multiplication algorithm computes

$$c(x) = a(x)b(x) \pmod{n(x)}$$

given $a(x)$, $b(x)$, and $n(x)$. In order to apply the TLBR method, we first construct the table $T_1$ using the generating polynomial $n(x)$. The algorithm then proceeds by multiplying one word of $a(x)$ by the entire $b(x)$, which is followed by a table lookup reduction to reduce the partial product. We will call this algorithm STDMUL, whose steps are given below:

Algorithm STDMUL

Step 0: Construct $T_1$ using $n(x)$ and $w$

Step 1. $c(x) := 0$

Step 2. for $i = s - 1$ downto 0 do

Step 3. $\quad c(x) := x^w c(x) + A_i(x)b(x)$

Step 4. $\quad c(x) := c(x) + T_1(C_s)$

Step 5. return $c(x)$

The operation in Step 4 of STDMUL is performed by first discarding the $s$th (the most significant) word of

$$c(x) = (C_s C_{s-1} \cdots C_1 C_0) \ ,$$

and then by adding the $s$-word number

$$T_1(C_s) = (V_{s-1} V_{s-2} \cdots V_1 V_0)$$

to the partial product $c(x)$ as

$$\begin{array}{cccccc} C_{s-1} & C_{i-2} & \cdots & C_1 & C_0 \\ V_{s-1} & V_{s-2} & \cdots & V_1 & V_0 \end{array}$$

Similarly, we perform the standard squaring operation using the TLBR method. This algorithm is denoted as STDSQU, whose steps are given below. An important saving in this case is that the cross product terms disappear because the ground field is $GF(2)$. Since

$$a^2(x) = \sum_{i=0}^{k-1} a_i x^{2i} \ = \ a_{k-1} x^{2(k-1)} + a_{k-2} x^{2(k-2)} + \cdots + a_1 x^2 + a_0 \ , \qquad (9.2)$$

the multiplication step (i.e., Step 3) in STDMUL can be skipped. The standard squaring algorithm, called STDSQU, starts with the degree $2(k-1)$ polynomial $c(x) = a^2(x)$ given by

$$c(x) = (a_{k-1} \mathbf{0} a_{k-2} \mathbf{0} \cdots \mathbf{0} a_1 \mathbf{0} a_0) \ ,$$

and then performs the reduction steps using the table $T_1$.

Algorithm STDSQU

Step 0: Construct $T_1$ using $n(x)$ and $w$

Step 1. $c(x) := \sum_{i=0}^{k-1} a_i x^{2i}$

Step 2. for $i = 2s - 1$ downto $s$ do

Step 3. $\quad c(x) := c(x) + T_1(C_i)$

Step 4. return $c(x)$

We perform the operation in Step 3 of STDSQU by first discarding the $i$th (the most significant) word of

$$c(x) = (C_i C_{i-1} \cdots C_1 C_0) \ ,$$

and then by adding the $s$-word number

$$T_1(C_i) = (V_{s-1} V_{s-2} \cdots V_1 V_0)$$

to the partial product $c(x)$ by aligning $V_{s-1}$ with $C_{i-1}$ from the left:

$$\begin{array}{cccccccc} C_{i-1} & C_{i-2} & \cdots & C_{i-s+1} & C_{i-s} & \cdots & C_1 & C_0 \\ V_{s-1} & V_{s-2} & \cdots & V_1 & V_0 & & & \end{array}$$

Thus, each addition operation in Step 3 requires exactly $s$ XOR operations.

## 9.4 Example : Standard Multiplication Using TLBR

We take the field $GF(2^8)$ to illustrate the construction of the table $T_1$, and also give an example of the standard multiplication operation using the TLBR method. We select the irreducible polynomial as

$$n(x) = x^8 + x^5 + x^3 + x^2 + 1 \ . \tag{9.3}$$

We also select $w = 4$, which gives $s = k/w = 8/4 = 2$. The table $T_1$ is constructed by taking a polynomial $q(x)$ from $Q_4$, multiplying it by $n(x)$ to obtain

$$v(x) = q(x)n(x) \ ,$$

and then placing the least significant $s = 2$ words of $v(x)$ to $T_1$ using the most significant word as the index. The step-by-step construction of $T_1$ is shown in Table 9.1. The multiples of $n(x)$ do not necessarily come in an increasing order, however, we have a complete set of most significant words, and thus, we can use these values as their indices to store them in $T_1$. The table $T_1$ is shown in Table 9.1 in its unsorted form. When sorted with respect to the most significant words, i.e. the $i$ values, we no longer need the first column.

**Table 9.1.** The construction of $T_1$ and $T_2$ for $n(x) = (0001\ 0010\ 1101)$.

| $q(x)$ | $v(x)$ |
|---|---|
| (0000) | (0000 0000 0000) |
| (0001) | (0001 0010 1101) |
| (0010) | (0010 0101 1010) |
| (0011) | (0011 0111 0111) |
| (0100) | (0100 1011 0100) |
| (0101) | (0101 1001 1001) |
| (0110) | (0110 1110 1110) |
| (0111) | (0111 1100 0011) |
| (1000) | (1001 0110 1000) |
| (1001) | (1000 0100 0101) |
| (1010) | (1011 0011 0010) |
| (1011) | (1010 0001 1111) |
| (1100) | (1101 1101 1100) |
| (1101) | (1100 1111 0001) |
| (1110) | (1111 1000 0110) |
| (1111) | (1110 1010 1011) |

| $i$ | $T_1(i)$ |
|---|---|
| (0000) | (0000 0000) |
| (0001) | (0010 1101) |
| (0010) | (0101 1010) |
| (0011) | (0111 0111) |
| (0100) | (1011 0100) |
| (0101) | (1001 1001) |
| (0110) | (1110 1110) |
| (0111) | (1100 0011) |
| (1001) | (0110 1000) |
| (1000) | (0100 0101) |
| (1011) | (0011 0010) |
| (1010) | (0001 1111) |
| (1101) | (1101 1100) |
| (1100) | (1111 0001) |
| (1110) | (1010 1011) |
| (1111) | (1000 0110) |

| $i$ | $T_2(i)$ |
|---|---|
| (0000) | (0000 0000) |
| (1101) | (0001 0010) |
| (1010) | (0010 0101) |
| (0111) | (0011 0111) |
| (0100) | (0100 1011) |
| (1001) | (0101 1001) |
| (1110) | (0110 1110) |
| (0011) | (0111 1100) |
| (1000) | (1001 0110) |
| (0101) | (1000 0100) |
| (0010) | (1011 0011) |
| (1111) | (1010 0001) |
| (1100) | (1101 1101) |
| (0001) | (1100 1111) |
| (0110) | (1111 1000) |
| (1011) | (1110 1010) |

Furthermore, we also give the table $T_2$ in Table 9.1, which is to be used by the right-to-left TLBR algorithm for computing the Montgomery multiplication and squaring operations in $GF(2^8)$. The table $T_2$ is constructed by placing the polynomial $v(x)$ in $T_2$ using its least significant word as the index. Again the table $T_2$ shown in Table 9.1 is in its unsorted form and we will get rid of the first column that includes the least significant words, i.e., the $i$ values, as soon as the rows of the table is sorted using them.

As an example for STDMUL, we take

$$\begin{array}{lllll} a(x) & = & x^7+x^6+x^4+x^3+x+1 & = & (1101\ 1011) & = & (A_1A_0) \\ b(x) & = & x^7+x^5+x^3+x^2+x & = & (1010\ 1110) & = & (B_1B_0) \end{array} \tag{9.4}$$

The algorithm starts with $c(x) = 0$ and then performs the following steps to find the result:

$$\begin{array}{llll} i=1 & \text{Step 3:} & c(x) := & c(x)x^4 + A_1(x)b(x) = (C_2C_1C_0) \\ & & = & 0 + (1101)(1010\ 1110) = (0111\ 0110\ 0110) \\ & \text{Step 4:} & c(x) := & c(x) + T_1(C_2) = (C_1C_0) + T_1(C_2) \\ & & = & (0110\ 0110) + (1100\ 0011) = (1010\ 0101) \\ i=0 & \text{Step 3:} & c(x) := & c(x)x^4 + A_0(x)b(x) = (C_2C_1C_0) \\ & & = & (1010\ 0101\ 0000) + (1011)(1010\ 1110) \\ & & = & (1110\ 1101\ 0010) \\ & \text{Step 4:} & c(x) := & c(x) + T_1(C_2) = (C_1C_0) + T_1(C_2) \\ & & = & (1010\ 1011) + (1101\ 0010) = (0111\ 1001) \end{array}$$

Therefore, the result is found as

$$c(x) = (0111\ 1001) = x^6 + x^5 + x^4 + x^3 + 1 \ .$$

## 9.5 Montgomery Multiplication Using TLBR Method

The Montgomery product of two elements $a(x)$ and $b(x)$ is defined as the computation of

$$c(x) = a(x)b(x)r^{-1}(x) \pmod{n(x)} \ ,$$

where $r(x)$ is a fixed element of the field [30]. It is established that when

$$r(x) = x^k \pmod{n(x)} \ ,$$

we can obtain efficient software implementations of the Montgomery multiplication operation in $GF(2^k)$. Thus in the sequel, we define the Montgomery multiplication of $a(x)$ and $b(x)$ as

$$x^{-k}a(x)b(x) \pmod{n(x)} \ .$$

Note that the inverse element

$$x^{-k} = (x^k)^{-1}$$

exists since $n(x)$ is an irreducible polynomial, and thus

$$\gcd(x^k, n(x)) = 1 \ .$$

The details of the Montgomery multiplication algorithm in $GF(2^k)$ and its properties are found in [30]. The algorithm requires that we compute $N_0'(x)$ in advance which is the least significant word of the polynomial $n'(x)$ defined as

$$n'(x) = -n(x)^{-1} \pmod{x^k} \ .$$

An algorithm for computing $N_0'(x)$ is also described in [30]. However, it turns out that the Montgomery multiplication method using the TLBR algorithm does not require the computation of $N_0'(x)$. This is not surprising since the table $T_2$ keeps all $2^w$ multiples of $n(x)$, and therefore, there is no need to separately compute $N_0'(x)$.

The steps of the Montgomery multiplication algorithm using the TLBR method are given below. The algorithm, which we call MONMUL, is based on the right-to-left TLBR algorithm described in Section 9.2.

Algorithm MONMUL

Step 0: Construct $T_2$ using $n(x)$ and $w$

Step 1. $c(x) := 0$

Step 2. for $i = 0$ to $s - 1$ do

Step 3. $\quad c(x) := c(x) + A_i(x)b(x)$

Step 4. $\quad c(x) := x^{-w}(c(x) + T_2(C_0))$

Step 5. return $c(x)$

At the end of Step 1, we have an $(s+1)$-word number

$$C = (C_s C_{s-1} \cdots C_1 C_0) \ .$$

We discard the 0th (the least significant) word $C_0$, and then add the $s$-word number

$$T_2(C_0) = (V_s V_{s-1} \cdots V_2 V_1)$$

to the partial product $c(x)$ by aligning $V_1$ with $C_1$ from the right:

$$\begin{array}{cccccc} C_s & C_{s-1} & \cdots & C_2 & C_1 \\ V_s & V_{s-1} & \cdots & V_2 & V_1 \end{array}$$

Thus, the 1-word shift operation denoted as the multiplication by $x^{-w}$ is implicitly performed. Each addition operation in Step 4 requires exactly $s$ XOR operations. Similarly, the Montgomery squaring method computes

$$c(x) = x^{-k}a^2(x) \pmod{n(x)} \ .$$

In order to compute $c(x)$, we first obtain $a^2(x)$ using the property (9.2), and then reduce the result with the help of the right-to-left TLBR algorithm, as seen below.

Algorithm MONSQU

Step 0: Construct $T_2$ using $n(x)$ and $w$

Step 1. $c(x) := \sum_{i=0}^{k-1} a_i x^{2i}$

Step 2. for $i = 0$ to $s - 1$ do

Step 3. $\quad c(x) := x^{-w}(c(x) + T_2(C_0))$

Step 4. return $c(x)$

When Step 1 completes, we have an $(2s-1)$-word number

$$C = (C_{2s-2}C_{2s-3}\cdots C_1C_0) \ .$$

In the beginning of the $i$th step, the number $c(x)$ is an $(2s-i-1)$-word number

$$C = (C_{2s-i-2}C_{2s-i-3}\cdots C_1C_0) \ .$$

We perform the operation in Step 3 of MONSQU by first discarding the 0th (the least significant) word of $c(x)$, and then by adding the $s$-word number

$$T_2(C_0) = (V_sV_{s-1}\cdots V_1)$$

to the partial product $c(x)$ by aligning $V_1$ with $C_1$ from right, as follows:

$$\begin{array}{ccccccccc} C_{2s-i-2} & C_{2s-i-1} & \cdots & C_s & C_{s-1} & \cdots & C_2 & C_1 \\ & & & V_s & V_{s-1} & \cdots & V_2 & V_1 \end{array}$$

As in the case for the Montgomery multiplication, the 1-word shift operation, i.e., the multiplication by $x^{-w}$, is implicitly performed, and each addition operation in Step 3 requires exactly $s$ `XOR` operations.

## 9.6 Example : Montgomery Multiplication Using TLBR

We take the field $GF(2^8)$ and the same irreducible polynomial $n(x)$ as the one exemplified in Section 9.4. The lookup table $T_2$ is exactly the same as the one in Table 9.1. In this case, however, we compute the Montgomery product of the same polynomials $a(x)$ and $b(x)$ given in Equation (9.4).

Therefore, we will be computing

$$c(x) = x^{-8}a(x)b(x) \pmod{n(x)} \ .$$

The right-to-left TLBR algorithm starts with

$$\begin{aligned} a(x) &= (A_1A_0) = (1101\ 1011) \\ b(x) &= (B_1B_0) = (1010\ 1110) \\ c(x) &= 0 \end{aligned}$$

and performs the following steps:

$$\begin{aligned} i=0 \quad \text{Step 3:} \quad c(x) &:= c(x) + A_0(x)b(x) = (C_2C_1C_0) \\ &= 0 + (1011)(1010\ 1110) = (0100\ 1000\ 0010) \\ \text{Step 4:} \quad c(x) &:= x^{-4}(c(x) + T_2(C_0)) = (C_2C_1) + T_2(C_0) \\ &= (0100\ 1000) + (1011\ 0011) = (1111\ 1011) \\ i=1 \quad \text{Step 3:} \quad c(x) &:= c(x) + A_1(x)b(x) = (C_2C_1C_0) \\ &= (1111\ 1011) + (1101)(1010\ 1110) = (0111\ 1001\ 1101) \\ \text{Step 4:} \quad c(x) &:= (x^{-4}c(x) + T_2(C_0)) = (C_2C_1) + T_2(C_0) \\ &= (0111\ 1001) + (0001\ 0010) = (0110\ 1011) \end{aligned}$$

The product is found as

$$c(x) = (0110\ 1011) = x^6 + x^5 + x^3 + x + 1 \ .$$

**Table 9.2.** The size of the table in bytes for the TABLEREAD operation.

| $k$ | $w = 4$ | $w = 8$ | $w = 10$ | $w = 16$ |
|---|---|---|---|---|
| 160 | 320 | 5,120 | 20,480 | 1,310,720 |
| 256 | 512 | 8,192 | 32,768 | 2,097,152 |
| 512 | 1,024 | 16,384 | 65,536 | 4,194,304 |
| 1024 | 2,048 | 32,768 | 131,072 | 8,388,608 |

## 9.7 Analyses of the Algorithms

In this section, we analyze the standard and Montgomery multiplication algorithms by calculating the size of the lookup tables and counting the total number of the table read and the word-level $GF(2)$ addition and multiplication operations. The implementation details of the word-level operations were studied in Section 8.3 before. We will denote the table read operation using TABLEREAD, and count the total number of TABLEREAD operations. In regards to the sizes of these tables, we note that the table $T_1$ (or $T_2$) has $2^w$ rows, each of which contains a polynomial of length $k$. This implies that the size of the tables is $2^w \times k$ bits. The space requirements for the tables $T_1$ (or $T_2$) for performing the TABLEREAD operation are exemplified in Table 9.2.

For example, if $w = 8$ and $k = 160$, the size of the table is $2^8 \times 20 = 5,120$ bytes, which is quite reasonable. However, the table size becomes excessive as we increase the word-size. For a fixed field size $k$, we can decide about the word-size $w$ given the memory capacity of the computer system.

Now, we give the steps of the algorithms STDMUL and MONMUL in detail in Table 9.3, together with the number of TABLEREAD, MULGF2, and XOR operations.

The total number of operations for STDMUL, STDSQU, MONMUL, MONSQU are summarized in Table 9.4. Furthermore, in Table 9.5, we give the coefficients of the highest term $s^2$ in the operation counts of the multiplication algorithms using the

**Table 9.3.** Operation counts for STDMUL and MONMUL algorithms.

| STDMUL | TABLEREAD | MULGF2 | XOR |
|---|---|---|---|
| `for i=0 to s do` | - | - | - |
| `  C[i]:=0` | - | - | - |
| `for i=s-1 downto 0 do` | - | - | - |
| `  P:=0` | - | - | - |
| `  for j=s-1 downto 0 do` | - | - | - |
| `    (H,L):=MULGF2(A[i],B[j])` | - | $s^2$ | - |
| `    C[j+1]:=C[j] XOR H XOR P` | - | - | $2s^2$ |
| `    P:=L` | - | - | - |
| `  C[0]:=P` | - | - | - |
| `  for j=0 to s-1 do` | - | - | - |
| `    C[j]:=C[j] XOR T[C[s]][j]` | $s$ | - | $s^2$ |

| MONMUL | TABLEREAD | MULGF2 | XOR |
|---|---|---|---|
| `for i=0 to s do` | - | - | - |
| `  C[i] := 0` | - | - | - |
| `for i=0 to s-1 do` | - | - | - |
| `  P := 0` | - | - | - |
| `  for j=0 to s-1 do` | - | - | - |
| `    (H,L):=MULGF2(A[i],B[j])` | - | $s^2$ | - |
| `    C[j]:=C[j] XOR L XOR P` | - | - | $2s^2$ |
| `    P := H` | - | - | - |
| `  C[s] := P` | - | - | - |
| `  for j=0 to s-1 do` | - | - | - |
| `    C[j] := C[j+1] XOR T[C[0]][j]` | $s$ | - | $s^2$ |

**Table 9.4.** The operation counts for the algorithms.

| | TABLEREAD | MULGF2 | XOR | SHIFT |
|---|---|---|---|---|
| STDMUL | $s$ | $s^2$ | $3s^2$ | - |
| STDSQU | $s$ | - | $s^2$ | - |
| MONMUL | $s$ | $s^2$ | $3s^2$ | - |
| MONSQU | $s$ | - | $s^2$ | - |

**Table 9.5.** The operation count orders for the algorithms.

| Using this method | MULGF2 | XOR | SHIFT |
|---|---|---|---|
| STDMUL | 1 | 3 | 0 |
| STDSQU | 0 | 1 | 0 |
| MONMUL | 1 | 3 | 0 |
| MONSQU | 0 | 1 | 0 |

| Using the method in [30] | MULGF2 | XOR | SHIFT |
|---|---|---|---|
| STDMUL | 1 | $\frac{3w}{2}+3$ | $2(w+1)$ |
| STDSQU | 0 | $\frac{9w}{4}$ | $3w$ |
| MONMUL | 2 | 4 | 0 |
| MONSQU | 1 | 2 | 0 |

proposed TLBR method, comparing them to those algorithms which do not utilize the TLBR method. The detailed timing requirements of the algorithms without the TLBR method are given in [30].

## 9.8 Special Cases

The presented TLBR methods and the resulting multiplication algorithms work for an arbitrary generating polynomial. The tables $T_1$ (or $T_2$) are constructed and used without assuming a special structure in $n(x)$. This is an important property of the TLBR method, making it applicable for any value of $k$ and for arbitrary generating polynomials. However, it may be possible to avoid the construction of the table or to reduce the size of it or to reduce the time taken by the algorithm when the generating polynomial has a special structure. In this section we consider several different forms of irreducible polynomials for generating the field $GF(2^k)$.

We start with the case in which the generating polynomial $n(x)$ is of the form

$$n(x) = x^k + a_j x^j + a_{j-1}x^{j-1} + \cdots + a_1 x + a_0 \ ,$$

where $j = tw$ and $t < s$ is an integer. This implies that $x^k$ is the only nonzero term among the $(s-t)w$ most significant terms of $n(x)$, and thus, there is no need to prepare the whole table $T_1$ as the multiples of $n(x)$ will have all zeros in the most significant $s-t$ words, except the most significant word which is used as an index. It suffices to store the least significant $t$ words of the multiples of $n(x)$ to reduce the partial product. Therefore, the size of the table $T_1$ will be $2^w \times t \times w$ instead of $2^w \times s \times w$, i.e., the table size reduces linearly depending on the value of $t$. Furthermore, we only add the $t$ least significant words of the operands during the addition operation since we know the remaining $s-t$ words are zero. Therefore, Step 4 of STDMUL is performed by first discarding the $s$th (the most significant) word of

$$c(x) = (C_s C_{s-1} \cdots C_1 C_0)$$

and then by adding the $t$-word number

$$T_1(C_s) = (V_{t-1} V_{t-2} \cdots V_1 V_0)$$

to the partial product as

$$\begin{array}{cccccccc} C_{s-1} & C_{s-2} & \cdots & C_t & C_{t-1} & \cdots & C_1 & C_0 \\ & & & & V_{t-1} & \cdots & V_1 & V_0 \end{array}$$

The most significant $(s-t)$ words are not added since $(V_{s-1}V_{s-2}\cdots V_t)$ is known to be all zero. This operation requires $t$ word-level XOR operations in the reduction step (Step 4) instead of $s$.

### 9.8.1 Trinomials

When the irreducible polynomial is a *trinomial* of the form

$$n(x) = x^k + x^j + 1 \ ,$$

where $1 \leq j < k$, then, it turns out that there is no need to prepare the table $T_1$ (or $T_2$) and thus we will drop the phrase TLB from the names of the reduction algorithms. We can use the most significant (or the least significant) word of the partial product in order to reduce it. There are different approaches, depending on the value of $j$. If $j$ is an integer multiple of the word size $w$, i.e., $j = tw$, we can use the word-level shifts of the partial product. If $j$ is not an integer multiple of $w$, i.e., $j = tw + u$ for some $1 \leq u < w$, then, certain bit-level operations will need to be performed.

For $j = tw$, the left-to-right reduction algorithm reduces the $(s+1)$-word partial product

$$c(x) = (C_s C_{s-1} \cdots C_1 C_0)$$

by adding $C_s$ multiple of the irreducible polynomial $n(x)$ to it as

$$c(x) := c(x) + (x^{sw} + x^{tw} + 1)C_s \ .$$

This implies that we need to add $C_s$ to $C_s$, $C_t$, and $C_0$ in Step 4 of the algorithm STDMUL. However, we do not perform the first addition $C_s + C_s = 0$, as follows:

$$\begin{array}{cccccccc} C_{s-1} & \cdots & C_{t+1} & C_t & \cdots & C_1 & C_0 \\ & & & C_s & & & C_s \end{array}$$

Thus, during the $i$th step of the reduction, we perform 2 XOR operations. The multiplication algorithms described in [64, 72] are essentially the same.

The right-to-left reduction algorithm, on the other hand, uses $C_0$ to reduce the partial product from the right (the least significant). Effectively, it performs the operation

$$c(x) := c(x) + (x^{sw} + x^{tw} + 1)C_0 \ ,$$

in order to reduce the partial product 1-word from the right. This implies that we add $C_0$ to $C_0$, $C_t$, and $C_s$ as follows:

$$\begin{array}{cccccccc} C_s & C_{s-1} & \cdots & C_{t+1} & C_t & \cdots & C_1 \\ C_0 & & & & C_0 & & \end{array}$$

Similarly, we do not perform the addition of the least significant words $C_0 + C_0 = 0$, and obtain the $s$-word partial product using only 2 XOR operations.

If $j$ is not a multiple of $w$, but, say $j = tw + u$ for a positive integer $1 \leq u < w$, we need to perform the operation

$$c(x) := c(x) + (x^{sw} + x^{tw+u} + 1)C_s$$

to reduce the most significant word of $C_s$ of $c(x)$. This implies that $C_s$ is added to $C_s$ and $C_0$, which takes care of the part $c(x) + (x^{sw} + 1)C_s$. In order to add $x^{tw}(x^u C_s)$ to the partial product, $C_s$ needs to be shifted $u$ bits to left, which produces a 2-word number $(V_1 V_0)$. Let

$$C_s = (c_{k+w-1} \cdots c_{k+1} c_k) \ ,$$

then, $(V_1 V_0)$ is obtained as

$$(V_1)\ (V_0) = (0 \cdots 0\ c_{k+w-1} c_{k+w-2} \cdots c_{k+w-u})\ (c_{k+w-u-1} \cdots c_{k+1} c_k\ 0 \cdots 0) \ .$$

We then add $V_1$ and $V_2$ to $C_{t+1}$ and $C_t$, respectively. The operations required to reduce $c(x)$ using the left-to-right reduction algorithm are shown below:

$$\begin{array}{cccccccc} C_{s-1} & \cdots & C_{t+1} & C_t & \cdots & C_1 & C_0 \\ & & V_1 & V_0 & & & C_s \end{array}$$

Similarly, the addition of the most significant words $C_s + C_s = 0$ is ignored. In summary, the reduction operation during the $i$th step requires a few bit operations to obtain $V_1$ and $V_0$, and then 3 word-level XOR operations.

On the other hand, the right-to-left reduction algorithm performs the operation

$$c(x) := c(x) + (x^{sw} + x^{tw+u} + 1)C_0$$

in order to reduce the least significant word of $C_0$ of $c(x)$. This implies that we add $C_0$ to $C_s$ and $C_0$, taking care of the part $c(x) + (x^{sw} + 1)C_0$. In order to perform the operation

$$c(x) := c(x) + x^{tw}(x^u C_0) \ ,$$

we shift $C_0$ to the left $u$ times, obtaining a 2-word number $(V_1'V_0')$ as before. We then add $V_1'$ and $V_0'$ to $C_{t+1}$ and $C_t$, respectively. The final reduction operation is

$$\begin{array}{cccccccc} C_s & \cdots & C_{t+1} & C_t & \cdots & C_2 & C_1 \\ C_0 & & V_1' & V_0' & & & \end{array}$$

The addition of the least significant words $C_0 + C_0 = 0$ is ignored. The right-to-left reduction algorithm requires a few bit operations to compute $V_1'$ and $V_0'$, followed by 3 word-level XOR operations.

### 9.8.2 All-One-Polynomials

In this case, the generating polynomial $n(x)$ will be an *all-one-polynomial* (AOP), i.e., of the form

$$n(x) = x^k + x^{k-1} + \cdots + x + 1 \ .$$

It is known that an AOP is irreducible if and only if $k+1$ is prime and 2 is primitive modulo $k+1$ [45]. For $k \leq 100$, the AOP is irreducible for the following values of $k$: 2, 4, 10, 12, 18, 28, 36, 52, 58, 60, 66, 82, and 100.

The reduction is often performed using the polynomial $(x+1)n(x) = x^{k+1} + 1$, thus as in the trinomial case we do not need the tables and we will drop the phrase TLB from the names of the reduction algorithms. Since $k+1$ is not an integer

multiple of $w$, the exact word-level shifting is not possible with this polynomial. In this case, we need to perform the operation on the $(s+1)$-word partial product $c(x)$

$$c(x) := c(x) + (x^{sw+1} + 1)A$$

where $A$ is a 1-word number obtained from $C_s$, as we will explain shortly. Since the most significant word of the new $c(x)$ needs to be zero, we obtain

$$C_s + (Ax) = 0 \ ,$$

and therefore, $Ax = C_s$. If the least significant bit of $C_s$, denoted as $c_k$, is equal to zero, the computation of $A$ is very simple: $A = C_s/x$, i.e., $C_s$ is shifted 1 bit to the right to obtain $A$. Thus, $A$ is actually an $(w-1)$-bit number, and when $Ax$ is added to $C_s$, the result is zero. The final reduction is performed using only one `XOR` operation as

$$\begin{array}{cccccc} C_{s-1} & C_{s-2} & \cdots & C_1 & C_0 \\ & & & & A \end{array}$$

However, when $c_k$ is not equal to zero, we have no other choice except to add the entire $n(x)$ to $c(x)$. This implies that we add the $w$-bit AOP

$$2^w - 1 = (11\cdots1) = 1_w$$

to each word of $c(x)$ starting from 0 ending at $s-1$, as

$$\begin{array}{cccccc} C_{s-1} & C_{s-2} & \cdots & C_1 & C_0 \\ 1_w & 1_w & \cdots & 1_w & 1_w \end{array}$$

This operation makes the least significant bit $c_k$ zero. Therefore, if $c_k$ is nonzero, we need to perform an additional $s$ `XOR` operations to reduce the $(s+1)$-word partial product $c(x)$.

On the other hand, the right-to-left reduction algorithm performs

$$c(x) := c(x) + (x^{sw+1} + 1)C_0$$

in order to reduce the least significant word $C_0$ of $c(x)$. When $C_0$ is added to $c(x)$, the new least significant word $C_0$ will be zero. However, we also need to

add $x^{sw+1}C_0$ to $c(x)$. If the most significant bit $c_{w-1}$ of $C_0$ is zero, this operation is easily accomplished. We compute $A_0 = C_0 x$, and since $c_{w-1}$ is zero, $A_0$ fits into $w$ bits, i.e., 1-word. The following operation accomplishes the reduction:

$$\begin{array}{cccc} C_s & C_{s-1} & \cdots & C_1 \\ A_0 & & & \end{array}$$

If $c_{w-1}$ is not zero, then $A_0$ is no longer a 1-word number: it is a $(w+1)$-bit number containing a one its $w$th position:

$$A_0 = (1 c_{w-1} c_{w-2} \cdots c_0 0) \ .$$

The reduced partial product in this case becomes

$$c(x) := (1 \ C_s C_{s-1} \cdots C_2 C_1) \ .$$

As in the case for the left-to-right reduction algorithm, we have no other choice except to add the entire AOP $n(x)$ in order to reduce the most significant bit:

$$\begin{array}{cccccc} C_s & C_{s-1} & \cdots & C_2 & C_1 \\ 1_w & 1_w & \cdots & 1_w & 1_w \end{array}$$

Therefore, when $c_{w-1}$, we need to perform an additional $s$ XOR operations to reduce the $(s+1)$-word partial product $c(x)$.

## 9.9 Integer Case

In this section we consider the extension of the TLBR method to the integers. We are interested in reducing the $(s+1)$-word integer $a$ modulo $n$, where $n$ is an arbitrary integer of length $s$ words. In the following, we show that the right-to-left TLBR algorithm works if and only if $n$ is odd. On the other hand, the left-to-right TLBR algorithm works for

$$n > \frac{2^{k+w}}{(2^w + 1)} \ ,$$

but it is inefficient.

First we concentrate on the right-to-left TLBR algorithm. Let $i \in I_w$ and $v_i = in$. Since $n$ is an $s$-word number and

$$0 \leq i \leq 2^w - 1 \ ,$$

the number $v_i$ for all $i$ is an $(s+1)$-word number, represented as

$$v_i = (V_{i,s} V_{i,s-1} \cdots V_{i,1} V_{i,0}) \ .$$

The table $T_2$ is constructed by the right-to-left TLBR algorithm using the least significant words $V_{i,0}$ as

$$T_1(-V_{i,0}) = (V_{i,s} V_{i,s-1} \cdots V_{i,1}) \ .$$

We note the minus sign in the index, which helps us to add the table entry to the partial product, instead of subtracting it.

An important requirement is that all $V_{i,s}$ be unique for $i \in I_w$. This way we construct the complete table of the multiples of $n$ to be added to the partial product. The uniqueness of the least significant words depends on whether $n$ is odd, which is easily proven as follows.

**Theorem 9.3** *The least significant words of $v_i = in$ are unique for $i \in I_w$ if and only if $n$ is odd.*

**Proof** The least significant word of $v_i$ is given as $in \pmod{2^w}$. The set of residues $in \pmod{2^w}$ for $i \in I_w$ is complete if and only if

$$\gcd(n, 2^w) = 1 \ .$$

This is satisfied when $n$ is odd. □

Therefore, the right-to-left TLBR algorithm used in the Montgomery multiplication and squaring algorithms works in the integer case for an odd $n$ only. This gives us a table lookup based Montgomery multiplication algorithm, which is more efficient than the regular Montgomery multiplication since the reduction step is significantly simplified. Furthermore, there is no need to compute $n_0' = -n_0^{-1}$.

**Table 9.6.** Multiples of $n = 35 = (100\ 011)$.

| $i$ | $v_i = in$ |
|---|---|
| 001 | 000 100 011 |
| 010 | 001 000 110 |
| 011 | 001 101 001 |
| 100 | 010 001 100 |
| 101 | 010 101 111 |
| 110 | 011 010 010 |
| 111 | 011 110 101 |

Unfortunately, the left-to-right TLBR algorithm does not work as well. It is impractical for two main reasons:

1. The most significant words are not always unique,
2. Even when they are unique, the left-to-right TLBR algorithm cannot use addition in place of subtraction since this will cause a carry to higher order bits.

As an example, we take $w = 3$ and $n = 35 = (100\ 011)$, and produce the values $v_i = in$ as shown in Table 9.6.

An inspection of Table 9.6 shows that the most significant words are not unique for $n = 35$, for example, 001, 010, and 011 appear twice, while 100, 101, 110, and 111 do not appear. We prove below that collisions occur if $n < 2^{k+w}/(2^w + 1)$.

**Theorem 9.4** *If*

$$n < \frac{2^{k+w}}{(2^w + 1)} \ ,$$

*then there is always a collision, i.e., there are at least two equal most significant words. Otherwise, the most significant words are unique.*

**Proof** We consider all multiples with $s$ words. Let $I$ be the integer such that

$$In < 2^{k+w} \leq (I + 1)n \ ,$$

i.e., $In$ is the greatest multiple of $n$, that has $(s+1)$ words. Note that the difference of any two consecutive multiples is $n$, which is less than $2^k$, and thus, the difference of the most significant words of the consecutive multiples can be at most one. In particular, $In$ has $2^w - 1 = 1 \cdots 1$ as the most significant word. Thus, the most significant words form a monotone increasing sequence. As the largest one is $1 \cdots 1$, all possible single words exist among them. Therefore, we have $I \geq 2^w$. When $I = 2^w$, we have uniqueness and when $I > 2^w$ we have collision(s).

For

$$n < \frac{2^{k+w}}{(2^w + 1)} ,$$

using the definition of $I$, we have

$$2^{k+w} \leq (I+1)n < (I+1)\frac{(2^{k+w})}{(2^w+1)} ,$$

which yields $2^w < I$, i.e., we always have a collision. For

$$n \geq \frac{2^{k+w}}{(2^w + 1)} ,$$

again by the definition of $I$, we have

$$2^{k+w} > I \cdot n \geq \frac{2^{k+w}}{(2^w+1)I} ,$$

which yields $2^w + 1 > I$, i.e., we have no collision. □

During the reduction, we need to make sure that the number we subtract is less than the partial product. This can be done by comparison, and if the selected multiple is larger, we can use the previous multiple of $n$ instead. If the most significant words of $v_i$s are not unique, i.e., if there are collisions, then we have the problem of choosing the correct multiple. We can solve this problem by assigning the smallest multiple for all of the of the indices at which the collision occurs, which will reduce the probability of a negative result. For example, for the most significant word 001, we can assign the smaller multiple which is 001 000 110. Similarly, we can use 010 001 100 and 011 010 010, for the most significant words 010 and 011, respectively. However, in order to reduce the products with the most significant

words not in our list, we will have to either assign the largest multiple 011 110 101 to all of them or continue to produce more multiples until we get all possible most significant words. Note that, even after using this method, we still have to compare the numbers and use the previous multiple if necessary. In short, when there is a collision, the proposed table lookup based approach is not practical.

# Chapter 10
# PARALLEL FINITE FIELD MULTIPLICATION USING POLYNOMIAL RESIDUE NUMBER SYSTEMS

## 10.1 Introduction

In this chapter a novel method of parallelization of the multiplication operation in $GF(2^k)$ is presented for an arbitrary value of $k$ and arbitrary irreducible polynomial $n(x)$ generating the field. The parallel algorithm is based on the Polynomial Residue Number System. The parallel algorithm receives the residue representations of the input operands (elements of the field) and produces the result in its residue form, however, it is guaranteed that the degree of this polynomial is less than $k$, i.e., it is an element of the field, properly reduced by the irreducible polynomial $n(x)$. By proper selection of the modulus polynomials, and the application of the Chinese Remainder Theorem, yields an efficient parallel algorithm.

## 10.2 Polynomial Residue Number Systems

Let

$$m_1(x), \quad m_2(x), \quad \ldots \quad , \quad m_L(x)$$

be a list of pairwise relatively prime polynomials such that the degree of $m_i(x)$ is equal to $d_i$ for $i = 1, 2, \ldots, L$. We choose $m_i(x)$ such that each $d_i$ is approximately equal to $2k/L$, and thus, the product polynomial

$$M(x) = \prod_{i=1}^{L} m_i(x)$$

is of degree $d \geq 2k$. A polynomial $p(x)$ is represented in the Polynomial Residue Number Systems (PRNS) with a list of remainders:

$$\vec{p} = (p_1, p_2, \ldots, p_L) \ ,$$

where

$$p_i(x) = p(x) \pmod{m_i(x)}$$

for $i = 1, 2, \ldots, L$. For efficiency reasons, we select each $m_i(x)$ so that $p_i(x)$ is represented using at most $w$ bits or 1 word. This implies that $\deg(p_i(x)) = d_i < w$. Furthermore, the polynomials $m_i(x)$ need to be pairwise relatively prime, i.e., $\gcd(m_i, m_j) = 1$ for $i \neq j$. Therefore, we construct the PRNS by finding $L$ pairwise relatively prime polynomials $m_i(x)$, each of which is of degree $w$, such that the degree of $M(x)$ is $Lw \geq 2k$. Since $k = sw$, we have $L \geq 2s$. The reason for choosing the PRNS range as twice the size of the inputs is that we need to represent the product of two operands (or the square of one operand) uniquely.

Once the PRNS is constructed by proper selection of the $L$ such polynomials, we can perform the PRNS arithmetic. The residue addition and multiplication operations in the PRNS are defined as follows:

- $\vec{c} := \vec{a} + \vec{b}$ represents the residue addition:

$$c_i := a_i + b_i \pmod{m_i} \qquad i = 1, 2, \ldots, n \ .$$

- $\vec{c} := \vec{a} * \vec{b}$ represents the residue multiplication:

$$c_i := a_i \cdot b_i \pmod{m_i} \qquad i = 1, 2, \ldots, n \ .$$

If an operand is not a vector but a single polynomial, e.g., $a$, (which is of length $w$), then we will assume that it is a vector with all entries equal to $a$ as $(a, a, \ldots, a)$. For example, $\vec{c} = a * \vec{b}$ implies that

$$(c_1, c_2, \ldots, c_L) = (a, a, \ldots, a) * (b_1, b_2, \ldots, b_L) \ ,$$

or in other words,

$$c_i = a \cdot b_i \pmod{m_i} \ .$$

The conversion from the PRNS representation to the polynomial representation is based on the extension of the Chinese Remainder Theorem (*see* Appendix C) to the polynomials [25]. Given the PRNS representation of $p(x)$ as

$$\vec{p} = (p_1, p_2, \ldots, p_L) \ ,$$

we use the Single Radix Conversion (SRC) algorithm to compute $p(x)$ using

$$p(x) = \left[ \sum_{i=1}^{L} p_i \cdot \left( M_i^{-1} \mod m_i \right) \cdot M_i \right] \mod M(x) \ ,$$

where

$$M_i(x) = \frac{M(x)}{m_i(x)} = m_1(x)m_2(x) \cdots m_{i-1}(x)m_{i+1}(x) \cdots m_L(x) \ ,$$

and, the inverse $M_i^{-1} \pmod{m_i}$ is defined as

$$M_i^{-1} \cdot M_i = 1 \pmod{m_i} \ ,$$

which exists since $\gcd(M_i, m_i) = 1$ and all $m_i$s are pairwise relatively prime. We will assume that the coefficients $M_i$ for $i = 1, 2, \ldots, L$, and the inverse vector

$$\vec{I} = (M_1^{-1} \bmod m_1,\ M_2^{-1} \bmod m_2,\ \ldots, M_L^{-1} \bmod m_L)$$

are precomputed and used in the SRC algorithm. Here the polynomials

$$M_i(x) = M(x)/m_i(x)$$

are of degree $(L-1)w$. The entries of $\vec{I}$ are reduced, i.e., the degree of $M_i^{-1} \pmod{m_i}$ is less than the degree of $m_i$. Using these definitions, we give the SRC algorithm which computes the polynomial $p(x)$ given its residue representation $\vec{p}$ as follows:

THE SRC ALGORITHM

| | |
|---|---|
| Input: | $\vec{p} = (p_1, p_2, \ldots, p_L)$ |
| Output: | $p(x) \pmod{M}(x)$ |
| Auxiliary: | $M_1, M_2, \ldots, M_L$ and $\vec{I}$ |
| Step 1. | $\vec{r} := \vec{p} * \vec{I}$ |
| Step 2. | $p(x) := \sum_{i=1}^{L} r_i \cdot M_i \mod M(x)$ |
| Step 3. | return $p(x)$ |

## 10.3 Finite Field Multiplication Using the PRNS

If

$$s(x) = a(x) + b(x) \ ,$$

then the degree of $s(x)$ is not larger than the maximum of the degrees $of a(x)$ and $b(x)$, thus, the PRNS arithmetic would yield the exact result, i.e.,

$$\vec{s} = \vec{a} + \vec{b} \ .$$

However, in multiplication

$$p(x) = a(x)b(x) \ ,$$

the degree of the resulting polynomial increases. The polynomial $p(x)$ needs to be reduced modulo the irreducible polynomial $n(x)$ in order to obtain the product $c = a \cdot b$ in $GF(2^k)$. Therefore, if we want to use the PRNS for multiplication modulo $n(x)$, we need to devise a method to reduce the resulting polynomial modulo $n(x)$. We will perform this reduction using the table lookup reduction algorithm proposed in [19]. This algorithm precomputes the multiples of the irreducible polynomial $n(x)$, and stores them in the table $T$ in such a way that a word-level reduction becomes possible using these stored values. We start with $n(x)$, which is a degree-$k$ polynomial, and compute all multiples of $n(x)$, which has at most $k+w$ bits (or $s+1$ words). Consider the set of all polynomials over $GF(2)$ of length $w$ and the set of all $w$-bit integers:

$$\begin{aligned} Q_w &= \{1, x, x+1, x^2, x^2+1, x^2+x, \ldots, x^{w-1}+x^{w-2}+\cdots+1\} \ , \\ I_w &= \{0, 1, 2, \ldots, 2^w - 1\} \ . \end{aligned}$$

Let $q_i(x)$ be the $i$th element of $Q_w$ and

$$v_i(x) = q_i(x)n(x)$$

for $i \in I_w$. The polynomial $v_i(x)$ is of degree less than $(k+w)$, which we represent as an $(s+1)$-word number

$$v_i(x) = (V_{i,s}V_{i,s-1}\cdots V_{i,1}V_{i,0}) \ .$$

We then construct the table $T$ containing $2^w$ rows, in which we store the polynomial $v_i(x)$ using its most significant word ($w$-bits) as the index, i.e.,

$$T[V_{i,s}] = (V_{i,s-1}\cdots V_{i,1}V_{i,0}) \ ,$$

for $i \in I_w$. An important observation is that the most significant words $V_{i,s}$ for $i \in I_w$ span the set $Q_w$, in other words, they are all unique [19]. We use the *left-to-right* reduction algorithm. In order to reduce

$$p(x) = (P_s P_{s-1} \cdots P_1 P_0) \ ,$$

we select the entry

$$(V_{s-1} V_{s-2} \cdots V_1 V_0)$$

from the table $T$ using the index $P_s = V_s$. Since $T$ was constructed so that the element

$$(P_s V_{s-1} V_{s-2} \cdots V_1 V_0)$$

resides in position $P_s$, we have

$$\begin{aligned} p(x) &:= p(x) + T[P_s] \\ &= (P_s P_{s-1} \cdots P_1 P_0) + (P_s V_{s-1} V_{s-2} \cdots V_1 V_0) \\ &= (0 P'_{s-1} \cdots P'_1 P'_0) \ , \end{aligned}$$

where $P'_j = P_j + V_j$ for $j = 0, 1, \ldots, s-1$. We also discard the most significant $w$ bits of the new $p(x)$. Since we add a multiple of $n(x)$ to $p(x)$, and obtain a polynomial of length $sw$, we effectively compute $p(x) \pmod{n(x)}$, as required.

Furthermore, we compute the multiples of $n(x)$ in the PRNS representation and store them similarly in the table $\vec{T}$, which has $2^w$ rows and $L$ independent columns, where each column contains a 1-word number at each row. If the PRNS representation of

$$(V_s V_{s-1} \ldots V_1 V_0)$$

is given as

$$\vec{v} = (v_1, v_2, \ldots, v_L) \ ,$$

then the row $V_s$ of the table $\vec{T}$ holds $v_i$ in column $i$ for $i = 1, 2, \ldots, L$, i.e.,

$$\vec{T}[V_s] = (v_1, v_2, \ldots, v_L) \ .$$

The tables $T$ and $\vec{T}$ are used to reduce a polynomial modulo the irreducible polynomial $n(x)$. The PRNS-based multiplication algorithm is given below.

THE PRNS-BASED MULTIPLICATION ALGORITHM

| | |
|---|---|
| Input: | $\vec{a}$ and $\vec{b}$ |
| Output: | $\vec{c}$ and $c(x)$ |
| Auxiliary: | $M_1, M_2, \ldots, M_L$, $\vec{I}$, $T$, and $\vec{T}$ |
| Step 1. | $\vec{c} := \vec{a} * \vec{b}$ |
| Step 2. | $\vec{r} := \vec{c} * \vec{I}$ |
| Step 3. | $c(x) := \sum_{i=1}^{L} r_i \cdot M_i \mod M(x)$ |
| Step 4. | for $i = 2s - 1$ downto $s$ |
| Step 5. | $\vec{c} := \vec{c} + \vec{T}[C_i] * x^{(i-s)w}$ |
| Step 6. | $c(x) := c(x) + T[C_i] \cdot x^{(i-s)w}$ |
| Step 7. | return $\vec{c}$ and $c(x)$ |

Assuming the input polynomials $a(x)$ and $b(x)$ are of degree at most $k-1$, we have the product polynomial $c(x)$ in its residue representation at the end of Step 1, however, this polynomial is of degree $2(k-1)$. The representation is still unique, since the degree of $M(x)$ is at least $2k$, and thus, the SRC algorithm will yield a unique result. However, we cannot use the resulting polynomial $c(x)$ and its residue representation $\vec{c}$ as an input to another multiplication. We need to use the generating polynomial $n(x)$ to reduce $\vec{c}$ and $c(x)$ so that the result is again less than $n(x)$. Steps 3–6 accomplish this reduction. First we use the SRC algorithm to compute $c(x)$, and then in Steps 5 and 6, we use the left-to-right reduction algorithm to reduce $c(x)$ so that it is of degree at most $k-1$. We perform the operation in Step 6 by first discarding the $i$th (the most significant) word of

$$c(x) = (C_i C_{i-1} \cdots C_1 C_0) \ ,$$

and then by adding the $s$-word number

$$T[C_i] = (V_{s-1} V_{s-2} \cdots V_1 V_0)$$

to the partial product $c(x)$ by aligning $V_{s-1}$ with $C_{i-1}$ from the left:

$$\begin{array}{cccccccccc} C_i & C_{i-1} & C_{i-2} & \cdots & C_{i-s+1} & C_{i-s} & C_{i-s-1} & \cdots & C_1 & C_0 \\ C_i & V_{s-1} & V_{s-2} & \cdots & V_1 & V_0 & & & & \end{array}$$

The addition of the most significant words $C_i + C_i = 0$ is not performed. Also the terms $C_{i-s-1}$ down to $C_0$ are not involved in the addition either. Only the terms starting from $V_{s-1}$ down to $V_0$ are added to the corresponding terms of $c(x)$ in order to reduce $c(x)$ modulo $n(x)$. Thus, the shift factor $x^{(i-s)w}$ is taken care of by this alignment process.

Furthermore, as we reduce $c(x)$ modulo $n(x)$ in Step 6, we also reduce its residue representation $\vec{c}$ using $\vec{n}$ and $\vec{T}$ in Step 5 by multiplying the residue numbers $\vec{T}[C_i]$ and

$$(x^{(i-s)w}, x^{(i-s)w}, \ldots, x^{(i-s)w}) \ ,$$

and then adding the result to $\vec{c}$.

## 10.4 Example : PRNS-based Multiplication

We will illustrate the PRNS-based multiplication algorithm using the same example as in Section 9.4, i.e. we will compute

$$c(x) = a(x)b(x) \pmod{n(x)} \ ,$$

using the same polynomials $a(x)$, $b(x)$ and $n(x)$ as before:

$$\begin{aligned} n(x) &= x^8 + x^5 + x^3 + x^2 + 1 &&= (1\ 0010\ 1101) \\ a(x) &= x^7 + x^6 + x^4 + x^3 + x + 1 &&= (1101\ 1011) &&= (A_1 A_0) \\ b(x) &= x^7 + x^5 + x^3 + x^2 + x &&= (1010\ 1110) &&= (B_1 B_0) \end{aligned}$$

The lookup table $T$ of multiples of $n(x)$ is exactly the same as $T_1$ in Table 9.1. The sorted form is shown in Table 10.1.

The precomputation of the multiples will be extended to include the PRNS representations. Since $k = 8$ for we select $w = 4$, and thus, $s = 2$. Furthermore, $L = 2s = 4$, which implies that we need 4 pairwise relatively prime polynomials $m_i(x)$, each of which is of degree 4, to construct the PRNS. We select the following polynomials as moduli:

$$m_1(x) = x^4 + x + 1$$

**Table 10.1.** The lookup tables $T$ and $\vec{T}$ for $n(x) = (1\ 0010\ 1101)$.

| index | $T$ | $\vec{T}$ |
|---|---|---|
| (0000) | (0000 0000) | (0000, 0000, 0000, 0000) |
| (0001) | (0010 1101) | (1110, 1000, 1000, 0100) |
| (0010) | (0101 1010) | (1111, 1001, 1101, 1000) |
| (0011) | (0111 0111) | (0001, 0001, 0101, 1100) |
| (0100) | (1011 0100) | (1101, 1011, 0111, 1111) |
| (0101) | (1001 1001) | (0011, 0011, 1111, 1011) |
| (0110) | (1110 1110) | (0010, 0010, 1010, 0111) |
| (0111) | (1100 0011) | (1100, 1010, 0010, 0011) |
| (1000) | (0100 0101) | (0111, 0111, 0110, 0101) |
| (1001) | (0110 1000) | (1001, 1111, 1110, 0001) |
| (1010) | (0001 1111) | (1000, 1110, 1011, 1101) |
| (1011) | (0011 0010) | (0110, 0110, 0011, 1001) |
| (1100) | (1111 0001) | (1010, 1100, 0001, 1010) |
| (1101) | (1101 1100) | (0100, 0100, 1001, 1110) |
| (1110) | (1010 1011) | (0101, 0101, 1100, 0010) |
| (1111) | (1000 0110) | (1011, 1101, 0100, 0110) |

$$
\begin{aligned}
m_2(x) &= x^4 + x^3 + 1 \\
m_3(x) &= x^4 + x^3 + x^2 + 1 \\
m_4(x) &= x^4 + x^3 + x^2 + x + 1
\end{aligned}
$$

This gives us

$$
\begin{aligned}
M(x) &= m_1(x)m_2(x)m_3(x)m_4(x) \\
&= x^{16} + x^{15} + x^{14} + x^{13} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + 1 \ .
\end{aligned}
$$

The following values are also easily computed:

$$
\begin{aligned}
M_1 &= x^{12} + x^{11} + x^{10} + x^7 + x + 1 \\
M_2 &= x^{12} + x^{10} + x^8 + x^3 + x^2 + 1 \\
M_3 &= x^{12} + x^9 + x^6 + x^3 + 1 \ , \\
M_4 &= x^{12} + x^8 + x^5 + x^4 + x^3 + x^2 + x + 1 \\
\vec{n} &= (x^3 + x^2 + x, x^3, x^3, x^2) \\
\vec{I} &= (M_1^{-1}, M_2^{-1}, M_3^{-1}, M_4^{-1}) \\
&= (x + 1, x^3 + x + 1, x^2 + x + 1, x^3 + x^2)
\end{aligned}
$$

We then compute the multiples of $\vec{n}$ and store them in table $\vec{T}$. The lookup table $\vec{T}$, is also added to Table 10.1, with the entries in the same row representing the same multiple.

We now illustrate the steps of the PRNS-based finite field multiplication algorithm that computes the product $c(x)$. The PRNS representations of $a$ and $b$ are found as

$$
\begin{aligned}
\vec{a} &= (1111, 1010, 1001, 0010) \\
\vec{b} &= (0011, 0010, 1000, 1011)
\end{aligned}
$$

The PRNS-based multiplication algorithm starts with $c = 0$ and performs the following steps to find the result

$$c(x) = x^6 + x^5 + x^4 + x^3 + 1 = (0111\ 1001)$$

as follows:

$$
\begin{aligned}
\text{Step 1:}\quad \vec{c} &= \vec{a} * \vec{b} = (1111, 1010, 1001, 0010) * (0011, 0010, 1000, 1011) \\
&= (0010, 1101, 0110, 1001) \\
\text{Step 2:}\quad \vec{r} &= \vec{c} * \vec{I} = (0010, 1101, 0110, 1001) * (0011, 1011, 0111, 1100) \\
&= (0110, 0010, 1111, 1111) \\
\text{Step 3:}\quad c(x) &= \textstyle\sum_{i=1}^{4} r_i \cdot M_i \mod M(x) \\
&= (0110) \cdot (0001\ 1100\ 1000\ 0011) + \\
&\quad (0010) \cdot (0001\ 0101\ 0000\ 1101) + \\
&\quad (1111) \cdot (0001\ 0010\ 0100\ 1001) + \\
&\quad (1111) \cdot (0001\ 0001\ 0011\ 1111) \\
&= (0111\ 0010\ 1110\ 0010) = (C_3 C_2 C_1 C_0) \\
(i = 3)\ \text{Step 5:}\quad \vec{c} &= \vec{c} + \vec{T}[C_3] * x^4 = \vec{c} + \vec{T}[0111] * x^4 \\
&= (0010, 1101, 0110, 1001) + (1100, 1010, 0010, 0011) * x^4 \\
&= (0101, 0001, 0001, 0111) \\
\text{Step 6:}\quad c(x) &= c(x) + T[C_3] \cdot x^4 = (0010\ 1110\ 0010) + T[0111] \cdot x^4 \\
&= (1110\ 1101\ 0010) = (C_2 C_1 C_0) \\
(i = 2)\ \text{Step 5:}\quad \vec{c} &= \vec{c} + \vec{T}[C_2] * x^0 = \vec{c} + \vec{T}[1110] \\
&= (0101, 0001, 0001, 0111) + (0101, 0101, 1100, 0010) \\
&= (0000, 0100, 1101, 0101) \\
\text{Step 6:}\quad c(x) &= c(x) + T[C_2] \cdot x^0 = (1101\ 0010) + T[1110] \\
&= (1101\ 0010) + (1010\ 1011) \ = \ (0111\ 1001) = (C_1 C_0)
\end{aligned}
$$

The last vector $\vec{c}$ in Step 5 or the last polynomial $c(x)$ in Step 6 yields the result. The result $\vec{c}$ in Step 5 is reduced modulo $n(x)$, and thus, it can be used as an input to another multiplication.

## 10.5 Improving the PRNS-based Multiplication Algorithm

In this section, we give an improved PRNS-based multiplication algorithm, which saves computational time and space as compared to the multiplication algorithm given in Section 10.3 and exemplified in Section 10.4. The improved algorithm is based on the following observations:

- We compute the polynomial representation of $c(x)$ along with its residue representation $\vec{c}$ only because we need the words of $c(x)$ to reduce $\vec{c}$ modulo $n(x)$.
- We only use the most significant $s$ words of $c(x)$ starting from $(2s-1)$ down to $s$ to perform this reduction, as seen in Step 4.

Therefore, the SRC algorithm for computing $c(x)$ can be modified so that we only compute the most significant $s$ words of $c(x)$, and thus, save space and time in the PRNS-based multiplication algorithm.

In Step 3 of the PRNS-based multiplication algorithm, we first compute

$$c(x) = \sum_{i=1}^{L} r_i \cdot M_i$$

and then perform a modulo $M(x)$ reduction. Since $L = 2s$, and each one of $M_i$ is of length (at most) $(2s-1)$ words, the above computation can be written as

$$c(x) = \sum_{i=1}^{L} r_i \cdot (M_{i,2s-2} M_{i,2s-3} \cdots M_{i,1} M_{i,0}) \ .$$

Since we are only interested in computing the most significant $s$ words of $c(x)$, the above sum must be divided by $x^{sw}$. To avoid unnecessary computation, we first divide $M_i$ by $x^{(s-1)w}$, then perform the multiplications and the summation, and finally divide the result by $x^w$. This way we first discard the part of $M_i$ which does not contribute to the final result, and then, we perform another division to get only the necessary part of $c(x)$. More explicitly, we truncate $M_i$ by ignoring the words indexed from 0 up to $(s-2)$, and only keep the ones from $(s-1)$ up to $(2s-2)$. This implies that we compute

$$\sum_{i=1}^{L} r_i \cdot (M_{i,2s-2} M_{i,2s-3} \cdots M_{i,s} M_{i,s-1} 00 \cdots 0) \ .$$

The least significant $(s-1)$ words of zeros can be ignored by computing

$$\sum_{i=1}^{L} r_i \cdot (M_{i,2s-2} M_{i,2s-3} \cdots M_{i,s} M_{i,s-1}) \ = \ \sum_{i=1}^{L} r_i \cdot \frac{M_i}{x^{(s-1)w}} \ .$$

The multiplication operation

$$r_i \cdot (M_{i,2s-2} M_{i,2s-3} \cdots M_{i,s} M_{i,s-1})$$

produces an $(s+1)$-word number; the addition of $L$ such numbers is still of length $(s+1)$ words. Since we only need the most significant $s$ words, we ignore the least significant word of the result as

$$c'(x) = \left(\sum_{i=1}^{L} r_i \cdot \frac{M_i}{x^{(s-1)w}}\right) \cdot \frac{1}{x^w} \;=\; (C'_{s-1}C'_{s-2}\cdots C'_1C_0) \;.$$

This result is less than $M(x)$ and it is not reduced modulo $M(x)$. Furthermore, the values

$$M'_i(x) = \frac{M_i(x)}{x^{(s-1)w}}$$

can be precomputed. The most significant $s$ words of $c(x)$ is computed using the modified SRC algorithm as follows:

THE MODIFIED SRC ALGORITHM

| | |
|---|---|
| Input: | $\vec{p} = (p_1, p_2, \ldots, p_L)$ |
| Output: | $p'(x)$: The most significant $s$ words of $p(x)$ |
| Auxiliary: | $M'_1, M'_2, \ldots, M'_L$ and $\vec{I}$ |
| Step 1. | $\vec{r} := \vec{p} * \vec{I}$ |
| Step 2. | $p'(x) := \left(\sum_{i=1}^{L} r_i \cdot M'_i\right) \cdot x^{-w}$ |
| Step 3. | return $p'(x)$ |

The modified PRNS-based multiplication algorithm uses the modified SRC algorithm given above to compute $c'(x)$ in order to reduce $\vec{c}$.

THE MODIFIED PRNS-BASED MULTIPLICATION ALGORITHM

| | |
|---|---|
| Input: | $\vec{a}$ and $\vec{b}$ |
| Output: | $\vec{c}$ |
| Auxiliary: | $M'_1, M'_2, \ldots, M'_L$, $\vec{I}$, $T$, and $\vec{T}$ |
| Step 1. | $\vec{c} := \vec{a} * \vec{b}$ |
| Step 2. | $\vec{r} := \vec{c} * \vec{I}$ |
| Step 3. | $c'(x) := \left(\sum_{i=1}^{L} r_i \cdot M'_i\right) \cdot x^{-w}$ |
| Step 4. | for $i = s-1$ downto 0 |
| Step 5. | $\quad \vec{c} := \vec{c} + \vec{T}[C'_i] * x^{iw}$ |

Step 6. $c'(x) := (\ c'(x) \bmod x^{iw}\ ) + T[C'_i] \cdot x^{-(s-i)w}$

Step 7. return $\vec{c}$

We have only the most significant $s$ words of $c(x)$. The entire $c(x)$ can be written as

$$c(x) \ = \ (C'_{s-1}C'_{s-2} \cdot C'_1C'_0C_{s-1}C_{s-2} \cdot C_1C_0) \ .$$

In order to reduce $c(x)$ in step for $i = s-1$, we need to take the most significant word $C'_{s-1}$ and obtain the table entry

$$T[C'_{s-1}] = (V_{s-1}V_{s-2} \cdots V_1V_0) \ ,$$

and add it to $c(x)$ as

$$\begin{array}{cccccccccc} C'_{s-1} & C'_{s-2} & \cdots & C'_1 & C'_0 & C_{s-1} & C_{s-2} & \cdots & C_1 & C_0 \\ C'_{s-1} & V_{s-1} & \cdots & V_2 & V_1 & V_0 & & & & \end{array}$$

In general, in the $i$th step, we need to take the least significant $i$ words of $c'(x)$ and add $(s-i)$ words right-shifted version of $T[C'_i]$ to $c'(x)$ in Step 6. Similarly, we perform the reduction on $\vec{c}$ in Step 5. The most significant words of $c(x)$ are completely zeroed during the reduction process in Step 6. We do not provide $c(x)'$ as an output, and the modified PRNS-based multiplication algorithm returns $\vec{c}$ only.

## 10.6 Example : Improved PRNS-based Algorithm

We illustrate the steps of the modified PRNS-based multiplication algorithm using the same example as the one in Section 10.4. The precomputation part and the tables remain the same. Additionally, we need to compute $M'_i(x)$ for $i = 1, 2, \ldots, L$, which are obtained using

$$M'_i = \frac{M_i}{x^{(s-1)w} = M_i/x^4}$$

as

$$\begin{aligned} M'_1 &= (0001\ 1100\ 1000\ 0011)/x^4 = (0001\ 1100\ 1000) \ , \\ M'_2 &= (0001\ 0101\ 0000\ 1101)/x^4 = (0001\ 0101\ 0000) \ , \end{aligned}$$

$$M_3' = (0001\ 0010\ 0100\ 1001)/x^4 = (0001\ 0010\ 0100)\ ,$$
$$M_4' = (0001\ 0001\ 0011\ 1111)/x^4 = (0001\ 0001\ 0011)\ .$$

The algorithm takes the same $\vec{a}$ and $\vec{b}$ as input operands, and performs the following steps to find the result

$$\vec{c} = (0000, 0100, 1101, 0101)\ .$$

$$\begin{aligned}
\text{Step 1:}\quad \vec{c} &= \vec{a} * \vec{b} = (1111, 1010, 1001, 0010) * (0011, 0010, 1000, 1011) \\
&= (0010, 1101, 0110, 1001) \\
\text{Step 2:}\quad \vec{r} &= \vec{c} * \vec{I} = (0010, 1101, 0110, 1001) * (0011, 1011, 0111, 1100) \\
&= (0110, 0010, 1111, 1111) \\
\text{Step 3:}\quad c'(x) &= (\textstyle\sum_{i=1}^{4} r_i \cdot M_i') \cdot x^{-4} \\
&= [(0110) \cdot (0001\ 1100\ 1000) + (0010) \cdot (0001\ 0101\ 0000) + \\
&\quad (1111) \cdot (0001\ 0010\ 0100) + \\
&\quad (1111) \cdot (0001\ 0001\ 0011)] \cdot x^{-4} \\
&= (0111\ 0010) \\
(i=1)\ \text{Step 5:}\quad \vec{c} &= \vec{c} + \vec{T}[C_1'] * x^4 = \vec{c} + \vec{T}[0111] * x^4 \\
&= (0010, 1101, 0110, 1001) + (1100, 1010, 0010, 0011) * x^4 \\
&= (0101, 0001, 0001, 0111) \\
\text{Step 6:}\quad c'(x) &= (\ c'(x) \bmod x^4\ ) + T[C_1'] \cdot x^{-4} = (0010) + T[0111] \cdot x^{-4} \\
&= (0010) + (1100) = (1110) \\
(i=0)\ \text{Step 5:}\quad \vec{c} &= \vec{c} + \vec{T}[C_0] * x^0 = \vec{c} + \vec{T}[1110] \\
&= (0101, 0001, 0001, 0111) + (0101, 0101, 1100, 0010) \\
&= (0000, 0100, 1101, 0101) \\
\text{Step 6:}\quad c(x)' &= (\ c'(x) \bmod x^0\ ) + T[C_0'] \cdot x^0 \\
&= (1110) + T[1110] = (0000)
\end{aligned}$$

Since $c'(x)$ after Step 6 for $i = 0$ is not needed, this computation may be skipped.

## 10.7 Analysis of the PRNS-based Multiplication Algorithm

In this section, we analyze the PRNS-based multiplication algorithm by counting the total number of word-level $GF(2)$ addition and multiplication operations. The word-

**Table 10.2.** Operation counts for the PRNS-based multiplication algorithm.

| Steps | MULGF2 | XOR |
|---|---|---|
| Step 1 | 1 | |
| Step 2 | 1 | |
| Step 3 | $s$ | $s-1+s\log_2(2s)$ |
| Step 5 ($s$ times) | 1 | 1 |
| Step 6 ($s$ times) | | 1 |
| Total | $2s+2$ | $3s-1+s\log_2(2s)$ |

level multiplication operation MULGF2 is performed using the table lookup method, and the word-level addition operation is performed using the bit-wise XOR which is available on most processors. The implementation details of these operations were studied in Section 8.3 before.

Following the assumption made in Section 10.2, we select $\deg(m_i) = w$, and perform modulo $m_i(x)$ multiplications using the table lookup method.

Since there are $L$ different moduli, we assume that we have $L = 2s$ processors each of which performs its arithmetic (addition and multiplication) operations with respect to its selected modulus. We also use a processor which we call the 'server' to perform a few table lookup operations. The server can be one of the processors. We ignore the server operations and the communication overhead in our analysis.

The analysis of the improved PRNS-based multiplication algorithm is given below. The results are summarized in Table 10.2.

**Step 1:** This step requires a single MULGF2 operation by each of $2s$ processors.

**Step 2:** This step requires a single MULGF2 operation by each of $2s$ processors.

**Step 3:** Let

$$M_i' = (M_{i,s-1}' M_{i,s-2}' \cdots M_{i,0}') \ .$$

This value is already precomputed and saved. During Step 3, the $i$th processor multiplies $r_i$ by $M'_i$ using MULGF2 operation and obtains the $(s+1)$-word result. Each word multiplication for $j = 0, 1, \ldots, (s-1)$ produces a 2-word result

$$r_j \cdot M'_{i,j} = (H_j L_j) \ .$$

These parts must then be added to obtain the final result $r_i \cdot M'_i$ as follows:

| | $M'_{i,s-1}$ | $M'_{i,s-2}$ | $\cdots$ | $M'_{i,2}$ | $M'_{i,1}$ | $M'_{i,0}$ |
|---|---|---|---|---|---|---|
| | | | | | | $r_i$ |
| | $L_{s-1}$ | $L_{s-2}$ | | $L_2$ | $L_1$ | $L_0$ |
| $H_{s-1}$ | $H_{s-2}$ | $H_{s-3}$ | | $H_1$ | $H_0$ | |
| $S_s$ | $S_{s-1}$ | $S_{s-2}$ | $\cdots$ | $S_2$ | $S_1$ | $S_0$ |

The above operation is performed using $s$ MULGF2 operations and $(s-1)$ XOR operations by the $i$th processor for all processors $i = 1, 2, \ldots, 2s$.

Then the 0th word is discarded and the remaining $s$-word polynomials are summed by all $2s$ processors using the binary tree algorithm. This operation takes $\log_2(2s)$ steps, where at each step two $s$-word polynomials are added. Therefore, $s \log_2(2s)$ XOR operations are required.

The resulting $s$-word polynomial $c'(x)$ is communicated to the server and to the first $s$ processors among all $2s$ processors. This polynomial is needed in Step 6.

**Step 5:** To speed up this step, the values

$$x^{iw} \pmod{m_j}$$

are precomputed and stored for all $i = 1, 2, \ldots, (s-1)$ and $j = 1, 2, \ldots, 2s$. The server performs a lookup operation for $x^{iw} \pmod{m_j}$ and also for $\vec{T}[C'_i]$, and sends them to the $j$th processor for all $j = 1, 2, \ldots, 2s$. Each processor then performs a single MULGF2 operation and a single XOR operation to obtain

$$\vec{c} := \vec{c} + \vec{T}[C'_i]$$

for a single vector entry.

**Step 6:** In this step, we add the $i$-word shifted polynomial $T[C_i']$ to $c'(x)$. Since $c'(x)$ is available in the first $s$ processors, each of one of these processors performs a single `XOR` operation. The updated polynomial $c'(x)$ is then communicated to the server.

## 10.8 Applications of the PRNS-based Multiplication

The improved algorithm takes two polynomials in their residue representation $\vec{a}$ and $\vec{b}$ such that the degree of each of $a(x)$ and $b(x)$ are less than $k$, and produces the product polynomial in its residue representation $\vec{c}$. The squaring algorithm can be given using the similar construction method, however, there may be certain optimizations.

The PRNS-based multiplication algorithm for the field $GF(2^k)$ finds its applications in cryptography where the range of the operands are large, usually

$$160 \leq k \leq 1024 \ ,$$

therefore, it is justifiable to use parallel polynomial arithmetic. We give an exponentiation algorithm for computing $g^e$ where $g \in GF(2^k)$ and $e$ is an $r$-bit integer

$$e = (e_{r-1}e_{r-2}\cdots e_1e_0)$$

below.

THE PRNS-BASED EXPONENTIATION ALGORITHM

| | |
|---|---|
| Input: | $g(x)$, $e$, and $n(x)$ |
| Output: | $c(x) = g^e$ |
| Auxiliary: | $M_1', M_2', \ldots, M_L'$, $\vec{I}$, $T$, and $\vec{T}$ |
| Step 1. | Compute $\vec{g}$ and $\vec{c} := (1, 1, \ldots, 1)$ |
| Step 2. | for $i = r - 1$ downto 0 |
| Step 3. | $\quad \vec{c} := \text{Multiply}(\vec{c}, \vec{c})$ |
| Step 4. | $\quad$ if $e_i = 1$ then $\vec{c} := \text{Multiply}(\vec{g}, \vec{c})$ |
| Step 5. | $c(x) := \text{SRC}(\vec{c})$ |
| Step 6. | return $c(x)$ |

Here the multiplication algorithm is the modified PRNS-based multiplication method given in Section 10.5, which uses the modified SRC algorithm within. Since we need the entire $c(x)$ as the output of the exponentiation operation, the original SRC algorithm is used in Step 6.

# Chapter 11
# A GENERAL MASTROVITO MULTIPLIER

In this chapter, we propose a new formulation of the multiplication matrix and an architecture for the multiplication operation in finite fields of characteristic 2. The proposed architecture generalizes the Mastrovito multiplication. Since $2^m$ is generally used to represent the finite field associated with the Mastrovito multiplier, different than the rest of the thesis, we will use the variable $m$ instead of $k$, to denote the dimension of the finite field in this chapter. The proposed method is particularly efficient when applied to a specific class of polynomials that is known in advance, as it uses all possible optimizations. We have also studied all known special cases in detail, and obtained space and time complexities, and furthermore, provided actual design examples.

## 11.1 Introduction

The efficiency of the architecture is measured by the number of 2-input gates (`XOR` and `AND`) and by the total gate delay of the circuit. The representation of the field elements have crucial role in the efficiency of the architecture. For example, the well-known Massey-Omura [53] algorithm uses the normal basis representation, where the squaring of a field element is equivalent to a cyclic shift in its binary representation (*see* Chapter 8). Efficient bit-parallel algorithms for the multiplication operation in the canonical basis representation, which have much less space and time complexity than the Massey-Omura multiplier, have also been proposed.

The standard (polynomial) basis multiplication requires a polynomial modular multiplication followed by a modular reduction. In practice, these two steps can be combined. A novel method of multiplication is proposed by Mastrovito in [39, 40], in

which a matrix product representation of the multiplication operation is used. The Mastrovito multiplier using the special generating trinomial

$$x^m + x + 1$$

is shown to require $(m^2 - 1)$ XOR gates and $m^2$ AND gates [39, 40, 54, 55]. It has been conjectured that the space complexity of the Mastrovito multiplier would also be the same for all trinomials of the form

$$x^m + x^n + 1$$

for $n = 1, 2, \ldots, m-1$. This conjecture was shown to be not true for the case of $m = 2n$ in [31]. The architecture proposed by Koç and Sunar in [31] requires $(m^2-1)$ XOR gates and $m^2$ AND gates, when $m \neq 2n$. However, the required number of XOR gates is reduced to $(m^2 - \frac{m}{2})$ for the trinomial

$$x^m + x^{\frac{m}{2}} + 1 \ ,$$

for an even $m$.

In this chapter, we generalize the approach of [31] in several ways. We describe a method of construction for the Mastrovito multiplier for a general irreducible polynomial. We give detailed space and time analysis of the proposed method for several different types of irreducible polynomials. It turns out that our method also requires $(m^2 - \frac{m}{2})$ XOR and $m^2$ AND gates for the trinomial of the form

$$x^m + x^{\frac{m}{2}} + 1 \ ,$$

for an even $m$.

The best special case of the proposed method is in the case of equally-spaced-polynomial (ESP), i.e., a polynomial of the form

$$p(x) = x^{k\Delta} + x^{(k-1)\Delta} + \cdots + x^{\Delta} + 1 \ .$$

The best-known architecture for this case requires $(m^2 - 1)$ XOR gates [21], whereas our method uses only $(m^2 - \Delta)$ XOR gates. The ESPs reduce to the special trinomials

$$x^m + x^{\frac{m}{2}} + 1$$

for $k = 2$ and reduce to the all-one-polynomials (AOPs)

$$x^m + x^{m-1} + \cdots + x^2 + x + 1 \ ,$$

for $\Delta = 1$. Thus, our proposed general architecture requires $(m^2 - 1)$ XOR gates and $m^2$ AND gates when the irreducible polynomial is an AOP. An architecture for the canonical basis multiplication operation in $GF(2^m)$ was proposed by Koç and Sunar in [32] for an irreducible AOP, which uses $(m^2 - 1)$ XOR gates and $m^2$ AND gates, while requiring significantly less space and time complexity than similar bit-parallel finite field multiplication architectures [39, 69, 21]. Therefore, our proposed architecture also captures this established lower bound on the number of XOR gates for multipliers based on irreducible AOPs.

## 11.2 Notation & Preliminaries

Let $p(x)$ be the irreducible polynomial generating the Galois field $GF(2^m)$. In order to compute the multiplication

$$c(x) = a(x)b(x) \pmod{p}(x)$$

in $GF(2^m)$, where $a(x)$, $b(x)$, $c(x) \in GF(2^m)$, we need to first compute the product polynomial

$$d(x) = a(x)b(x) = \left(\sum_{i=0}^{m-1} a_i x^i\right)\left(\sum_{i=0}^{m-1} b_i x^i\right)$$

and then reduce $d(x)$ using $p(x)$ to find the result $c(x) \in GF(2^m)$. We can compute the coefficients of $d(x)$ using following matrix-vector product:

$$\begin{bmatrix} d_0 \\ d_1 \\ d_2 \\ \vdots \\ d_{m-2} \\ d_{m-1} \\ d_m \\ d_{m+1} \\ \vdots \\ d_{2m-3} \\ d_{2m-2} \end{bmatrix} = \begin{bmatrix} a_0 & 0 & 0 & \cdots & 0 & 0 \\ a_1 & a_0 & 0 & \cdots & 0 & 0 \\ a_2 & a_1 & a_0 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ a_{m-2} & a_{m-3} & a_{m-4} & \cdots & a_0 & 0 \\ a_{m-1} & a_{m-2} & a_{m-3} & \cdots & a_1 & a_0 \\ 0 & a_{m-1} & a_{m-2} & \cdots & a_2 & a_1 \\ 0 & 0 & a_{m-1} & \cdots & a_3 & a_2 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & a_{m-1} & a_{m-2} \\ 0 & 0 & 0 & \cdots & 0 & a_{m-1} \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ \vdots \\ b_{m-2} \\ b_{m-1} \end{bmatrix}$$

The above multiplication matrix will be called $\mathbf{M}$, and its rows will be denoted by $\mathbf{M}_i$ where $i = 0, 1, \ldots, 2m-2$. Note that the entries of $\mathbf{M}$ solely consist of coefficients of $a(x)$. The $m \times m$ submatrix $\mathbf{U}^{(0)}$ of $\mathbf{M}$ will be defined as the first $m$ rows of $\mathbf{M}$, and the $(m-1) \times m$ submatrix $\mathbf{L}^{(0)}$ of $\mathbf{M}$ will be defined as the last $(m-1)$ rows of $\mathbf{M}$, i.e.,

$$\mathbf{U}^{(0)} = \begin{bmatrix} a_0 & 0 & 0 & \cdots & 0 & 0 \\ a_1 & a_0 & 0 & \cdots & 0 & 0 \\ a_2 & a_1 & a_0 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ a_{m-2} & a_{m-3} & a_{m-4} & \cdots & a_0 & 0 \\ a_{m-1} & a_{m-2} & a_{m-3} & \cdots & a_1 & a_0 \end{bmatrix}$$

$$\mathbf{L}^{(0)} = \begin{bmatrix} 0 & a_{m-1} & a_{m-2} & \cdots & a_2 & a_1 \\ 0 & 0 & a_{m-1} & \cdots & a_3 & a_2 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & a_{m-1} & a_{m-2} \\ 0 & 0 & 0 & \cdots & 0 & a_{m-1} \end{bmatrix}$$

Superscripts will be used to denote the step numbers during the reduction process and the superscript $(f)$ will be used to indicate the final form of the corresponding matrix. When referring to a matrix in general, no superscript will be used.

All the rows in the submatrix $\mathbf{L}^{(0)}$ of matrix $\mathbf{M}$ will be reduced, using the irreducible polynomial $p(x)$, so that, at the end of reduction, $\mathbf{L}^{(f)}$ will become the zero matrix. During that process, the rows of $\mathbf{L}$ will be added to the rows with lower indices according to the irreducible polynomial. All the row vectors, that are added to the others due to the reduction of a single row, will be called its *children*. The final submatrix $\mathbf{U}^{(f)}$ will be equal to the so called Mastrovito matrix $\mathbf{Z}$, which is multiplied by the column vector $\mathbf{b}$ to produce the result $\mathbf{c}$.

Adding two vectors will be done by adding the corresponding entries. For example adding

$$\mathbf{V} = [\ v_n \quad v_{n-1} \quad \cdots \quad v_1 \quad v_0\ ]$$

and

$$\mathbf{W} = [\ w_n \quad w_{n-1} \quad \cdots \quad w_1 \quad w_0\ ]$$

we get

$$\mathbf{V}+\mathbf{W} = [\ (v_n+w_n) \quad (v_{n-1}+w_{n-1}) \quad \cdots \quad (v_1+w_1) \quad (v_0+w_0)\ ]$$

Also *concatenation* of vectors will be needed during the reduction algorithm. This will be represented using the operator $\|$. For example, the above vectors $\mathbf{V}$ and $\mathbf{W}$, which are both of length $(n+1)$, can be concatenated to form a new vector of length $2(n+1)$, as follows

$$\mathbf{V} \,\|\, \mathbf{W} = [\ v_n \quad v_{n-1} \quad \cdots \quad v_1 \quad v_0 \quad w_n \quad w_{n-1} \quad \cdots \quad w_1 \quad w_0\ ] \ .$$

Note that, in general, the equality of the lengths is not a requirement for the vectors to be concatenated.

During the reduction, the vectors will be shifted to the right and left and the new entries will be filled with zeros. For example $(\mathbf{V} \rightarrow \mathbf{3})$ and $(\mathbf{V} \leftarrow \mathbf{2})$ represent *right and left shifts* of the vector $\mathbf{V}$, by 3 and 2 positions, respectively. For the vector

$$\mathbf{V} = [\ v_n \quad v_{n-1} \quad v_{n-2} \quad \cdots \quad v_2 \quad v_1 \quad v_0\ ] \ ,$$

the results will be

$$(\mathbf{V} \rightarrow \mathbf{3}) = [\ 0 \quad 0 \quad 0 \quad v_n \quad v_{n-1} \quad \cdots \quad v_6 \quad v_5 \quad v_4 \quad v_3\ ],$$
$$(\mathbf{V} \leftarrow \mathbf{2}) = [\ v_{n-2} \quad v_{n-3} \quad v_{n-4} \quad v_{n-5} \quad v_{n-6} \quad \cdots \quad v_1 \quad v_0 \quad 0 \quad 0\ ].$$

Also, at some steps of the reduction, vectors will be used as rows to form matrices. For example, to form a matrix using the last $(n-1)$ entries of the above vectors, the following notation will be adopted

$$\begin{bmatrix} \mathbf{V} \\ (\mathbf{V} \rightarrow 3) \\ (\mathbf{V} \leftarrow 2) \end{bmatrix}_{3\times(n-1)} = \begin{bmatrix} v_{n-2} & v_{n-3} & v_{n-4} & v_{n-5} & \cdots & v_3 & v_2 & v_1 & v_0 \\ 0 & v_n & v_{n-1} & v_{n-2} & \cdots & v_6 & v_5 & v_4 & v_3 \\ v_{n-4} & v_{n-5} & v_{n-6} & v_{n-7} & \cdots & v_2 & v_1 & 0 & 0 \end{bmatrix}.$$

As seen above, although the original vectors are longer, only the last $(n-1)$ entries of each are used, and the rest is discarded.

A final note will be on special matrices that will be encountered frequently during the reduction process. In mathematics literature, a matrix whose entries are constant along each diagonal is called a *Toeplitz matrix*. Sum of two Toeplitz matrices is also a Toeplitz matrix [16]. This property will be used to establish a recursion in the proposed method, as will be seen later.

## 11.3 General Case

Let

$$\begin{aligned} p(x) &= x^{n_k} + x^{n_{(k-1)}} + \cdots + x^{n_1} + x^{n_0} \\ &= x^m + x^{n_{(k-1)}} + \cdots + x^{n_1} + 1 \end{aligned}$$

be a general irreducible polynomial where $n_0$, $n_1$, ..., $n_{(k-1)}$, $n_k$ are integers and

$$0 = n_0 < n_1 < \cdots < n_{(k-1)} < n_k = m\ .$$

The difference between highest two orders will be denoted by $\Delta$, i.e.,

$$\Delta = n_k - n_{(k-1)} = m - n_{(k-1)}$$

First the overall processes will be summarized and then a method will be proposed to find the same result efficiently.

When the irreducible polynomial is used to reduce the rows of $\mathbf{L}^{(0)}$, each row will have $k$ children. The one corresponding to the constant term $x^{n_0} = 1$, is guaranteed to be added to a row in $\mathbf{U}$, but the others might be added to the rows of $\mathbf{L}$ and will need to be further reduced later. To simplify the notation and observe the regularity, $k$ additional matrices will be used. The children produced due to the reductions corresponding to the $x^{n_i}$ term will be added to the $m \times m$ matrix $\mathbf{Xi}^{(1)}$, for $i = 0, 1, \ldots, (k-1)$. The children that fall back into the submatrix $\mathbf{L}$ will be stored in $\mathbf{L}^{(1)}$ to be reduced later.

By the introduction of $\mathbf{Xi}$ matrices, the matrix $\mathbf{U}^{(0)}$ is preserved during the reduction. At the end of the first step, i.e., when every row of matrix $\mathbf{L}^{(0)}$ is reduced exactly once, the following matrices will be produced

$$\mathbf{U}^{(1)} = \mathbf{U}^{(0)} + \mathbf{X0}^{(1)} + \mathbf{X1}^{(1)} + \cdots + \mathbf{X(k-1)}^{(1)} \ ,$$

where

$$\mathbf{Xi}^{(1)} = \left[\begin{array}{ccccccccc} 0 & 0 & 0 & \cdots & 0 & \cdots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & 0 & \cdots & 0 & 0 & (n_i - 1) \\ 0 & a_{m-1} & a_{m-2} & \cdots & a_{n_i} & \cdots & a_2 & a_1 & n_i \\ 0 & 0 & a_{m-1} & \cdots & a_{n_i+1} & \cdots & a_3 & a_2 & (n_i + 1) \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & a_{m-1} & \cdots & a_{m-n_i+1} & a_{m-n_i} & (m-1) \end{array}\right]$$

for $i = 0, 1, \ldots, (k-1)$. The part of matrix $\mathbf{M}$ to be further reduced after the first step, will be

$$\mathbf{L}^{(1)} = \left[\begin{array}{cccccccc|c} 0 & \cdots & 0 & l^{(1)}_{m-1} & l^{(1)}_{m-2} & \cdots & l^{(1)}_{\Delta+2} & l^{(1)}_{\Delta+1} & 0 \\ 0 & \cdots & 0 & 0 & l^{(1)}_{m-1} & \cdots & l^{(1)}_{\Delta+3} & l^{(1)}_{\Delta+2} & 1 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & \cdots & 0 & 0 & 0 & \cdots & l^{(1)}_{m-1} & l^{(1)}_{m-2} & (n_{(k-1)}-3) \\ 0 & \cdots & 0 & 0 & 0 & \cdots & 0 & l^{(1)}_{m-1} & (n_{(k-1)}-2) \\ 0 & \cdots & 0 & 0 & 0 & \cdots & 0 & 0 & n_{(k-1)-1} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & \cdots & 0 & 0 & 0 & \cdots & 0 & 0 & (m-2) \end{array}\right]$$

As can be seen above, the new matrix $\mathbf{L}^{(1)}$, that will be reduced in the next step, is again triangular. That means the new children will again be in the same form, except that they will contain more zero terms at the beginning.

Thus it is clear that if the same procedure is recursively applied, the submatrix $\mathbf{U}^{(0)}$ will never change and the forms of the matrices **Xi** and **L** will remain the same after every step, i.e., **Xi** will all be trapezoidal and **L** will be triangular. The entries that are zero and outside the indicated geometric regions after the first iteration, will always remain zero. Only the values inside these regions will be changed during the rest of the reduction. The number of nonzero rows of **L** after step $j$, i.e., number of nonzero rows in $\mathbf{L}^{(j)}$ can be denoted by

$$r_j = (m-1) - j(m - n_{(k-1)}) = (m-1) - j\Delta \ ,$$

as there are $(m-1)$ nonzero rows initially, i.e., $r_0 = (m-1)$, and the number is reduced by $\Delta = (m - n_{(k-1)})$ after each step. Thus it will take

$$N[m, \Delta] = \left\lceil \frac{m-1}{\Delta} \right\rceil$$

steps to reduce the whole matrix $\mathbf{L}^{(0)}$. This number is also equal to the number of nonzero terms in the row $\mathbf{L}_0$ at the end of step $j$, i.e., number of nonzero terms in the row $\mathbf{L}_0^{(j)}$ for $j = 1, 2, \ldots, N[m, \Delta] - 1$. Note that the range of $j$ does not include $j = N[m, \Delta]$, as the number of nonzero rows becomes zero after step $N[m, \Delta]$, but the number $r_j$ will be negative for $j = N[m, \Delta]$.

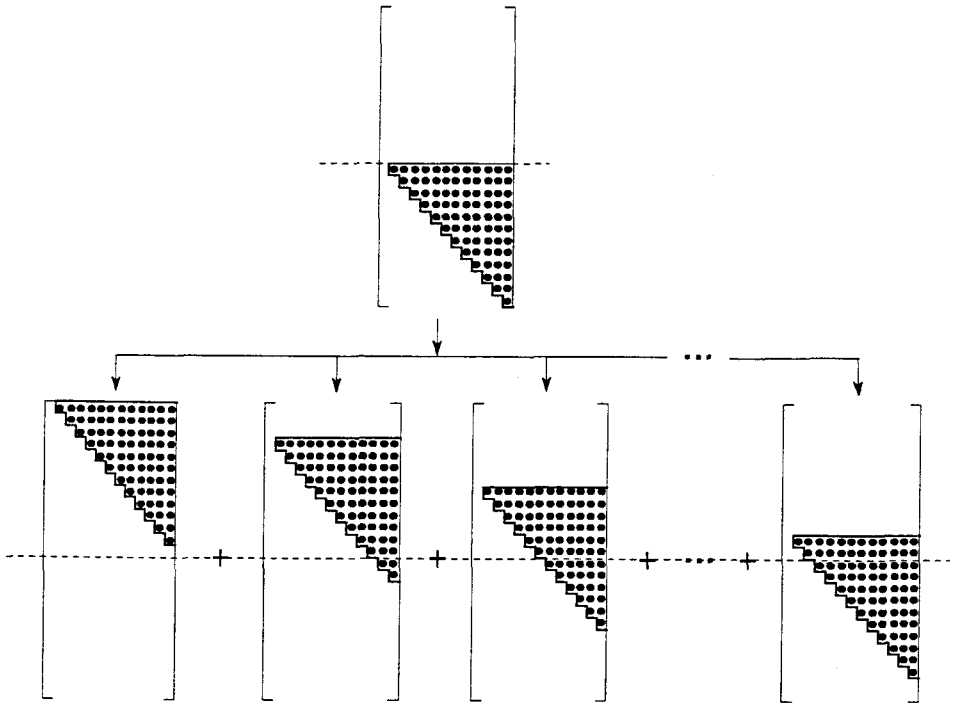

**Figure 11.1.** Reduction of the lower submatrix

First the matrix $\mathbf{M}$ is divided into the upper and lower submatrices $\mathbf{U}^{(0)}$ and $\mathbf{L}^{(0)}$. Then, as shown in Figure 11.1, $\mathbf{L}^{(0)}$ is reduced into $k$ matrices using the irreducible polynomial, while $\mathbf{U}^{(0)}$ is kept unchanged. Upper and lower parts of the new matrices are separated next (*see* Figure 11.2). While upper parts form the matrices $\mathbf{Xi}^{(1)}$, lower parts are accumulated into the matrix $\mathbf{L}^{(1)}$ as shown in Figure 11.3, to be further reduced in the next step. This procedure is repeated until the matrix $\mathbf{L}$ becomes the zero matrix, i.e., all rows are reduced. The sum of the matrices in the last row, i.e., $\mathbf{U}^{(0)}$ and $\mathbf{Xi}^{(f)}$ for $i = 0, 1, \ldots, (k-1)$, yields the Mastrovito matrix $\mathbf{Z}$.

The summary of the reduction will be as shown in Figure 11.4.

When a more detailed analysis is performed, one will see that all the submatrices formed by the nonzero rows of the matrices produced after the first iteration are Toeplitz matrices. When the matrix $\mathbf{L}^{(1)}$ is reduced further, the children will be added to the nonzero rows of the matrices $\mathbf{Xi}^{(1)}$ and $\mathbf{L}^{(2)}$ which are Toeplitz submatrices. As the sum of Toeplitz matrices is again a Toeplitz matrix [16], the submatrices formed by the nonzero rows will all be Toeplitz submatrices again. And because these are

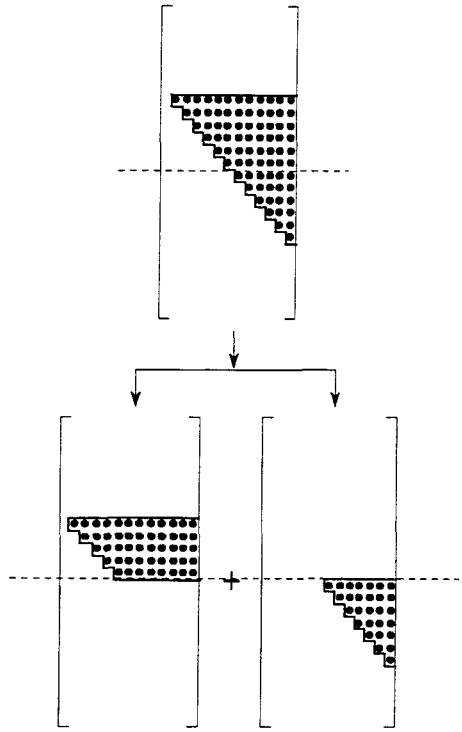

**Figure 11.2.** Separating the upper and lower parts

special Toeplitz matrices, computing only the first nonzero rows of $\mathbf{Xi}^{(f)}$ will be enough to reconstruct them. Similarly the matrix $\mathbf{M}$, and hence the submatrices, $\mathbf{U}^{(0)}$ and $\mathbf{L}^{(0)}$, can be constructed using only the row $\mathbf{M}_{m-1}$. Furthermore, since all the first nonzero rows of the matrices $\mathbf{Xi}^{(f)}$ are identical, it is enough to compute only one of them. Thus it will suffice to work on only $\mathbf{X0}_0^{(f)}$.

Here is how the final $\mathbf{X0}_0^{(f)}$ is computed:

$$\sum_{j=0}^{N[m,\Delta]-1} \mathbf{L}_0^{(j)} \;=\; \mathbf{L}_0^{(0)} + \mathbf{L}_0^{(1)} + \cdots + \mathbf{L}_0^{(N[m,\Delta]-1)} \;.$$

This will be used with $\mathbf{U}^{(0)}$ to construct the final matrix $\mathbf{Z}$. First, the rows $\mathbf{Z}_{n_i}$ are formed by adding the corresponding rows of the matrix $\mathbf{U}^{(0)}$ to $\mathbf{Xi}_{n_i}^{(f)}$, for $i = 0, 1, \ldots, (k-1)$. Then they are extended to larger vectors $\mathbf{Yi}$ by concatenating the necessary parts of $\mathbf{U}^{(0)}$, to their beginning, so that the shifts of $\mathbf{Yi}$ produce the rows below them up to the row $n_{i+1}$, or up to the row $(m-1)$ if $i = (k-1)$. This will simplify the construction of the matrix $\mathbf{Z}$. To further simplify the representations,

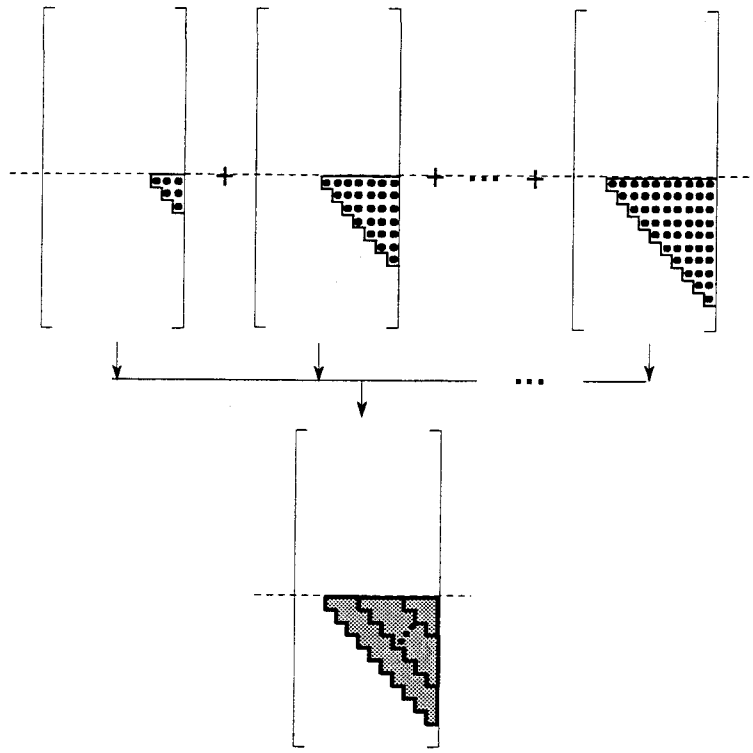

**Figure 11.3.** Accumulation of the lower parts

the first nonzero rows of **Xi**, which are all identical, will be represented by the vector **V** will be used. And instead of referring to $\mathbf{U}^{(0)}$ and $\mathbf{L}^{(0)}$, the original multiplication matrix **M**, or its entries $a_i$ will be used.

So here is the summary of the proposed method:

First compute **V**

$$\mathbf{V} = \sum_{j=0}^{N[m,\Delta]-1} \mathbf{L}_0^{(j)} = [\ 0 \quad v_{m-1} \quad v_{m-2} \quad \cdots \quad v_3 \quad v_2 \quad v_1\ ] \tag{11.5}$$

using the recursive definition of

$$\mathbf{L}_0^{(j)} = \sum_{\substack{r_{(j-1)} > (m-n_i) \\ (k-1) \geq i \geq 0}} (\mathbf{L}_0^{(j-1)} \rightarrow (m-n_i)) \tag{11.6}$$

for $1 \leq j \leq N[m,\Delta]-1$ to reduce everything to the sum of shifts of the row $\mathbf{L}_0^{(0)}$, or equivalently to the sum of rows in **M**. The above summation means that the row

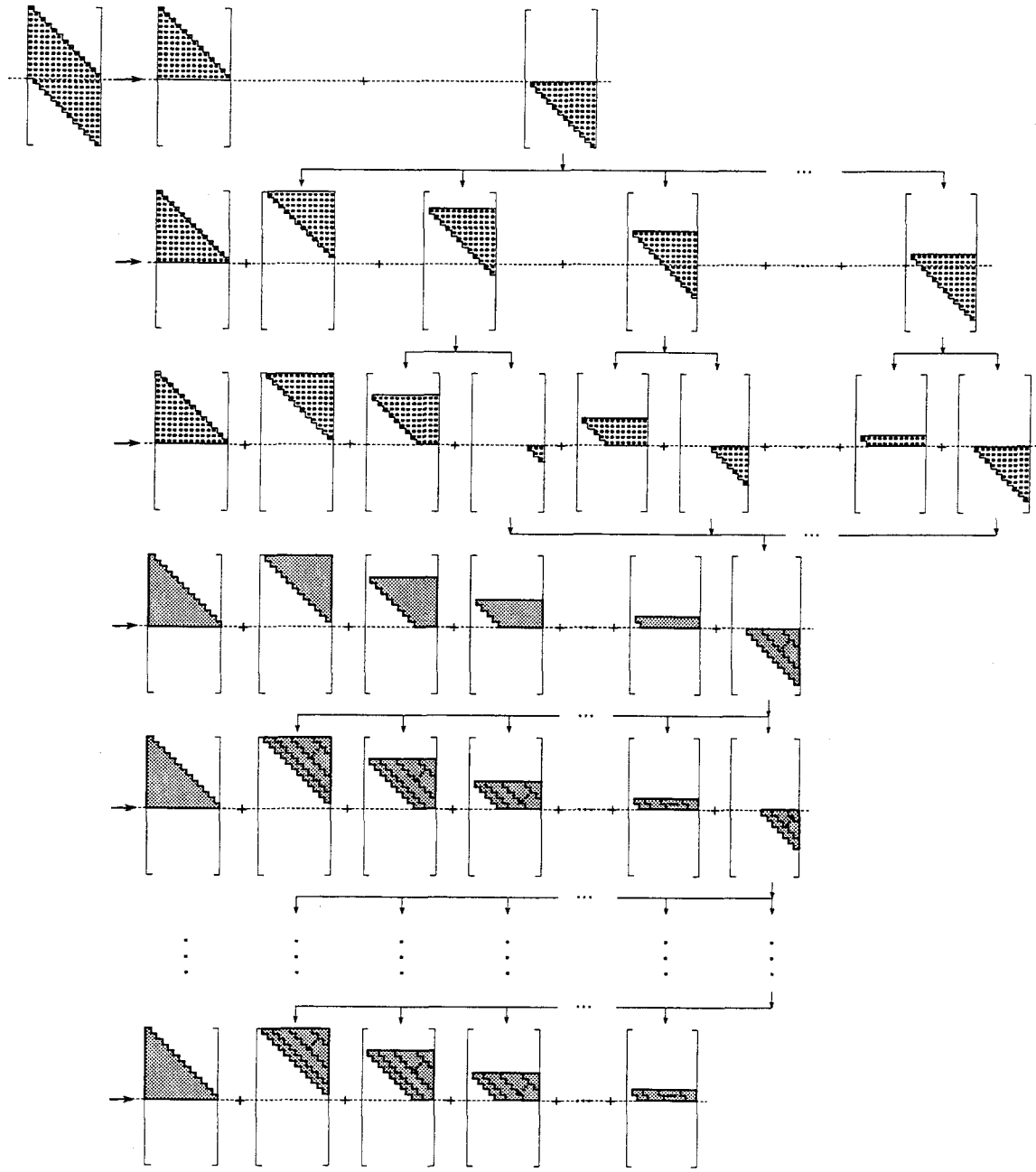

**Figure 11.4.** Reduction of the multiplication matrix for the general case

$\mathbf{L}_0^{(j-1)}$ is shifted by $(m - n_{(k-1)})$,$(m - n_{k-2})$, etc. until all the entries become zero. Then all are added to form $\mathbf{L}_0^{(j)}$. One performance note here: $\mathbf{V}$ is not computed until it is totally reduced to the sum of rows of $\mathbf{M}$, as there might be cancellations. Then, compute $\mathbf{Z}_{n_i}$ for $i = 0, 1, \ldots, (k-1)$ using the following recursive relations:

$$\begin{aligned} \mathbf{Z}_0 &= [a_0] \parallel [\mathbf{V}]_{1\times(m-1)} = [\ a_0\ \ v_{m-1}\ \ v_{m-2}\ \ \cdots\ \ v_3\ \ v_2\ \ v_1\ ] \\ \mathbf{Z}_{n_i} &= \left([\mathbf{M}_{m+n_{(i-1)}}]_{1\times\Delta_i} \parallel [\mathbf{Z}_{n_{(i-1)}} \rightarrow \Delta_i]_{1\times(m-\Delta_i)}\right) + \mathbf{V}\ , \end{aligned} \tag{11.7}$$

where $\Delta_i = (n_i - n_{(i-1)})$ for $i = 1, 2, \ldots, (k-1)$. Thus if

$$\begin{aligned} \mathbf{V} &= [\ \ 0\ \ v_{m-1}\ \ v_{m-2}\ \ \cdots\ \ v_3\ \ v_2\ \ v_1\ ] \text{ and} \\ \mathbf{Z}_{n_{(i-1)}} &= [\ a_{n_{(i-1)}}\ \ w_{m-1}\ \ w_{m-2}\ \ \cdots\ \ w_3\ \ w_2\ \ w_1\ ]\ , \end{aligned}$$

then

$$\begin{aligned} \mathbf{Z}_{n_i} = [\ & a_{n_i}\ \ (a_{n_i-1} + v_{m-1})\ \cdots \\ & \cdots\ (a_{n_{(i-1)}} + v_{m-\Delta_i})\ \ (w_{m-1} + v_{m-1-\Delta_i})\ \cdots\ (w_{\Delta_i+1} + v_1)\ ]\ . \end{aligned}$$

Next find $\mathbf{Yi}$ for $i = 0, 1, \ldots, (k-1)$ by extending $\mathbf{Z}_{n_i}$

$$\begin{aligned} \mathbf{Yi} &= [\mathbf{M}_{m+n_i}]_{1\times(\Delta_{(i+1)}-1)} \parallel \mathbf{Z}_{n_i} \\ &= [\ a_{n_{(i+1)}-1}\ \cdots\ a_{n_i+1}\ \ a_{n_i}\ \ (a_{n_i-1} + v_{m-1})\ \cdots \\ &\qquad \cdots\ (a_0 + v_{m-n_i})\ \ (w_{m-1} + v_{m-n_i-1})\ \cdots\ (w_{n_i+1} + v_1)\ ]\ . \end{aligned} \tag{11.8}$$

Finally, the whole $\mathbf{Z}$ matrix can be constructed as follows:

$$
\mathbf{Z} = \left[\begin{array}{c|c}
\mathbf{Y0} & 0 \\
\mathbf{Y0} \rightarrow 1 & 1 \\
\vdots & \vdots \\
\mathbf{Y0} \rightarrow (\Delta_1 - 1) & (n_1 - 1) \\
\mathbf{Y1} & n_1 \\
\mathbf{Y1} \rightarrow 1 & (n_1 + 1) \\
\vdots & \vdots \\
\vdots & \vdots \\
\mathbf{Y(i-1)} \rightarrow (\Delta_i - 1) & (n_i - 1) \\
\mathbf{Yi} & n_i \\
\mathbf{Yi} \rightarrow 1 & (n_i + 1) \\
\vdots & \vdots \\
\vdots & \vdots \\
\mathbf{Y(k-2)} \rightarrow (\Delta_{(k-1)} - 1) & (n_{(k-1)} - 1) \\
\mathbf{Y(k-1)} & n_{(k-1)} \\
\mathbf{Y(k-1)} \rightarrow 1 & (n_{(k-1)} + 1) \\
\vdots & \vdots \\
\mathbf{Y(k-1)} \rightarrow (\Delta - 1) & (m - 1)
\end{array}\right]_{m \times m} \tag{11.9}
$$

### 11.3.1 Analysis

The formula of the vector $\mathbf{V}$ in Equation (11.5) includes shifts of $\mathbf{L}^{(j)}$, which finally reduce to the shifts of row $\mathbf{L}_0^{(0)}$ when the recursive formula is used. As all right shifts of the row $\mathbf{L}_0^{(0)} = \mathbf{M}_m$ are present in the original multiplication matrix $\mathbf{M}$, it is possible to represent the vector $\mathbf{V}$ as a sum of rows of this matrix. Except for the row $\mathbf{L}_0^{(0)}$ itself, the minimum shift is $\Delta$. Thus, in terms of the multiplication matrix $\mathbf{M}$, the shifted vectors will correspond to a subset of the rows of $\mathbf{M}$, with the minimum index being $(m + \Delta)$. After cancellations, the indices of the rows will be a subset $\mathcal{S}$ of the set of indices $m, (m + \Delta), (m + \Delta + 1), \ldots, (2m - 3), (2m - 2)$ The first row with smallest index can be used as a base for the addition and the rest will

be added to it. Thus the actual subset to be added to this base vector is $(\mathcal{S} - \min \mathcal{S})$, which will be called as $\mathcal{S}^*$. As row $\mathbf{M}_j$ has exactly $(2m-1-j)$ nonzero terms for $2m-2 \geq j \geq m$ and as adding each nonzero term requires a single XOR, the total number of XOR gates required to compute the first form of $\mathbf{V}$ will be equal to the sum

$$\sum_{j \in \mathcal{S}^*} (2m-1-j) \ .$$

The delay in the computation of the vector $\mathbf{V}$, can be minimized when the binary tree method is used to compute the summation in each entry. As there are at most $|\mathcal{S}|$ numbers to be added to compute any entry, the delay of computation will be $\lceil \log_2 |\mathcal{S}| \rceil \, T_X$, where $|\mathcal{S}|$ represents the order of the set $\mathcal{S}$.

When the recursive relations in Equation (11.7) is used, construction of $\mathbf{Z}_0$ requires only rewiring and $\mathbf{Z}_{n_i}$ is then constructed by adding the vector $\mathbf{V}$, to the vector formed by concatenation, using $(m-1)$ XOR gates, as the vector $\mathbf{V}$ has only $(m-1)$ nonzero terms. Thus a total of $(k-1)(m-1)$ XOR gates is needed to find all $\mathbf{Z}_{n_i}$ for $i = 1, 2, \ldots, (k-1)$. As the time needed to compute a single row $\mathbf{Z}_{n_i}$ is $T_X$, the total delay to compute all is $(k-1)T_X$.

The vectors $\mathbf{Y0}$ and $\mathbf{Y1}$ are then found using Equation (11.8) by only rewiring. Construction of $\mathbf{Z}$ in Equation (11.9) is also done using only rewiring, as it only consists of shifts.

To find the final result $c(x)$ via the product $\mathbf{c} = \mathbf{Zb}$, $m^2$ AND and $m(m-1)$ XOR gates is needed. Each coefficient of the final result $c(x)$ can be computed independently via the product $\mathbf{c}_i = \mathbf{Z}_i\mathbf{b}$. All the multiplications can be done in one level and the $m$ terms can be added using the binary tree method in $\lceil \log_2 m \rceil \, T_X$ time.

Thus, overall computation for the general case requires $m^2$ AND gates and

$$(m-1)(m+k-1) + \sum_{j \in \mathcal{S}^*} (2m-1-j) \tag{11.10}$$

XOR gates. And the total delay of the circuit is

$$T_A + (\lceil \log_2 |\mathcal{S}| \rceil + (k-1) + \lceil \log_2 m \rceil) \, T_X \ . \tag{11.11}$$

### 11.3.2 Example

Let the irreducible polynomial be $p(x) = x^7 + x^5 + x^3 + x + 1$. Then $m = 7$, $k = 4$, $n_3 = 5$ and $\Delta = 7 - 5 = 2$. The standard reduction will reduce the last $\Delta = 2$ rows of matrix $\mathbf{M}$ at each step, and finds the final $\mathbf{Z}$ matrix in $N[7, \Delta] = \left\lceil \frac{7-1}{\Delta} \right\rceil = \left\lceil \frac{6}{2} \right\rceil = 3$ steps.

The standard reduction operations performed can be summarized as follows:

$$\mathbf{M} = \begin{bmatrix} a_0 & 0 & 0 & 0 & 0 & 0 & 0 \\ a_1 & a_0 & 0 & 0 & 0 & 0 & 0 \\ a_2 & a_1 & a_0 & 0 & 0 & 0 & 0 \\ a_3 & a_2 & a_1 & a_0 & 0 & 0 & 0 \\ a_4 & a_3 & a_2 & a_1 & a_0 & 0 & 0 \\ a_5 & a_4 & a_3 & a_2 & a_1 & a_0 & 0 \\ a_6 & a_5 & a_4 & a_3 & a_2 & a_1 & a_0 \\ 0 & a_6 & a_5 & a_4 & a_3 & a_2 & a_1 \\ 0 & 0 & a_6 & a_5 & a_4 & a_3 & a_2 \\ 0 & 0 & 0 & a_6 & a_5 & a_4 & a_3 \\ 0 & 0 & 0 & 0 & a_6 & a_5 & a_4 \\ 0 & 0 & 0 & 0 & 0 & a_6 & a_5 \\ 0 & 0 & 0 & 0 & 0 & 0 & a_6 \end{bmatrix} \rightarrow \begin{bmatrix} a_0 & 0 & 0 & 0 & 0 & 0 & 0 \\ a_1 & a_0 & 0 & 0 & 0 & 0 & 0 \\ a_2 & a_1 & a_0 & 0 & 0 & 0 & 0 \\ a_3 & a_2 & a_1 & a_0 & 0 & 0 & 0 \\ a_4 & a_3 & a_2 & a_1 & a_0 & a_6 & a_5 \\ a_5 & a_4 & a_3 & a_2 & a_1 & (a_0 + a_6) & (a_5 + a_6) \\ a_6 & a_5 & a_4 & a_3 & a_2 & a_1 & (a_0 + a_6) \\ 0 & a_6 & a_5 & a_4 & a_3 & (a_2 + a_6) & (a_1 + a_5) \\ 0 & 0 & a_6 & a_5 & a_4 & a_3 & (a_2 + a_6) \\ 0 & 0 & 0 & a_6 & a_5 & (a_4 + a_6) & (a_3 + a_5) \\ 0 & 0 & 0 & 0 & a_6 & a_5 & (a_4 + a_6) \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

$$
\rightarrow \begin{bmatrix}
a_0 & 0 & 0 & 0 & 0 & 0 & 0 \\
a_1 & a_0 & 0 & 0 & 0 & 0 & 0 \\
a_2 & a_1 & a_0 & a_6 & a_5 & (a_4+a_6) & (a_3+a_5) \\
a_3 & a_2 & a_1 & (a_0+a_6) & (a_5+a_6) & (a_4+a_5+a_6) & (a_3+a_4+a_5+a_6) \\
a_4 & a_3 & a_2 & a_1 & (a_0+a_6) & (a_5+a_6) & (a_4+a_5+a_6) \\
a_5 & a_4 & a_3 & (a_2+a_6) & (a_1+a_5) & (a_0+a_4) & (a_3+a_6) \\
a_6 & a_5 & a_4 & a_3 & (a_2+a_6) & (a_1+a_5) & (a_0+a_4) \\
0 & a_6 & a_5 & (a_4+a_6) & (a_3+a_5) & (a_2+a_4) & (a_1+a_3) \\
0 & 0 & a_6 & a_5 & (a_4+a_6) & (a_3+a_5) & (a_2+a_4) \\
0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0
\end{bmatrix}
$$

$$
\rightarrow \begin{bmatrix}
a_0 & a_6 & a_5 & (a_4+a_6) & (a_3+a_5) & (a_2+a_4) & (a_1+a_3) \\
a_1 & (a_0+a_6) & (a_5+a_6) & (a_4+a_5+a_6) & (a_3+a_4+a_5+a_6) & (a_2+a_3+a_4+a_5) & (a_1+a_2+a_3+a_4) \\
a_2 & a_1 & (a_0+a_6) & (a_5+a_6) & (a_4+a_5+a_6) & (a_3+a_4+a_5+a_6) & (a_2+a_3+a_4+a_5) \\
a_3 & (a_2+a_6) & (a_1+a_5) & (a_0+a_4) & (a_3+a_6) & (a_2+a_5+a_6) & (a_1+a_4+a_5+a_6) \\
a_4 & a_3 & (a_2+a_6) & (a_1+a_5) & (a_0+a_4) & (a_3+a_6) & (a_2+a_5+a_6) \\
a_5 & (a_4+a_6) & (a_3+a_5) & (a_2+a_4) & (a_1+a_3) & (a_0+a_2) & (a_1+a_6) \\
a_6 & a_5 & (a_4+a_6) & (a_3+a_5) & (a_2+a_4) & (a_1+a_3) & (a_0+a_2) \\
0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0
\end{bmatrix}
$$

The first 7 rows the above result will form the **Z** matrix.

The proposed method will do the following to get the above result: First the vector $\mathbf{V}$ is found using Equation (11.5)

$$\mathbf{V} = \sum_{j=0}^{N[7,5]-1} \mathbf{L}_0^{(j)} = \mathbf{L}_0^{(0)} + \mathbf{L}_0^{(1)} + \mathbf{L}_0^{(2)} \ .$$

which can be computed using the recursive definition in Equation (11.6):

$$\mathbf{L}_0^{(j)} = \sum_{\substack{r_{(j-1)} > (7-n_i) \\ 3 \geq i \geq 0}} (\mathbf{L}_0^{(j-1)} \rightarrow (7-n_i)) \ .$$

Using this formula $\mathbf{L}_0^{(1)}$ and $\mathbf{L}_0^{(2)}$ can be reduced as follows:

$$\begin{aligned}
\mathbf{L}_0^{(1)} &= \sum_{\substack{3 \geq i \geq 0 \\ 6 > (7-n_i)}} (\mathbf{L}_0^{(0)} \rightarrow (7-n_i)) = (\mathbf{L}_0^{(0)} \rightarrow (7-5)) + (\mathbf{L}_0^{(0)} \rightarrow (7-3)) \\
&= (\mathbf{L}_0^{(0)} \rightarrow 2) + (\mathbf{L}_0^{(0)} \rightarrow 4) = \mathbf{M}_9 + \mathbf{M}_{11}
\end{aligned}$$

$$\begin{aligned}
\mathbf{L}_0^{(2)} &= \sum_{\substack{3 \geq i \geq 0 \\ (7-n_i) < 4}} (\mathbf{L}_0^{(1)} \rightarrow (7-n_i)) = \mathbf{L}_0^{(1)} \rightarrow (7-5) \\
&= ((\mathbf{L}_0^{(0)} \rightarrow 2) + (\mathbf{L}_0^{(0)} \rightarrow 4)) \rightarrow (7-5) = (\mathbf{L}_0^{(0)} \rightarrow 4) + (\mathbf{L}_0^{(0)} \rightarrow 6) \\
&= (\mathbf{L}_0^{(0)} \rightarrow 4) = \mathbf{M}_{11} \ .
\end{aligned}$$

Thus the final form of the vector $\mathbf{V}$ will be

$$\begin{aligned}
\mathbf{V} &= \mathbf{L}_0^{(0)} + \mathbf{L}_0^{(1)} + \mathbf{L}_0^{(2)} = \mathbf{M}_7 + (\mathbf{M}_9 + \mathbf{M}_{11}) + \mathbf{M}_{11} \\
&= \mathbf{M}_7 + \mathbf{M}_9 \\
&= \begin{bmatrix} 0 & a_6 & a_5 & a_4 & a_3 & a_2 & a_1 \end{bmatrix} \\
&\quad + \begin{bmatrix} 0 & 0 & 0 & a_6 & a_5 & a_4 & a_3 \end{bmatrix} \\
&= \begin{bmatrix} 0 & a_6 & a_5 & (a_4+a_6) & (a_3+a_5) & (a_2+a_4) & (a_1+a_3) \end{bmatrix} \ .
\end{aligned}$$

As can be seen from above, $\mathcal{S} = \{7, 9\}$ and $\mathcal{S}^* = \{9\}$.

Next compute $\mathbf{Z}_0$, $\mathbf{Z}_1$, $\mathbf{Z}_3$ and $\mathbf{Z}_5$ using the recursive relations in Equation (11.7):

$$\begin{aligned}
\mathbf{Z}_0 &= [a_0] \parallel [\mathbf{V}]_{1\times(7-1)} \\
&= \begin{bmatrix} a_0 & a_6 & a_5 & (a_4+a_6) & (a_3+a_5) & (a_2+a_4) & (a_1+a_3) \end{bmatrix}
\end{aligned}$$

$$\begin{aligned}
\mathbf{Z}_1 &= \left([\mathbf{M}_7]_{1\times1} \parallel [\mathbf{Z}_0 \rightarrow 1]_{1\times(7-1)}\right) + \mathbf{V} \\
&= \begin{bmatrix} a_1 & a_0 & a_6 & a_5 & (a_4+a_6) & (a_3+a_5) & (a_2+a_4) \end{bmatrix} \\
&\quad + \begin{bmatrix} 0 & a_6 & a_5 & (a_4+a_6) & (a_3+a_5) & (a_2+a_4) & (a_1+a_3) \end{bmatrix} \\
&= \begin{bmatrix} a_1 & (a_0+a_6) & (a_5+a_6) & (a_4+a_5+a_6) & (a_3+a_4+a_5+a_6) & (a_2+a_3+a_4+a_5) & (a_1+a_2+a_3+a_4) \end{bmatrix}
\end{aligned}$$

$$\begin{aligned}
\mathbf{Z}_3 &= \left([\mathbf{M}_{7+1}]_{1\times2} \parallel [\mathbf{Z}_1 \rightarrow 2]_{1\times(7-2)}\right) + \mathbf{V} \\
&= \begin{bmatrix} a_3 & a_2 & a_1 & (a_0+a_6) & (a_5+a_6) & (a_4+a_5+a_6) & (a_3+a_4+a_5+a_6) \end{bmatrix} \\
&\quad + \begin{bmatrix} 0 & a_6 & a_5 & (a_4+a_6) & (a_3+a_5) & (a_2+a_4) & (a_1+a_3) \end{bmatrix} \\
&= \begin{bmatrix} a_3 & (a_2+a_6) & (a_1+a_5) & (a_0+a_4) & (a_3+a_6) & (a_2+a_5+a_6) & (a_1+a_4+a_5+a_6) \end{bmatrix}
\end{aligned}$$

$$\begin{aligned}
\mathbf{Z}_5 &= \left([\mathbf{M}_{7+3}]_{1\times2} \parallel [\mathbf{Z}_3 \rightarrow 2]_{1\times(7-2)}\right) + \mathbf{V} \\
&= \begin{bmatrix} a_5 & a_4 & a_3 & (a_2+a_6) & (a_1+a_5) & (a_0+a_4) & (a_3+a_6) \end{bmatrix} \\
&\quad + \begin{bmatrix} 0 & a_6 & a_5 & (a_4+a_6) & (a_3+a_5) & (a_2+a_4) & (a_1+a_3) \end{bmatrix} \\
&= \begin{bmatrix} a_5 & (a_4+a_6) & (a_3+a_5) & (a_2+a_4) & (a_1+a_3) & (a_0+a_2) & (a_1+a_6) \end{bmatrix}
\end{aligned}$$

Extending $\mathbf{Z}_0$, $\mathbf{Z}_1$, $\mathbf{Z}_3$ and $\mathbf{Z}_5$ using Equation (11.8) will result in the following vectors:

$$\begin{aligned}
\mathbf{Y0} &= [\mathbf{M}_7]_{1\times(1-1)} \parallel \mathbf{Z}_0 \\
&= \mathbf{Z}_0 = \begin{bmatrix} a_0 & a_6 & a_5 & (a_4+a_6) & (a_3+a_5) & (a_2+a_4) & (a_1+a_3) \end{bmatrix} \\
\mathbf{Y1} &= [\mathbf{M}_8]_{1\times(2-1)} \parallel \mathbf{Z}_1 \\
&= \begin{bmatrix} a_2 & a_1 & (a_0+a_6) & (a_5+a_6) & (a_4+a_5+a_6) \\
& & (a_3+a_4+a_5+a_6) & (a_2+a_3+a_4+a_5) & (a_1+a_2+a_3+a_4) \end{bmatrix} \\
\mathbf{Y2} &= [\mathbf{M}_{10}]_{1\times(2-1)} \parallel \mathbf{Z}_3 \\
&= \begin{bmatrix} a_4 & a_3 & (a_2+a_6) & (a_1+a_5) \\
& & (a_0+a_4) & (a_3+a_6) & (a_2+a_5+a_6) & (a_1+a_4+a_5+a_6) \end{bmatrix} \\
\mathbf{Y3} &= [\mathbf{M}_{12}]_{1\times(2-1)} \parallel \mathbf{Z}_5 \\
&= \begin{bmatrix} a_6 & a_5 & (a_4+a_6) & (a_3+a_5) & (a_2+a_4) & (a_1+a_3) & (a_0+a_2) & (a_1+a_6) \end{bmatrix}
\end{aligned}$$

Finally the whole **Z** matrix can be constructed using the Equation (11.9):

$$\mathbf{Z} = \begin{bmatrix} \mathbf{Y0} \\ \mathbf{Y1} \\ (\mathbf{Y1} \rightarrow 1) \\ \mathbf{Y2} \\ (\mathbf{Y2} \rightarrow 1) \\ \mathbf{Y3} \\ (\mathbf{Y3} \rightarrow 1) \end{bmatrix}_{7\times 7}$$

$$= \begin{bmatrix} a_0 & a_6 & a_5 & (a_4+a_6) & (a_3+a_5) & (a_2+a_4) & (a_1+a_3) \\ a_1 & (a_0+a_6) & (a_5+a_6) & (a_4+a_5+a_6) & (a_3+a_4+a_5+a_6) & (a_2+a_3+a_4+a_5) & (a_1+a_2+a_3+a_4) \\ a_2 & a_1 & (a_0+a_6) & (a_5+a_6) & (a_4+a_5+a_6) & (a_3+a_4+a_5+a_6) & (a_2+a_3+a_4+a_5) \\ a_3 & (a_2+a_6) & (a_1+a_5) & (a_0+a_4) & (a_3+a_6) & (a_2+a_5+a_6) & (a_1+a_4+a_5+a_6) \\ a_4 & a_3 & (a_2+a_6) & (a_1+a_5) & (a_0+a_4) & (a_3+a_6) & (a_2+a_5+a_6) \\ a_5 & (a_4+a_6) & (a_3+a_5) & (a_2+a_4) & (a_1+a_3) & (a_0+a_2) & (a_1+a_6) \\ a_6 & a_5 & (a_4+a_6) & (a_3+a_5) & (a_2+a_4) & (a_1+a_3) & (a_0+a_2) \end{bmatrix}$$

Note that the final matrix found using our method is the same as the one found using the standard method.

There are 4 additions in the first computation of vector **V** and 3*6 additions in the computation of rows **Zi**. The vectors **Yi** and the construction of the matrix **Z** is done only using rewiring. To find the final result $c(x)$ via the product $\mathbf{c} = \mathbf{Zb}$, $7^2$ `AND` and $7 * 6 = 42$ `XOR` gates will be needed.

Thus, the overall computation for the general case requires 49 `AND` gates and $4 + 3 * 6 + 42 = 64$ `XOR` gates.

To compute the required `XOR` gates, one can alternately use the Equation (11.10) with $\mathcal{S}^* = \{9\}$, $m = 7$ and $k = 4$, which yields

$$(7-1)(7+4-1) + \sum_{j\in\{9\}} (14-1-j) = 60 + 4 = 64$$

XOR gates. Using $\mathcal{S} = \{7, 9\}$ in Equation (11.11), the total delay of the circuit turns out to be

$$T_A + (\lceil \log_2 2 \rceil + (4-1) + \lceil \log_2 7 \rceil) T_X = T_A + 7T_X \ .$$

## 11.4 Special Cases

When the irreducible polynomial has a multiple with fewer terms or some special form, the reduction of some higher terms can be simplified. These yield some simpler multipliers. Binomial, trinomial, all-one-polynomial (AOP) and equally-spaced-polynomial (ESP) cases will be studied.

### 11.4.1 Binomials

When $p(x)$ is binomial, it is always reducible, and a Galois Field can not be constructed using it. But still reduction by a binomial can be needed in some cases. For example while doing a reduction using the AOP

$$p(x) = x^{m-1} + x^{m-2} + x^{m-3} + \cdots + x^2 + x^1 + 1 \ ,$$

the $(x+1)$ multiple of it, which will be the binomial

$$(x+1)p(x) = x^m + 1 \ ,$$

can be used, to reduce the terms with order higher than $m-1$. Thus, binomial case will be presented to be used as a base for the other special cases and for the sake of completeness.

Let $p(x) = x^m + 1$. When we use this polynomial to reduce the rows of $\mathbf{L}^{(0)}$, each row will have a single child and that child will be in $\mathbf{U}$, as explained in the general case. Furthermore the row number of the child in $\mathbf{U}$ will be the same as the reduced row in $\mathbf{L}^{(0)}$. In that case, all the reduction can be done in a single step as follows:

$$
\begin{aligned}
\mathbf{Z} &= \mathbf{U}^{(0)} + \mathbf{L}^{(0)} \\
&= \begin{bmatrix}
a_0 & 0 & 0 & \cdots & 0 & 0 & 0 \\
a_1 & a_0 & 0 & \cdots & 0 & 0 & 0 \\
a_2 & a_1 & a_0 & \cdots & 0 & 0 & 0 \\
\vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\
a_{m-3} & a_{m-4} & a_{m-5} & \cdots & a_0 & 0 & 0 \\
a_{m-2} & a_{m-3} & a_{m-4} & \cdots & a_1 & a_0 & 0 \\
a_{m-1} & a_{m-2} & a_{m-3} & \cdots & a_2 & a_1 & a_0
\end{bmatrix} + \begin{bmatrix}
0 & a_{m-1} & a_{m-2} & \cdots & a_3 & a_2 & a_1 \\
0 & 0 & a_{m-1} & \cdots & a_4 & a_3 & a_2 \\
0 & 0 & 0 & \cdots & a_5 & a_4 & a_3 \\
\vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\
0 & 0 & 0 & \cdots & 0 & a_{m-1} & a_{m-2} \\
0 & 0 & 0 & \cdots & 0 & 0 & a_{m-1} \\
0 & 0 & 0 & \cdots & 0 & 0 & 0
\end{bmatrix} \\
&= \begin{bmatrix}
a_0 & a_{m-1} & a_{m-2} & \cdots & a_3 & a_2 & a_1 \\
a_1 & a_0 & a_{m-1} & \cdots & a_4 & a_3 & a_2 \\
a_2 & a_1 & a_0 & \cdots & a_5 & a_4 & a_3 \\
\vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\
a_{m-3} & a_{m-4} & a_{m-5} & \cdots & a_0 & a_{m-1} & a_{m-2} \\
a_{m-2} & a_{m-3} & a_{m-4} & \cdots & a_1 & a_0 & a_{m-1} \\
a_{m-1} & a_{m-2} & a_{m-3} & \cdots & a_2 & a_1 & a_0
\end{bmatrix}
\end{aligned}
$$

As can be seen above, we needed no gate to get the last matrix. And in fact all can be done by just working on a single vector. Define the intermediate vector $\mathbf{Y}$ as

$$
\begin{aligned}
\mathbf{Y} &= M_{m-1} \parallel [M_{m-1} \rightarrow 1]_{1\times(m-1)} \\
&= \begin{bmatrix} a_{m-1} & a_{m-2} & \cdots & a_2 & a_1 & a_0 & a_{m-1} & a_{m-2} & \cdots & a_2 & a_1 \end{bmatrix} .
\end{aligned}
$$

Then the whole $\mathbf{Z}$ matrix can be constructed as follows:

$$
\mathbf{Z} = \begin{bmatrix}
\mathbf{Y} \\
\mathbf{Y} \rightarrow 1 \\
\mathbf{Y} \rightarrow 2 \\
\vdots \\
\mathbf{Y} \rightarrow (m-3) \\
\mathbf{Y} \rightarrow (m-2) \\
\mathbf{Y} \rightarrow (m-1)
\end{bmatrix}_{m\times m} = \begin{bmatrix}
a_0 & a_{m-1} & a_{m-2} & \cdots & a_3 & a_2 & a_1 \\
a_1 & a_0 & a_{m-1} & \cdots & a_4 & a_3 & a_2 \\
a_2 & a_1 & a_0 & \cdots & a_5 & a_4 & a_3 \\
\vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\
a_{m-3} & a_{m-4} & a_{m-5} & \cdots & a_0 & a_{m-1} & a_{m-2} \\
a_{m-2} & a_{m-3} & a_{m-4} & \cdots & a_1 & a_0 & a_{m-1} \\
a_{m-1} & a_{m-2} & a_{m-3} & \cdots & a_2 & a_1 & a_0
\end{bmatrix}
$$

#### 11.4.1.1 Analysis

Since $\mathbf{Y}$ is never processed, and as the rows of matrix $\mathbf{Z}$ can be obtained by only rewiring, the final $\mathbf{Z}$ matrix costs nothing in terms of gates. To find the final result $c(x)$ via the product $\mathbf{c} = \mathbf{Zb}$, $m^2$ AND and $m(m-1)$ XOR gates will be needed.

While computing $\mathbf{c} = \mathbf{Zb}$ the multiplications can be done in one level and then the outcomes can be added using the binary tree method. Using this procedure the delay of the circuit will be

$$T_A + \lceil \log_2 m \rceil T_X \ .$$

### 11.4.2 Trinomials

Let the irreducible polynomial be the *trinomial*

$$p(x) = x^m + x^n + 1$$

Note that $\Delta = m - n$ in this case. The special case $n = \frac{m}{2}$ will be treated as a special case of the equally-spaced-polynomials (ESPs) later and will not be studied here.

First the overall processes for trinomials will be summarized and then a method will be proposed to find the same result efficiently. When this irreducible polynomial is used to reduce the rows of $\mathbf{L}^{(0)}$, each row will have two children. The one corresponding to the constant term $x^{n_0} = 1$, is guaranteed to be in $\mathbf{U}$, but the other might need to be further reduced depending on the position of the parent row. To simplify the notation and observe the regularity, two additional matrices will be used. The reductions due to the constant term will always be added to the matrix $\mathbf{X0}$, and the reductions due to the $x^n$ term will always be added to $\mathbf{X1}$. At the end of the first step, the following matrices will be produced:

$$\mathbf{Z}^{(1)} = \mathbf{U}^{(0)} + \mathbf{X0}^{(1)} + \mathbf{X1}^{(1)}$$

$$= \begin{bmatrix} a_0 & 0 & \cdots & 0 & 0 & 0 \\ a_1 & a_0 & \cdots & 0 & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ a_{m-3} & a_{m-4} & \cdots & a_0 & 0 & 0 \\ a_{m-2} & a_{m-3} & \cdots & a_1 & a_0 & 0 \\ a_{m-1} & a_{m-2} & \cdots & a_2 & a_1 & a_0 \end{bmatrix} + \begin{bmatrix} 0 & a_{m-1} & a_{m-2} & \cdots & a_2 & a_1 \\ 0 & 0 & a_{m-1} & \cdots & a_3 & a_2 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & a_{m-1} & a_{m-2} \\ 0 & 0 & 0 & \cdots & 0 & a_{m-1} \\ 0 & 0 & 0 & \cdots & 0 & 0 \end{bmatrix}$$

$$+ \begin{bmatrix} 0 & 0 & 0 & \cdots & 0 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & 0 & \cdots & 0 & 0 \\ 0 & a_{m-1} & a_{m-2} & \cdots & a_n & \cdots & a_2 & a_1 \\ 0 & 0 & a_{m-1} & \cdots & a_{n+1} & \cdots & a_3 & a_2 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & a_{m-1} & \cdots & a_{\Delta+1} & a_\Delta \end{bmatrix} \begin{matrix} 0 \\ \vdots \\ (n-1) \\ n \\ (n+1) \\ \vdots \\ (m-1) \end{matrix}$$

$$\mathbf{L}^{(1)} = \begin{bmatrix} 0 & \cdots & 0 & a_{m-1} & a_{m-2} & \cdots & a_{\Delta+2} & a_{\Delta+1} \\ 0 & \cdots & 0 & 0 & a_{m-1} & \cdots & a_{\Delta+3} & a_{\Delta+2} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & \cdots & 0 & 0 & 0 & \cdots & a_{m-1} & a_{m-2} \\ 0 & \cdots & 0 & 0 & 0 & \cdots & 0 & a_{m-1} \\ 0 & \cdots & 0 & 0 & 0 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & \cdots & 0 & 0 & 0 & \cdots & 0 & 0 \end{bmatrix} \begin{matrix} 0 \\ 1 \\ \vdots \\ (n-3) \\ (n-2) \\ (n-1) \\ \vdots \\ (m-2) \end{matrix}$$

As can be seen above, the new matrix $\mathbf{L}^{(1)}$ that will be reduced further is again triangular. That means the new children will again be in the same form, except that they will contain more zero terms at the beginning.

Thus it is clear that if the same procedure is recursively applied, as in the general case, the submatrix $\mathbf{U}^{(0)}$ will never change and the forms of the matrices **Xi** and **L**

will remain the same after every step, i.e., **Xi** will all be trapezoidal and **L** will be triangular. The entries that are zero and outside the indicated geometric regions after the first iteration, will remain zero. Only the values inside these regions will be changed during the rest of the reduction. The number of nonzero rows of **L** after step $j$, i.e., number of nonzero rows in $\mathbf{L}^{(j)}$ can be denoted by

$$r_j = (m-1) - j(m-n) = (m-1) - j\Delta \ ,$$

as there are $(m-1)$ nonzero rows initially, and the number of rows is reduced by $\Delta = (m-n)$ after each step. Thus it will take

$$N[m,\Delta] = \left\lceil \frac{m-1}{\Delta} \right\rceil$$

steps to reduce the whole matrix $\mathbf{L}^{(0)}$. This number is also equal to the number of nonzero terms in the row $\mathbf{L}_0^{(0)}$ at the end of step $j$, i.e., number of nonzero terms in the row $\mathbf{L}_0^{(j)}$ for $j = 1, 2, \ldots, N[m,\Delta]-1$. Note that, similar to the general case, the range of $j$ does not include $j = N[m,\Delta]$, as the number of nonzero rows becomes zero after step $N[m,\Delta]$, but the number $r_j$ will be negative for $j = N[m,\Delta]$.

When a more detailed analysis is performed, as in the general case, one will see that all the submatrices formed by the nonzero rows of the matrices produced after the first iteration are Toeplitz matrices. When the matrix $\mathbf{L}^{(1)}$ is reduced further, the children will be added to the nonzero rows of the matrices $\mathbf{X0}^{(1)}$, $\mathbf{X1}^{(1)}$, and $\mathbf{L}^{(2)}$ which are Toeplitz submatrices. As the sum of Toeplitz matrices is again a Toeplitz matrix [16], the submatrices formed by the nonzero rows will all be Toeplitz submatrices again. And because these are special Toeplitz matrices, computing only the first nonzero rows of $\mathbf{X0}^{(f)}$ and $\mathbf{X1}^{(f)}$ will be enough to reconstruct them. Similarly the matrix **M**, and hence the submatrices, $\mathbf{U}^{(0)}$ and $\mathbf{L}^{(0)}$, can be constructed using only the row $\mathbf{M}_{m-1}$. Furthermore since the first row of $\mathbf{X0}^{(f)}$ and the $n^{th}$ row of $\mathbf{X1}^{(f)}$ are identical, it is enough to compute only one of them. Thus it will suffice to work on only $\mathbf{X0}_0^{(f)}$.

Here is how $\mathbf{X0}_0^{(f)}$ is computed:

$$\sum_{j=0}^{N[m,\Delta]-1} \mathbf{L}_0^{(j)} \ = \ \mathbf{L}_0^{(0)} + \mathbf{L}_0^{(1)} + \cdots + \mathbf{L}_0^{(N[m,\Delta]-1)} \ .$$

But since for trinomials the recursive relation in Equation (11.6) simplifies to

$$\mathbf{L}_0^{(j)} = \sum_{\substack{r_{(j-1)} \geq (m-n_i) \\ 1 \geq i \geq 0}} (\mathbf{L}_0^{(j-1)} \rightarrow (m-n_i)) = (\mathbf{L}_0^{(j-1)} \rightarrow (m-n)) = (\mathbf{L}_0^{(j-1)} \rightarrow \Delta) \ ,$$

$\mathbf{X0}_0^{(f)}$ becomes

$$\sum_{j=0}^{N[m,\Delta]-1} (\mathbf{L}_0^{(0)} \rightarrow j\Delta) = \mathbf{L}_0^{(0)} + (\mathbf{L}_0^{(0)} \rightarrow \Delta) + \cdots + (\mathbf{L}_0^{(0)} \rightarrow (N[m,\Delta]-1)\Delta) \ .$$

This will be used with $\mathbf{U}^{(0)}$ to construct the final matrix$\mathbf{Z}$. First, the rows $\mathbf{Z}_0$ and $\mathbf{Z}_n$ are formed by adding the corresponding rows of the matrix $\mathbf{U}^{(0)}$ to $\mathbf{X0}_0^{(f)}$ and $\mathbf{X1}_n^{(f)}$ and then they are extended to larger vectors $\mathbf{Y0}$ and $\mathbf{Y1}$ by concatenating the proper parts of $\mathbf{U}^{(0)}$ to their beginning, so that the shifts of $\mathbf{Y0}$ and $\mathbf{Y1}$ produce the rows below them. This will simplify the construction of the matrix $\mathbf{Z}$. To further simplify the representations, $\mathbf{X0}_0^{(f)}$ and $\mathbf{X1}_n^{(f)}$, which are all identical, will be represented by the vector $\mathbf{V}$. And instead of referring to $\mathbf{U}^{(0)}$ and $\mathbf{L}^{(0)}$, the original multiplication matrix $\mathbf{M}$, or its entries $a_i$ will be used.

The vector $\mathbf{V}$ will be defined as

$$\begin{aligned} \mathbf{V} &= \sum_{j=0}^{N[m,\Delta]-1} (\mathbf{L}_0^{(0)} \rightarrow j\Delta) = \sum_{j=0}^{N[m,\Delta]-1} \mathbf{M}_{m+j\Delta} \\ &= \begin{bmatrix} 0 & v_{m-1} & v_{m-2} & \cdots & v_3 & v_2 & v_1 \end{bmatrix} \ . \end{aligned}$$

Before proceeding any further consider the following equality:

$$\begin{aligned} \mathbf{V} &= \sum_{j=0}^{N[m,\Delta]-1} (\mathbf{L}_0^{(0)} \rightarrow j\Delta) \\ &= \mathbf{L}_0^{(0)} + \sum_{j=1}^{N[m,\Delta]-1} (\mathbf{L}_0^{(0)} \rightarrow j\Delta) \\ &= \mathbf{L}_0^{(0)} + \left[ \sum_{j=1}^{N[m,\Delta]-1} (\mathbf{L}_0^{(0)} \rightarrow j\Delta) \right] + (\mathbf{L}_0^{(0)} \rightarrow N[m,\Delta]\Delta) \\ &= \mathbf{L}_0^{(0)} + \sum_{j=1}^{N[m,\Delta]} (\mathbf{L}_0^{(0)} \rightarrow j\Delta) \end{aligned}$$

$$
\begin{aligned}
&= \mathbf{L}_0^{(0)} + \left[ \sum_{j=0}^{N[m,\Delta]-1} (\mathbf{L}_0^{(0)} \to j\Delta) \right] \to \Delta \\
&= \mathbf{L}_0^{(0)} + (\mathbf{V} \to \Delta)
\end{aligned}
$$

Note that the row $(\mathbf{L}_0^{(0)} \to N[m,\Delta]\Delta)$ added to the sum above is a zero vector since $\mathbf{L}_0^{(0)}$ has a zero at the first entry and is shifted at least $(m-1)$ positions to the right.

The above equality suggests the following recursive formula for the computation of the entries of the vector $\mathbf{V}$:

$$
v_i = \begin{cases} 0 & i = m \ , \\ a_i & (m-1) \geq i > (n-1) \ , \\ a_i + v_{i+\Delta} & (n-1) \geq i \geq 1 \ . \end{cases} \tag{11.12}
$$

Formulas for $\mathbf{Z}_0$ and $\mathbf{Z}_n$ can be found using the general recursive formula in Equation (11.7):

$$
\begin{aligned}
\mathbf{Z}_0 &= [a_0] \parallel [\mathbf{V}]_{1\times(m-1)} \\
&= [\ a_0 \ v_{m-1} \ v_{m-2} \ \cdots \ v_3 \ v_2 \ v_1 \ ] \\
\mathbf{Z}_n &= (\mathbf{M}_n + (\mathbf{V} \to n)) + \mathbf{V} \\
&= [\ a_n \ (a_{n-1} + v_{m-1}) \ \cdots \ (a_0 + v_\Delta) \ (v_{m-1} + v_{\Delta-1}) \ \cdots \ (v_{n+1} + v_1) \ ]
\end{aligned}
$$

However, there are further simplifications here, which can be seen by comparing the entries of $\mathbf{Z}_n$ and $\mathbf{Z}_0$ carefully. The first $n$ entries in $\mathbf{Z}_n$ are identical to the last $n$ entries of $\mathbf{Z}_0$, and thus need not be computed again, rewiring will be enough. This can be proven using the recursive formula of the vector $\mathbf{V}$ in Equation (11.12)

$$
\begin{aligned}
(\mathbf{Z}_0 \leftarrow \Delta) &= (\mathbf{M}_0 \leftarrow \Delta) + (\mathbf{V} \leftarrow \Delta) \\
&= \mathbf{V} \leftarrow \Delta \\
&= (\mathbf{L}_0^{(0)} + (\mathbf{V} \to \Delta)) \leftarrow \Delta \\
&= (\mathbf{L}_0^{(0)} \leftarrow \Delta) + ((\mathbf{V} \to \Delta) \leftarrow \Delta) \\
&= \mathbf{M}_n + ((\mathbf{V} \to \Delta) \leftarrow \Delta) \ .
\end{aligned}
$$

During the double shift, the first $n$ entries of the second term above is preserved, i.e., they are the same as in vector $\mathbf{V}$. Thus the difference between the first $n$ entries of

the vectors $(\mathbf{Z}_0 \leftarrow \Delta)$ and $\mathbf{Z}_n$ is only the vector $(\mathbf{V} \rightarrow n)$. But since the first $(n+1)$ entries of the vector $(\mathbf{V} \rightarrow n)$ are all zero, the vectors $(\mathbf{Z}_0 \leftarrow \Delta)$ and $\mathbf{Z}_n$ agree in the first $n$ positions. Then during the computation of $\mathbf{Z}_n$, the first $n$ positions can be produced by rewiring the last $n$ entries of $\mathbf{Z}_0$ or $\mathbf{V}$, as they are the same, i.e., $[\mathbf{V}]_{1\times n}$. Note also that the last $\Delta$ entries of the sum $(\mathbf{M}_n + (\mathbf{V} \rightarrow n))$ is equal to

$$[(\mathbf{Z}_0 \rightarrow n)]_{1\times\Delta} = [\ a_0\ \ v_{m-1}\ \ \cdots\ \ v_{n+1}\ ] = [\ a_0\ \ a_{m-1}\ \ \cdots\ \ a_{n+1}\ ]\ .$$

Note that the Equation (11.12) is used to substitute $v_i = a_i$ for $i = (n+1), \ldots, (m-1)$.

So here is the summary of the proposed method:
First compute the vector $\mathbf{V}$ using the recursive definition in Equation (11.12).
Then compute $\mathbf{Z}_0$ and $\mathbf{Z}_n$ using:

$$\begin{aligned}
\mathbf{Z}_0 &= [a_0] \parallel [\mathbf{V}]_{1\times(m-1)} = [\ a_0\ \ v_{m-1}\ \ v_{m-2}\ \ \cdots\ \ v_3\ \ v_2\ \ v_1\ ] \\
\mathbf{Z}_n &= [\mathbf{V}]_{1\times n} \parallel \left([\mathbf{V}]_{1\times\Delta} + [(\mathbf{Z}_0 \rightarrow n)]_{1\times\Delta}\right) \\
&= [\ v_n\ \ v_{n-1}\ \ \cdots\ \ v_1\ ] \parallel \\
&\qquad \left([\ v_\Delta\ \ v_{\Delta-1}\ \ \cdots\ \ v_1\ ] + [\ a_0\ \ a_{m-1}\ \ \cdots\ \ a_{n+1}\ ]\right) \\
&= [\ a_n\ \ v_{n-1}\ \ \cdots\ \ v_1\ \ (a_0 + v_\Delta)\ \ (a_{m-1} + v_{\Delta-1})\ \ \cdots\ \ (a_{n+1} + v_1)\ ]
\end{aligned} \tag{11.13}$$

Next find **Y0** and **Y1** by extending $\mathbf{Z}_0$ and $\mathbf{Z}_n$:

$$\begin{aligned}
\mathbf{Y0} &= [\mathbf{M}_m]_{1\times(n-1)} \parallel \mathbf{Z}_0 \\
&= [\ a_{n-1}\ \ a_{n-2}\ \ \cdots\ \ a_1\ \ a_0\ \ v_{m-1}\ \ v_{m-2}\ \ \cdots\ \ v_2\ \ v_1\ ] \\
\mathbf{Y1} &= [\mathbf{M}_{m+n}]_{1\times(\Delta-1)} \parallel \mathbf{Z}_n \\
&= [\ a_{m-1}\ \ a_{m-2}\ \ \cdots\ \ a_{n+1}\ \ a_n\ \ v_{n-1}\ \ \cdots \\
&\qquad \cdots\ \ v_1\ \ (a_0 + v_\Delta)\ \ (a_{m-1} + v_{\Delta-1})\ \ \cdots\ \ (a_{n+1} + v_1)\ ]
\end{aligned} \tag{11.14}$$

Finally the whole $\mathbf{Z}$ matrix can be constructed as follows:

$$\mathbf{Z} = \begin{bmatrix} \mathbf{Y0} \\ \mathbf{Y0} \rightarrow 1 \\ \vdots \\ \mathbf{Y0} \rightarrow (n-1) \\ \mathbf{Y1} \\ \mathbf{Y1} \rightarrow 1 \\ \vdots \\ \mathbf{Y1} \rightarrow (\Delta - 2) \\ \mathbf{Y1} \rightarrow (\Delta - 1) \end{bmatrix}_{m \times m} = \begin{bmatrix} a_0 & v_{m-1} & \cdots & v_{\Delta+1} & v_{\Delta} & \cdots & v_1 \\ a_1 & a_0 & \cdots & v_{\Delta+2} & v_{\Delta+1} & \cdots & v_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ a_{n-1} & a_{n-2} & \cdots & a_0 & v_{m-1} & \cdots & v_n \\ a_n & v_{n-1} & \cdots & v_1 & (a_0 + v_{\Delta}) & \cdots & (a_{n+1} + v_1) \\ a_{n+1} & a_n & \cdots & v_2 & v_1 & \cdots & (a_{n+2} + v_2) \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ a_{m-2} & a_{m-3} & \cdots & v_{\Delta-1} & v_{\Delta-2} & \cdots & (a_{m-1} + v_{\Delta-1}) \\ a_{m-1} & a_{m-2} & \cdots & v_{\Delta} & v_{\Delta-1} & \cdots & (a_0 + v_{\Delta}) \end{bmatrix} \tag{11.15}$$

Note that the $v_{\Delta-i}$ type of entries for $i \geq 0$ in the last rows, is present only if their indices are less than $n$. Otherwise they will be taken as $a_{\Delta-i}$.

#### 11.4.2.1 Analysis

Computation of the vector $\mathbf{V}$ using the recursive relation in Equation (11.12) requires only one XOR gate for each of the last $(n-1)$ entries which sums up to a total of $(n-1)$ XOR gates. The total delay is $(N[m, \Delta] - 1)T_X$ as the entries can be computed in blocks of $\Delta$, no computation is needed for the first block and computations within each block can be done in parallel.

Construction of $\mathbf{Z}_0$ requires only rewiring. During the computation of $\mathbf{Z}_n$, the first $n$ positions are produced by rewiring, and only $\Delta$ XOR gates are needed to compute the last $\Delta$ entries as each requires a single addition. This computation takes $T_X$ time, as all can be added in parallel. The vectors $\mathbf{Y0}$ and $\mathbf{Y1}$ are then found using only rewiring. Construction of the matrix $\mathbf{Z}$ is also done using only rewiring, as it only consists of shifts. To find the final result $c(x)$ via the product $\mathbf{c} = \mathbf{Zb}$, $m^2$ AND and $m(m-1)$ XOR gates will be needed. And the delay for this last part is $T_A + \lceil \log_2 m \rceil T_X$ when the binary tree method is used to compute the sums.

Thus, overall computation for the trinomial case requires $m^2$ **AND** gates and

$$(n-1)+\Delta+m(m-1)=m^2-1$$

**XOR** gates. And the total delay of the circuit is

$$T_A+(N[m,\Delta]+\lceil\log_2 m\rceil)\,T_X=T_A+\left(\left\lceil\frac{m-1}{\Delta}\right\rceil+\lceil\log_2 m\rceil\right)T_X\ . \qquad (11.16)$$

#### 11.4.2.2 Example

Let the irreducible polynomial be $p(x)=x^7+x^4+1$. Then $m=7$, $n=4$. The standard reduction will reduce the last $\Delta=7-4=3$ rows of matrix $\mathbf{M}$ at each step, and finds the final $\mathbf{Z}$ matrix in $N[7,4]=\left\lceil\frac{7-1}{7-4}\right\rceil=2$ steps.

The operations performed by the regular reduction method can be summarized as follows:

$$\mathbf{M}=\begin{bmatrix}
a_0 & 0 & 0 & 0 & 0 & 0 & 0\\
a_1 & a_0 & 0 & 0 & 0 & 0 & 0\\
a_2 & a_1 & a_0 & 0 & 0 & 0 & 0\\
a_3 & a_2 & a_1 & a_0 & 0 & 0 & 0\\
a_4 & a_3 & a_2 & a_1 & a_0 & 0 & 0\\
a_5 & a_4 & a_3 & a_2 & a_1 & a_0 & 0\\
a_6 & a_5 & a_4 & a_3 & a_2 & a_1 & a_0\\
0 & a_6 & a_5 & a_4 & a_3 & a_2 & a_1\\
0 & 0 & a_6 & a_5 & a_4 & a_3 & a_2\\
0 & 0 & 0 & a_6 & a_5 & a_4 & a_3\\
0 & 0 & 0 & 0 & a_6 & a_5 & a_4\\
0 & 0 & 0 & 0 & 0 & a_6 & a_5\\
0 & 0 & 0 & 0 & 0 & 0 & a_6
\end{bmatrix}
\rightarrow
\begin{bmatrix}
a_0 & 0 & 0 & 0 & 0 & 0 & 0\\
a_1 & a_0 & 0 & 0 & 0 & 0 & 0\\
a_2 & a_1 & a_0 & 0 & 0 & 0 & 0\\
a_3 & a_2 & a_1 & a_0 & a_6 & a_5 & a_4\\
a_4 & a_3 & a_2 & a_1 & a_0 & a_6 & a_5\\
a_5 & a_4 & a_3 & a_2 & a_1 & a_0 & a_6\\
a_6 & a_5 & a_4 & a_3 & a_2 & a_1 & a_0\\
0 & a_6 & a_5 & a_4 & (a_3+a_6) & (a_2+a_5) & (a_1+a_4)\\
0 & 0 & a_6 & a_5 & a_4 & (a_3+a_6) & (a_2+a_5)\\
0 & 0 & 0 & a_6 & a_5 & a_4 & (a_3+a_6)\\
0 & 0 & 0 & 0 & 0 & 0 & 0\\
0 & 0 & 0 & 0 & 0 & 0 & 0\\
0 & 0 & 0 & 0 & 0 & 0 & 0
\end{bmatrix}$$

$$
\rightarrow \begin{bmatrix}
a_0 & a_6 & a_5 & a_4 & (a_3+a_6) & (a_2+a_5) & (a_1+a_4) \\
a_1 & a_0 & a_6 & a_5 & a_4 & (a_3+a_6) & (a_2+a_5) \\
a_2 & a_1 & a_0 & a_6 & a_5 & a_4 & (a_3+a_6) \\
a_3 & a_2 & a_1 & a_0 & a_6 & a_5 & a_4 \\
a_4 & (a_3+a_6) & (a_2+a_5) & (a_1+a_4) & (a_0+a_3+a_6) & (a_2+a_5+a_6) & (a_1+a_4+a_5) \\
a_5 & a_4 & (a_3+a_6) & (a_2+a_5) & (a_1+a_4) & (a_0+a_3+a_6) & (a_2+a_5+a_6) \\
a_6 & a_5 & a_4 & (a_3+a_6) & (a_2+a_5) & (a_1+a_4) & (a_0+a_3+a_6) \\
0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0
\end{bmatrix}
$$

The first 7 rows will form the $\mathbf{Z}$ matrix.

The proposed method will do the following:

First compute $\mathbf{V}$, using the recursive relations in Equation (11.12):

$$
\begin{aligned}
\mathbf{V} &= [\ 0\ ] \parallel [\ a_6\ \ a_5\ \ a_4\ ] \parallel [\ (a_3+v_{3+3})\ \ (a_2+v_{2+3})\ \ (a_1+v_{1+3})\ ] \\
&= [\ 0\ ] \parallel [\ a_6\ \ a_5\ \ a_4\ ] \parallel [\ (a_3+a_6)\ \ (a_2+a_5)\ \ (a_1+a_4)\ ] \\
&= [\ 0\ \ a_6\ \ a_5\ \ a_4\ \ (a_3+a_6)\ \ (a_2+a_5)\ \ (a_1+a_4)\ ]
\end{aligned}
$$

Then compute $\mathbf{Z}_0$ and $\mathbf{Z}_4$ using Equation (11.13):

$$
\mathbf{Z}_0 = [a_0] \parallel [\mathbf{V}]_{1\times(7-1)} = [\ a_0\ \ a_6\ \ a_5\ \ a_4\ \ (a_3+a_6)\ \ (a_2+a_5)\ \ (a_1+a_4)\ ]
$$

$$
\begin{aligned}
\mathbf{Z}_4 &= [\mathbf{V}]_{1\times 4} \parallel \left([\mathbf{V}]_{1\times 3} + [(\mathbf{Z}_0 \rightarrow 4)]_{1\times 3}\right) \\
&= [\ a_4\ \ (a_3+a_6)\ \ (a_2+a_5)\ \ (a_1+a_4)\ ] \parallel \\
&\qquad \Big( [\ (a_3+a_6)\ \ (a_2+a_5)\ \ (a_1+a_4)\ ] + [\ a_0\ \ a_6\ \ a_5\ ] \Big) \\
&= [\ a_4\ \ (a_3+a_6)\ \ (a_2+a_5)\ \ (a_1+a_4)\ \ (a_0+a_3+a_6)\ \ (a_2+a_5+a_6)\ \ (a_1+a_4+a_5)\ ]
\end{aligned}
$$

Next find the vectors **Y0** and **Y1** using Equation (11.14):

$$\begin{aligned}\mathbf{Y0} &= [\mathrm{M}_7]_{1\times(4-1)} \parallel \mathbf{Z}_0 \\ &= [\ a_3\ \ a_2\ \ a_1\ \ a_0\ \ a_6\ \ a_5\ \ a_4\ \ (a_3+a_6)\ \ (a_2+a_5)\ \ (a_1+a_4)\ ]\end{aligned}$$

$$\begin{aligned}\mathbf{Y1} &= [\mathrm{M}_{11}]_{1\times(3-1)} \parallel \mathbf{Z}_4 \\ &= [\ a_6\ \ a_5\ \ a_4\ \ (a_3+a_6)\ \ (a_2+a_5)\ \ (a_1+a_4) \\ &\qquad\qquad (a_0+a_3+a_6)\ \ (a_2+a_5+a_2)\ \ (a_1+a_4+a_5)\ ]\end{aligned}$$

Finally the whole **Z** matrix can be constructed using Equation (11.15):

$$\mathbf{Z} = \begin{bmatrix} \mathbf{Y0} \\ \mathbf{Y0}\rightarrow 1 \\ \mathbf{Y0}\rightarrow 2 \\ \mathbf{Y0}\rightarrow 3 \\ \mathbf{Y1} \\ \mathbf{Y1}\rightarrow 1 \\ \mathbf{Y1}\rightarrow 2 \end{bmatrix}_{7\times 7} = \begin{bmatrix} a_0 & a_6 & a_5 & a_4 & (a_3+a_6) & (a_2+a_5) & (a_1+a_4) \\ a_1 & a_0 & a_6 & a_5 & a_4 & (a_3+a_6) & (a_2+a_5) \\ a_2 & a_1 & a_0 & a_6 & a_5 & a_4 & (a_3+a_6) \\ a_3 & a_2 & a_1 & a_0 & a_6 & a_5 & a_4 \\ a_4 & (a_3+a_6) & (a_2+a_5) & (a_1+a_4) & (a_0+a_3+a_6) & (a_2+a_5+a_2) & (a_1+a_4+a_5) \\ a_5 & a_4 & (a_3+a_6) & (a_2+a_5) & (a_1+a_4) & (a_0+a_3+a_6) & (a_2+a_5+a_2) \\ a_6 & a_5 & a_4 & (a_3+a_6) & (a_2+a_5) & (a_1+a_4) & (a_0+a_3+a_6) \end{bmatrix}$$

Note that the final matrix found using our method is the same as the one found using the standard method.

There are 3 additions in the computation of the vector **V**. And $\mathbf{Z}_4$ is produced using rewiring for the first 4 entries and 3 additions for the last 3 entries. The vectors **Y0**, **Y1** and the construction of the matrix **Z** is done only using rewiring. To find the final result $c(x)$ via the product $\mathbf{c} = \mathbf{Zb}$, $7^2$ `AND` and $7*6 = 42$ `XOR` gates will be needed. Thus, the overall computation for the trinomial case requires 49 `AND` gates and $3+3+42 = 48$ `XOR` gates.

Note that the above results all agree with the results found theoretically.

The total delay of the circuit can be found using Equation (11.16) with $m = 7$ and $\Delta = 3$

$$T_A + \left(\left\lceil \frac{7-1}{3} \right\rceil + \lceil \log_2 7 \rceil\right) T_X = T_A + 5T_X \ .$$

### 11.4.3 Equally-Spaced-Polynomials

Let $p(x)$ be an *equally-spaced-polynomial* (ESP) with $n_i = i\Delta$ for $i = 0, 1, \ldots, k$ and $m = n_k = k\Delta$, i.e.,

$$\begin{aligned} p(x) &= x^{n_k} + x^{n_{k-1}} + \cdots + x^{n_1} + x^{n_0} \\ &= x^{k\Delta} + x^{(k-1)\Delta} + \cdots + x^{\Delta} + 1 \ . \end{aligned}$$

$\Delta$ is called the *spacing factor*. Note that $\Delta$ still shows the difference between the highest two orders as before, i.e., $\Delta = n_k - n_{(k-1)}$. When the row $\mathbf{M}_j$ of the initial matrix $\mathbf{M}$, is reduced, it produces children on rows

$$(j - \Delta), (j - 2\Delta), \ldots, (j - k\Delta) \ ,$$

for $(j \geq (k+1)\Delta)$ . Similarly when the child on row $(j - \Delta)$ is further reduced, the new children will be on rows

$$(j - 2\Delta), (j - 3\Delta), \ldots, (j - (k+1)\Delta) \ .$$

As all of these are identical the new ones will cancel all of the old ones. The only new child left will be the one on row $(j - (k+1)\Delta)$. Hence we can say that the only offspring of the row $\mathbf{M}_j$ for $j \geq (k+1)\Delta$ will be added to the row $(j - (k+1)\Delta)$.

This is because, what effectively done above is reducing the row using the $(x^{\Delta}+1)$ multiple of the original irreducible polynomial $p(x)$, which is

$$(x^{\Delta} + 1)p(x) \ = \ x^{(k+1)\Delta} + 1 \ .$$

All the rows in $\mathbf{L}^{(0)}$ except the first $\Delta$, can be reduced using this multiple. The first $\Delta$ rows has to be reduced using $p(x)$ itself. In terms of the matrix $\mathbf{M}$, this means that the rows $\mathbf{M}_{(k+1)\Delta}, \mathbf{M}_{(k+1)\Delta+1}, \ldots, \mathbf{M}_{2m-2}$ will be reduced using the multiple $(x^{\Delta} + 1)p(x)$ and the rows $\mathbf{M}_{k\Delta}, \mathbf{M}_{k\Delta+1}, \ldots, \mathbf{M}_{(k+1)\Delta-1}$ will be reduced using the original polynomial $p(x)$. Note that the first part of the reduction does not use any addition as all the entries reduced will be added to the upper triangular section of the matrix $\mathbf{U}^{(0)}$ which consist of zeros. The second part of the reduction can be managed using two different approaches.

#### 11.4.3.1 First Solution

As before the proposed method will only compute $\mathbf{Z}_{n_i} = \mathbf{Z}_{i\Delta}$, for $i = 0, \ldots, (k-1)$, but without using the vector $\mathbf{V}$ this time. First we define $v = (k-i-1)$ to simplify the notation. When the above procedure is used, the row $\mathbf{Z}_{i\Delta}$ becomes

$$\begin{aligned}
\mathbf{Z}_{i\Delta} &= (\mathbf{M}_{i\Delta} + \mathbf{M}_{i\Delta+(k+1)\Delta}) + \mathbf{M}_{k\Delta} \\
&= \begin{bmatrix} a_{i\Delta} & a_{i\Delta-1} & \cdots & a_0 & 0 & \cdots & 0 & a_{k\Delta-1} & \cdots & a_{(i+1)\Delta+1} \end{bmatrix} \\
&\quad + \begin{bmatrix} 0 & a_{k\Delta-1} & \cdots & a_{(v+1)\Delta} & a_{(v+1)\Delta-1} & \cdots & a_{v\Delta} & a_{v\Delta-1} & \cdots & a_1 \end{bmatrix} \\
&= \begin{bmatrix} a_{i\Delta} & (a_{i\Delta-1} + a_{k\Delta-1}) & \cdots & (a_0 + a_{(v+1)\Delta}) & a_{(v+1)\Delta-1} & \cdots \\
& & \cdots & a_{v\Delta} & (a_{k\Delta-1} + a_{v\Delta-1}) & \cdots & (a_{(i+1)\Delta+1} + a_1) \end{bmatrix},
\end{aligned}$$

where we combined the first two vectors, as no gate is needed while adding them. The combined vector has $\Delta$ consecutive zeros, except for the row $\mathbf{Z}_{(k-1)\Delta}$ which has only $\Delta - 1$. As a result of this and the leading zero in $\mathbf{M}_k\Delta$, computing $\mathbf{Z}_{i\Delta}$, for $i = 0, \ldots, (k-2)$ only requires $((k-1)\Delta - 1)$ additions, and $\mathbf{Z}_{(k-1)\Delta}$ requires $(k-1)\Delta$ additions.

However there are further simplifications in the computation due to same additions appearing twice during the whole process. Using the definition of $\mathbf{Z}_{i\Delta}$ the row $\mathbf{Z}_{v\Delta}$ is found as

$$\begin{aligned}
\mathbf{Z}_{v\Delta} = \begin{bmatrix} a_{v\Delta} & (a_{v\Delta-1} + a_{k\Delta-1}) \cdots & (a_0 + a_{(i+1)\Delta}) & a_{(i+1)\Delta-1} & \cdots \\
& \cdots & a_{i\Delta} & (a_{k\Delta-1} + a_{i\Delta-1}) & \cdots & (a_{(v+1)\Delta+1} + a_1) \end{bmatrix}.
\end{aligned}$$

As seen above, the first $v\Delta$ entries of the row $\mathbf{Z}_{v\Delta}$ are identical to the last $v\Delta$ entries of the row $\mathbf{Z}_{i\Delta}$, and the last $i\Delta$ entries of the row $\mathbf{Z}_{v\Delta}$ are identical to the first $i\Delta$ entries of the row $\mathbf{Z}_{i\Delta}$. Thus to obtain the row $\mathbf{Z}_{v\Delta}$, the only addition needed to be carried out is $(a_0 + a_{(i+1)\Delta})$, the rest can be constructed by rewiring the row $\mathbf{Z}_{i\Delta}$ and the original matrix $\mathbf{M}$.

For the case $k$ odd, the row $\mathbf{Z}_{\frac{(k-1)}{2}\Delta}$ is

$$\begin{aligned}
\mathbf{Z}_{\frac{(k-1)}{2}\Delta} = \begin{bmatrix} a_{\frac{(k-1)}{2}\Delta} & (a_{\frac{(k-1)}{2}\Delta-1} + a_{k\Delta-1}) & \cdots & (a_0 + a_{\frac{(k+1)}{2}\Delta}) & a_{\frac{(k+1)}{2}\Delta-1} & \cdots \\
& & \cdots & a_{\frac{(k-1)}{2}\Delta} & (a_{k\Delta-1} + a_{\frac{(k-1)}{2}\Delta-1}) & \cdots & (a_{\frac{(k+1)}{2}\Delta+1} + a_1) \end{bmatrix}.
\end{aligned}$$

As can be seen above the first $\frac{(k-1)}{2}\Delta$ and the last $\frac{(k-1)}{2}\Delta$ entries are identical. Thus it is enough to compute the first $\frac{(k-1)}{2}\Delta$ entries and the term $(a_0 + a_{\frac{(k+1)}{2}\Delta})$ only.

So the proposed method computes the rows of the final matrix $\mathbf{Z}$ directly using the following set of recursive relations:

$$\mathbf{Z}_{i\Delta} = \begin{cases} (\mathbf{M}_{i\Delta} + \mathbf{M}_{(k+i+1)\Delta}) + \mathbf{M}_{k\Delta} & 0 \le i \le \left\lfloor \frac{k-2}{2} \right\rfloor \\ [\mathbf{Z}_{v\Delta}]_{1\times i\Delta} \parallel [(a_0 + a_{(v+1)\Delta})\ a_{(v+1)\Delta-1} \cdots \\ \qquad \cdots a_{v\Delta}] \parallel [\mathbf{Z}_{v\Delta} \rightarrow (i+1)\Delta]_{1\times v\Delta} & \left\lceil \frac{k+1}{2} \right\rceil \le i \le (k-1)\Delta \\ [a_{i\Delta}\ (a_{i\Delta-1} + a_{k\Delta-1}) \cdots (a_0 + a_{(i+1)\Delta})\ a_{(i+1)\Delta-1} \\ \qquad \cdots a_{i\Delta}\ (a_{k\Delta-1} + a_{i\Delta-1}) \cdots (a_{(i+1)\Delta+1} + a_1)] & i = \frac{k-1}{2} \text{ for } k \text{ odd} \end{cases} \tag{11.17}$$

where for $k$ odd, the last $(i\Delta - 1)$ entries in the $i = \frac{k-1}{2}$ case is not computed, but rewired using the first part of the same vector.

Next find $\mathbf{Yi\Delta}$ for $i = 0, 1, \ldots, (k-1)$ by extending $\mathbf{Z}_{i\Delta}$:

$$\begin{aligned} \mathbf{Yi\Delta} &= [\mathbf{M}_{m+i\Delta}]_{1\times(\Delta-1)} \parallel \mathbf{Z}_{i\Delta} \\ &= [a_{(i+1)\Delta-1} \cdots a_{i\Delta}\ (a_{i\Delta-1} + a_{k\Delta-1}) \cdots (a_0 + a_{(v+1)\Delta})\ a_{(v+1)\Delta-1} \\ &\qquad \cdots a_{v\Delta}\ (a_{k\Delta-1} + a_{v\Delta-1}) \cdots (a_{(i+1)\Delta+1} + a_1)\ ] \end{aligned} \tag{11.18}$$

Finally, the whole $\mathbf{Z}$ matrix can be constructed as follows:

$$\mathbf{Z} = \begin{bmatrix} \mathbf{Y0} & 0 \\ \mathbf{Y0} \rightarrow 1 & 1 \\ \vdots & \vdots \\ \mathbf{Y0} \rightarrow (\Delta - 1) & (\Delta - 1) \\ \mathbf{Y\Delta} & \Delta \\ \mathbf{Y\Delta} \rightarrow 1 & (\Delta + 1) \\ \vdots & \vdots \\ \vdots & \vdots \\ \mathbf{Y(i-1)\Delta} \rightarrow (\Delta - 1) & (i\Delta - 1) \\ \mathbf{Yi\Delta} & i\Delta \\ \mathbf{Yi\Delta} \rightarrow 1 & (i\Delta + 1) \\ \vdots & \vdots \\ \vdots & \vdots \\ \mathbf{Y(k-2)\Delta} \rightarrow (\Delta - 1) & ((k-1)\Delta - 1) \\ \mathbf{Y(k-1)\Delta} & (k-1)\Delta \\ \mathbf{Y(k-1)\Delta} \rightarrow 1 & ((k-1)\Delta + 1) \\ \vdots & \vdots \\ \mathbf{Y(k-1)\Delta} \rightarrow (\Delta - 1) & (m-1) \end{bmatrix}_{m \times m} \tag{11.19}$$

#### 11.4.3.2 Analysis of the First Solution

For the case $k$ even, the rows $\mathbf{Z}_{i\Delta}$, with indices $i = 0, \ldots, \frac{k-2}{2}$ can be computed using $(k-1)\Delta - 1$ additions each, and then the rows for $i = \frac{k}{2}, \ldots, (k-1)$ can be computed using one addition each, which sums up to a total of

$$(\frac{k-2}{2} + 1)((k-1)\Delta - 1) + (k - 1 - \frac{k}{2} + 1) = \frac{k}{2}(k-1)\Delta$$

additions.

For the case $k$ odd, as the first $\frac{(k-1)}{2}\Delta$ and the last $\frac{(k-1)}{2}\Delta$ entries in the row $\mathbf{Z}_{\frac{(k-1)}{2}\Delta}$ are identical, it is enough to compute the first $\frac{(k-1)}{2}\Delta$ additions only. The rows $\mathbf{Z}_{i\Delta}$, with $i = 0, \ldots, \frac{(k-1)}{2} - 1$ can be computed using $(k-1)\Delta - 1$ additions each,

and then the rows with $i = \frac{(k+1)}{2}, \ldots, (k-1)$ can be computed using one addition each as in the previous case, which sums up to a total of

$$\frac{(k-1)}{2}((k-1)\Delta - 1) + \frac{(k-1)}{2}\Delta + [(k-1) - (\frac{(k-1)}{2} + 1) + 1] = \frac{(k-1)}{2}k\Delta$$

additions.

Thus in both cases $\frac{k(k-1)\Delta}{2} = \frac{m(k-1)}{2}$ additions, i.e., XOR gates are needed to compute the rows $\mathbf{Z}_{i\Delta}$ matrix. The vectors $\mathbf{Yi\Delta}$ and the final matrix $\mathbf{Z}$ can be found using only rewiring. Also $m^2 = k^2\Delta^2$ AND and $m(m-1)$ XOR gates are needed to find the final result $c(x)$ via the product $\mathbf{c} = \mathbf{Zb}$. Hence the overall computation for this special case requires $m^2$ AND gates and

$$\frac{m(k-1)}{2} + m(m-1) = m^2 + m\frac{(k-3)}{2}$$

XOR gates.

Note that the total delay of the circuit that computes the matrix $\mathbf{Z}$ is only $T_X$ as all entries only requires a single addition and can be computed in parallel. The final product $\mathbf{c} = \mathbf{Zb}$ can be computed by a doing products in the first layer and then using the binary tree method to add the results. The total delay of the circuit will then be

$$T_A + (1 + \lceil \log_2 m \rceil) T_X \ .$$

#### 11.4.3.3 Second Solution

There is also another approach that depends on the observation that the last vector used in the computation of $\mathbf{Z}_{i\Delta}$ is always the same row, i.e., $\mathbf{M}_{k\Delta} = \mathbf{M}_m$. Then the contribution of this vector can be computed separately and can be added to the contribution of the other vectors for each row. That is the result $c_{i\Delta} = \mathbf{Z}_{i\Delta}\mathbf{b}$ can be partitioned as

$$c_{i\Delta} = \mathbf{Z}_{i\Delta}\mathbf{b} = \mathbf{Z}'_{i\Delta}\mathbf{b} + \mathbf{Z}''_{i\Delta}\mathbf{b} \ ,$$

where $\mathbf{Z}'b$ contains the contribution of the first two vectors in the definition of $\mathbf{Z}_{i\Delta}$ in Equation (11.17) and $\mathbf{Z}''\mathbf{b}$ contains the contribution of the last vector. However,

since the results are not vectors but numbers, shifting cannot be applied to construct the rows of $\mathbf{c}$ with indices that are not multiples of $\Delta$. Then one has to construct the whole matrices $\mathbf{Z}'$ and $\mathbf{Z}''$. Still there will be simplifications as there are only $\Delta$ different rows in the matrix $\mathbf{Z}''$.

In terms of the row $\mathbf{c}_i$ this corresponds to the partition

$$\mathbf{c}_i = \mathbf{Z}_i\mathbf{b} = (\mathbf{M}_i + \mathbf{M}_{i+(k+1)\Delta})\mathbf{b} + \mathbf{M}_{k\Delta+j}\mathbf{b} \ , \tag{11.20}$$

where $i \equiv j \pmod{\Delta}$ and $0 \leq j < \Delta$.

The result $\mathbf{c}$ is then directly found by using these entries.

#### 11.4.3.4 Analysis of the Second Solution

As shown before the sum $(\mathbf{M}_i + \mathbf{M}_{i+(k+1)\Delta})$ requires no addition and has $\Delta$ consecutive zeros, except for the rows $\mathbf{Z}_{(k-1)\Delta+l}$ with $0 \leq l < \Delta$, where it has only $(\Delta-1-l)$ zeros.

Thus the first product requires

$$\begin{aligned} (k-1)\Delta\texttt{AND} \quad (k-1)\Delta - 1\texttt{XOR} \quad &\text{gates for} \quad i = 0, \ldots, (k-1)\Delta - 1, \\ ((k-1)\Delta + 1 + l)\texttt{AND} \quad ((k-1)\Delta + l)\texttt{XOR} \quad &\text{gates for} \quad i = (k-1)\Delta, \ldots, k\Delta - 1 \ . \end{aligned}$$

Since the row $\mathbf{M}_{k\Delta+j}$ has $(k\Delta-j-1)$ non-zero terms, the second product requires

$$(k\Delta - j - 1)\texttt{AND} \quad \text{and} \quad (k\Delta - j - 2)\texttt{XOR}$$

gates for $0 \leq j < \Delta$. Another XOR gate is needed for the final sum of each row. Thus to compute the result using this second approach

$$(k-1)\Delta((k-1)\Delta) + \left[\sum_{l=0}^{\Delta-1}(k-1)\Delta + 1 + l\right] + \left[\sum_{j=0}^{\Delta-1} k\Delta - j - 1\right] \ = \ k^2\Delta^2 = m^2$$

AND gates and

$$(k-1)\Delta((k-1)\Delta - 1) + \left[\sum_{l=0}^{\Delta-1}(k-1)\Delta + l\right] + \left[\sum_{j=0}^{\Delta-1} k\Delta - j - 2\right] + k\Delta \ = \ m^2 - \Delta$$

XOR gates are required.

All vector products can be carried out in parallel. The bottleneck for the delay is the last row in the first product, i.e., when $l = (\Delta - 1)$, where two full vectors of length $m$ are multiplied. When the binary tree method is used to compute the additions, the total delay becomes

$$T_A + (1 + \lceil \log_2 m \rceil) T_X \ ,$$

where $1T_X$ is the delay of the final vector sum.

#### 11.4.3.5 Comparison of the Two Solutions

The delay and the number of AND gates needed are equal in both solutions. For $k > 2$,

$$m^2 + m\frac{(k-3)}{2} \geq m^2 > m^2 - \Delta \ ,$$

i.e., the second solution uses less XOR gates. And for $k = 2$, i.e., for the special trinomials $x^m + x^{\frac{m}{2}} + 1$, both solutions uses the same number of XOR gates as

$$m^2 + m\frac{(k-3)}{2} = m^2 - \frac{m}{2} = m^2 - \Delta \ .$$

In conclusion, the second solution is always better, but for trinomials the first solution can also be used, as well.

#### 11.4.3.6 Example

Let the irreducible polynomial be the ESP

$$p(x) = x^6 + x^3 + 1 \ .$$

Then $\Delta = 3$ and $m = k\Delta = 6$.

As $k = 2$, the both solutions have the same efficiency. So to present and compare the solutions, both cases, along with the general reduction, will be presented.

The multiple $(x^3 + 1)p(x)$ will be

$$(x^3 + 1)p(x) \ = \ x^9 + 1 \ .$$

In the proposed methods, the rows $\mathbf{M}_9$ and $\mathbf{M}_{10}$, will be reduced using this multiple and the rows $\mathbf{M}_6$, $\mathbf{M}_7$ and $\mathbf{M}_8$ will be reduced using the original polynomial. The standard reduction will proceed as follows:

$$\mathbf{M} = \begin{bmatrix} a_0 & 0 & 0 & 0 & 0 & 0 \\ a_1 & a_0 & 0 & 0 & 0 & 0 \\ a_2 & a_1 & a_0 & 0 & 0 & 0 \\ a_3 & a_2 & a_1 & a_0 & 0 & 0 \\ a_4 & a_3 & a_2 & a_1 & a_0 & 0 \\ a_5 & a_4 & a_3 & a_2 & a_1 & a_0 \\ 0 & a_5 & a_4 & a_3 & a_2 & a_1 \\ 0 & 0 & a_5 & a_4 & a_3 & a_2 \\ 0 & 0 & 0 & a_5 & a_4 & a_3 \\ 0 & 0 & 0 & 0 & a_5 & a_4 \\ 0 & 0 & 0 & 0 & 0 & a_5 \end{bmatrix} \rightarrow \begin{bmatrix} a_0 & 0 & 0 & 0 & a_5 & a_4 \\ a_1 & a_0 & 0 & 0 & 0 & a_5 \\ a_2 & a_1 & a_0 & 0 & 0 & 0 \\ a_3 & a_2 & a_1 & a_0 & 0 & 0 \\ a_4 & a_3 & a_2 & a_1 & a_0 & 0 \\ a_5 & a_4 & a_3 & a_2 & a_1 & a_0 \\ 0 & a_5 & a_4 & a_3 & a_2 & a_1 \\ 0 & 0 & a_5 & a_4 & a_3 & a_2 \\ 0 & 0 & 0 & a_5 & a_4 & a_3 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

$$\rightarrow \begin{bmatrix} a_0 & a_5 & a_4 & a_3 & (a_2+a_5) & (a_1+a_4) \\ a_1 & a_0 & a_5 & a_4 & a_3 & (a_2+a_5) \\ a_2 & a_1 & a_0 & a_5 & a_4 & a_3 \\ a_3 & (a_2+a_5) & (a_1+a_4) & (a_0+a_3) & a_2 & a_1 \\ a_4 & a_3 & (a_2+a_5) & (a_1+a_4) & (a_0+a_3) & a_2 \\ a_5 & a_4 & a_3 & (a_2+a_5) & (a_1+a_4) & (a_0+a_3) \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

The first 6 rows form the matrix $\mathbf{Z}$.

The first solution starts by computing $\mathbf{Z}_0$ and $\mathbf{Z}_3$ first, using Equation (11.17):

$$
\begin{aligned}
\mathbf{Z}_0 &= (\mathbf{M}_0 + \mathbf{M}_9) + \mathbf{M}_6 \\
&= \Big( \big[\, a_0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \,\big] \\
&\quad + \big[\, 0 \quad 0 \quad 0 \quad 0 \quad a_5 \quad a_4 \,\big] \Big) \\
&\quad + \big[\, 0 \quad a_5 \quad a_4 \quad a_3 \quad a_2 \quad a_1 \,\big] \\
&= \big[\, a_0 \quad 0 \quad 0 \quad 0 \quad a_5 \quad a_4 \,\big] \\
&\quad + \big[\, 0 \quad a_5 \quad a_4 \quad a_3 \quad a_2 \quad a_1 \,\big] \\
&= \big[\, a_0 \quad a_5 \quad a_4 \quad a_3 \quad (a_2 + a_5) \quad (a_1 + a_4) \,\big]
\end{aligned}
$$

$$
\mathbf{Z}_3 = \big[\, a_3 \quad (a_2 + a_5) \quad (a_1 + a_4) \quad (a_0 + a_3) \quad a_2 \quad a_1 \,\big]
$$

**YiΔ** is then found using Equation (11.18):

$$
\begin{aligned}
\mathbf{Y0} &= [\mathbf{M}_6]_{1\times 2} \,\|\, \mathbf{Z}_0 \\
&= \big[\, a_2 \quad a_1 \quad a_0 \quad a_5 \quad a_4 \quad a_3 \quad (a_2 + a_5) \quad (a_1 + a_4) \,\big]
\end{aligned}
$$

$$
\begin{aligned}
\mathbf{Y1} &= [\mathbf{M}_9]_{1\times 2} \,\|\, \mathbf{Z}_3 \\
&= \big[\, a_5 \quad a_4 \quad a_3 \quad (a_2 + a_5) \quad (a_1 + a_4) \quad (a_0 + a_3) \quad a_2 \quad a_1 \,\big]
\end{aligned}
$$

Then the whole $\mathbf{Z}$ matrix can be constructed using Equation (11.19), as follows:

$$
\mathbf{Z} = \begin{bmatrix} \mathbf{Y0} \\ \mathbf{Y0} \to 1 \\ \mathbf{Y0} \to 2 \\ \mathbf{Y1} \\ \mathbf{Y1} \to 1 \\ \mathbf{Y1} \to 2 \end{bmatrix}_{7\times 7} = \begin{bmatrix} a_0 & a_5 & a_4 & a_3 & (a_2 + a_5) & (a_1 + a_4) \\ a_1 & a_0 & a_5 & a_4 & a_3 & (a_2 + a_5) \\ a_2 & a_1 & a_0 & a_5 & a_4 & a_3 \\ a_3 & (a_2 + a_5) & (a_1 + a_4) & (a_0 + a_3) & a_2 & a_1 \\ a_4 & a_3 & (a_2 + a_5) & (a_1 + a_4) & (a_0 + a_3) & a_2 \\ a_5 & a_4 & a_3 & (a_2 + a_5) & (a_1 + a_4) & (a_0 + a_3) \end{bmatrix}
$$

3 XOR gates are needed to construct $\mathbf{Z}_0$ and $\mathbf{Z}_3$. $m^2 = 36$ AND and $m(m-1) = 30$ XOR gates are needed to find the final result $c(x)$ via the product $\mathbf{c} = \mathbf{Zb}$. Hence the overall computation for this special case requires 36 AND gates and $3 + 30 = 33$ XOR gates. The total delay is $(T_A + 4T_X)$.

In the second solution, the column vector $\mathbf{Z'b}$ is computed first

$$\mathbf{Z'b} = \begin{bmatrix} a_0 & 0 & 0 & 0 & a_5 & a_4 \\ a_1 & a_0 & 0 & 0 & 0 & a_5 \\ a_2 & a_1 & a_0 & 0 & 0 & 0 \\ a_3 & a_2 & a_1 & a_0 & 0 & 0 \\ a_4 & a_3 & a_2 & a_1 & a_0 & 0 \\ a_5 & a_4 & a_3 & a_2 & a_1 & a_0 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \end{bmatrix} = \begin{bmatrix} a_0b_0 + a_5b_4 + a_4b_5 \\ a_1b_0 + a_0b_1 + a_5b_5 \\ a_2b_0 + a_1b_1 + a_0b_2 \\ a_3b_0 + a_2b_1 + a_1b_2 + a_0b_3 \\ a_4b_0 + a_3b_1 + a_2b_2 + a_1b_3 + a_0b_4 \\ a_5b_0 + a_4b_1 + a_3b_2 + a_2b_3 + a_1b_4 + a_0b_5 \end{bmatrix}$$

which requires 24 multiplications and 18 additions. Then the column vector $\mathbf{Z''b}$ is computed

$$\mathbf{Z''b} = \begin{bmatrix} 0 & a_5 & a_4 & a_3 & a_2 & a_1 \\ 0 & 0 & a_5 & a_4 & a_3 & a_2 \\ 0 & 0 & 0 & a_5 & a_4 & a_3 \\ 0 & a_5 & a_4 & a_3 & a_2 & a_1 \\ 0 & 0 & a_5 & a_4 & a_3 & a_2 \\ 0 & 0 & 0 & a_5 & a_4 & a_3 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \end{bmatrix} = \begin{bmatrix} a_5b_1 + a_4b_2 + a_3b_3 + a_2b_4 + a_1b_5 \\ a_5b_2 + a_4b_3 + a_3b_4 + a_2b_5 \\ a_5b_3 + a_4b_4 + a_3b_5 \\ a_5b_1 + a_4b_2 + a_3b_3 + a_2b_4 + a_1b_5 \\ a_5b_2 + a_4b_3 + a_3b_4 + a_2b_5 \\ a_5b_3 + a_4b_4 + a_3b_5 \end{bmatrix}$$

As noted before there are only $\Delta = 3$ different rows in matrix $\mathbf{Z''b}$, so computing it requires $5 + 4 + 3 = 12$ multiplications and $4 + 3 + 2 = 9$ additions, and adding it to the column vector $\mathbf{Z'}b$ requires 6 additions. Thus a total of $24 + 12 = 36$ `AND` and $18 + 9 + 6 = 33$ `XOR` gates are needed to find the final result $c(x)$ directly using the second solution. The total delay is $(T_A + 4T_X)$.

Since $k = 2$ for the trinomials both solutions required exactly the same number of gates and have the save delay.

### 11.4.4 All-One-Polynomials

Let the irreducible polynomial be the *all-one-polynomial* (AOP)

$$p(x) = x^{n_k} + x^{n_{k-1}} + \cdots + x^{n_1} + x^{n_0} = x^k + x^{(k-1)} + \cdots + x + 1$$

i.e. , $n_i = i$ for $i = 0, 1, \ldots, k$ and $m = n_k = k$.

This is a special case of ESP with $\Delta = 1$. So all the rows, except the first one, in the lower submatrix can be reduced using the multiple

$$(x+1)p(x) \;=\; x^{(k+1)} + 1 \;.$$

This produces the partial result

$$\begin{bmatrix}
a_0 & 0 & a_{m-1} & a_{m-2} & \cdots & a_3 & a_2 & a_1 \\
a_1 & a_0 & 0 & a_{m-1} & \cdots & a_4 & a_3 & a_2 \\
a_2 & a_1 & a_0 & 0 & \cdots & a_5 & a_4 & a_3 \\
a_3 & a_2 & a_1 & a_0 & \cdots & a_6 & a_5 & a_4 \\
\vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\
a_{m-3} & a_{m-4} & a_{m-5} & a_{m-6} & \cdots & a_0 & 0 & a_{m-1} \\
a_{m-2} & a_{m-3} & a_{m-4} & a_{m-5} & \cdots & a_1 & a_0 & 0 \\
a_{m-1} & a_{m-2} & a_{m-3} & a_{m-4} & \cdots & a_2 & a_1 & a_0 \\
0 & a_{m-1} & a_{m-2} & a_{m-3} & \cdots & a_3 & a_2 & a_1 \\
0 & 0 & 0 & 0 & \cdots & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & \cdots & 0 & 0 & 0 \\
\vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\
0 & 0 & 0 & 0 & \cdots & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & \cdots & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & \cdots & 0 & 0 & 0
\end{bmatrix}$$

The first row in lower submatrix has to be reduced using the original irreducible polynomial $p(x)$. One of the two solutions proposed in the ESP case can be used here.

#### 11.4.4.1 Analysis

As before, both solutions require $m^2$ **AND** gates and the delay is

$$T_A + (1 + \lceil \log_2 m \rceil)\, T_X$$

in both cases. The second solution requires $m^2 - 1$ **XOR** gates since $\Delta = 1$. The **XOR** gates needed to apply the first solution depends on the choice of $p(x)$, but it always

uses at least $m^2$ XOR gates, except for the case $p(x) = x^2 + x + 1$ which requires $m^2 - \frac{m}{2} = 3$ gates. So for all other cases the second solution is better.

#### 11.4.4.2 Example

Let the irreducible polynomial be the AOP

$$p(x) = x^{10} + x^9 + x^8 + x^7 + x^6 + x^5 + x^4 + x^3 + x^2 + x + 1 \ .$$

Then $m = k = 10$ and $\Delta = 1$. The multiple $(x+1)p(x)$ then will be

$$(x+1)p(x) \ = \ x^{11} + 1 \ .$$

The second method will compute the column vectors

$$\mathbf{Z'b} = \begin{bmatrix} a_0 & 0 & a_9 & a_8 & a_7 & a_6 & a_5 & a_4 & a_3 & a_2 \\ a_1 & a_0 & 0 & a_9 & a_8 & a_7 & a_6 & a_5 & a_4 & a_3 \\ a_2 & a_1 & a_0 & 0 & a_9 & a_8 & a_7 & a_6 & a_5 & a_4 \\ a_3 & a_2 & a_1 & a_0 & 0 & a_9 & a_8 & a_7 & a_6 & a_5 \\ a_4 & a_3 & a_2 & a_1 & a_0 & 0 & a_9 & a_8 & a_7 & a_6 \\ a_5 & a_4 & a_3 & a_2 & a_1 & a_0 & 0 & a_9 & a_8 & a_7 \\ a_6 & a_5 & a_4 & a_3 & a_2 & a_1 & a_0 & 0 & a_9 & a_8 \\ a_7 & a_6 & a_5 & a_4 & a_3 & a_2 & a_1 & a_0 & 0 & a_9 \\ a_8 & a_7 & a_6 & a_5 & a_4 & a_3 & a_2 & a_1 & a_0 & 0 \\ a_9 & a_8 & a_7 & a_6 & a_5 & a_4 & a_3 & a_2 & a_1 & a_0 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \\ b_8 \\ b_9 \end{bmatrix}$$

and

$$\mathbf{Z}_0''\mathbf{b} = \begin{bmatrix} 0 & a_9 & a_8 & a_7 & a_6 & a_5 & a_4 & a_3 & a_2 & a_1 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \\ b_8 \\ b_9 \end{bmatrix}$$

$$= a_9b_1 + a_8b_2 + a_7b_3 + a_6b_4 + a_5b_5 + a_4b_6 + a_3b_7 + a_2b_8 + a_1b_9 \ .$$

Computing $\mathbf{Z}'\mathbf{b}$ require $9 * 9 + 10 = 91$ **AND** gates and $8 * 9 + 9 = 81$ **XOR** gates. Similarly computing $\mathbf{Z}_0''\mathbf{b}$ require 9 **AND** gates and 8 **XOR** gates. The final result is found by adding the result of the product $\mathbf{Z}_0''\mathbf{b}$, to every row of $\mathbf{Z}'\mathbf{b}$, which uses another 10 **XOR** gates.

Thus the total number of gates required to find the result $\mathbf{c}$ is 100 **AND** gates and 99 **XOR** gates, which agrees with the theoretical results found.

The bottleneck operation for the delay is the multiplication $\mathbf{Z}_9'\mathbf{b}$, which takes $(T_A + \lceil \log_2 10 \rceil T_X)$ time, assuming that the binary tree method is used to add the terms. Adding $\mathbf{Z}_0''\mathbf{b}$ to this result takes another $T_X$ time. Thus the total delay of the circuit becomes $(T_A + 5T_X)$.

# Chapter 12
# CONCLUSIONS

This chapter summarizes the results of the thesis, lists the most significant contributions and finally discusses future research directions in this largely unexplored area.

## 12.1 Discussion of Results

Chapter 9 is based on the paper [19]. In this chapter, we proposed a table lookup based reduction method for performing the standard and Montgomery multiplication and squaring operations in $GF(2^k)$ using the polynomial basis. The proposed method yields word-level algorithms, enabling software implementations of the finite field arithmetic operations which find applications most notably in elliptic curve cryptography. We treated the special irreducible polynomials and the integer case in detail and gave the algorithmic details for these methods together with the complexity analysis in terms of the number of basic arithmetic operations. The proposed algorithm is more efficient than the previously published results, and in the case of special irreducible polynomials (particularly, trinomials), the proposed method reduces to already known algorithms found in the literature.

In the integer case, the right-to-left version of the algorithm works well, providing an efficient version of the Montgomery multiplication algorithm. However, the left-to-right version is not efficient mainly due to the fact that the most significant words of the integer multiples of the modulus are not unique.

Chapter 10 is based on the paper [18]. The proposed parallel algorithm requires $O(s \log s)$ arithmetic operations using $2s$ processors. Additionally there are some table lookup operations performed by the server, and there is also the communication

overhead. The sequential multiplication algorithm using the table lookup method given in [19] requires $s^2$ MULGF2 and $3s^2$ XOR operations, therefore, a total of $O(s^2)$ arithmetic operations. Thus, the efficiency of the parallel algorithm proposed turns out to be

$$\frac{s^2}{(s^2 \log s)} = \frac{1}{\log s} .$$

The parallel algorithm is highly suitable for hardware implementation. For example, by selecting $w = 8$ and $L = 40$, we can implement $GF(2^{160})$ arithmetic, which is desired in several applications of elliptic curve cryptography [46]. By selecting special irreducible polynomials, for example, trinomials or all-one-polynomials, we may not have a need for the tables $T$ and $\vec{T}$ and the reduction operation will be greatly simplified [19]. A hardware implementation of this algorithm would be highly useful.

Chapter 11 is based on the paper [17]. In this chapter, a general new method is introduced to implement Mastrovito multipliers for all polynomials. When applied to a given specific irreducible polynomial the space complexity, i.e., the number of gates required, and the time complexity, i.e., the total delay of the architecture, of the proposed method turns out to be very efficient. The special cases described have the same efficiency as the best known architectures proposed to date.

The multiplier for the special generating trinomial

$$x^m + x + 1$$

is shown to use $(m^2 - 1)$ XOR and $m^2$ AND gates [39, 40, 54, 55]. The method we proposed also requires $(m^2 - 1)$ XOR and $m^2$ AND gates, moreover, it is applicable for *general* trinomials. This result agrees with the best known architectures [31].

Paar [31] conjectured that the space complexity of the Mastrovito multiplier would be the same for all trinomials. While $(m^2-1)$ XOR and $m^2$ AND gates seemed like a natural lower bound for Mastrovito multipliers, and Koç and Sunar [31] proposed an architecture for trinomials of the form

$$x^m + x^{\frac{m}{2}} + 1 \ ,$$

that only requires $(m^2 - \frac{m}{2})$ XOR and $m^2$ AND gates. In this chapter, a more general multiplier that requires only $(m^2 - \Delta)$ XOR and $m^2$ AND gates is proposed for the

equally spaced polynomials, where $\Delta$ is the spacing factor (*see* page 129). Our multiplier improved the best-known algorithm for this case, which uses $(m^2-1)$ XOR and $m^2$ AND gates [21]. Furthermore, the method can also be applied to the special forms of the equally-spaced-polynomials, such as trinomials of the form

$$x^m + x^{\frac{m}{2}} + 1$$

and the all-one-polynomials. In both cases our algorithm achieves the best known lower bounds, as given in [32, 31].

## 12.2 Summary of Contributions

Below is the summary of the contributions made:

- A new word-level serial reduction algorithm for integers and polynomials (alternative to division-based reduction)
- A new word-level serial reduction algorithm for polynomials (alternative to the Montgomery multiplication)
- Two simplified versions of the above algorithms applicable to trinomials and all-one-polynomials
- A new word-level parallel multiplication algorithm for polynomials using PRNS (two versions)
- A general Mastrovito multiplier architecture for polynomials
- Four simplified versions of the above architecture for special polynomials
- Improvement on the best-known Mastrovito multiplier architecture for equally-spaced-polynomials
- Detailed analyses of all methods proposed

In some special cases [45, 32], the canonical basis is equivalent to the normal basis, and the proposed multipliers can also be used for normal basis multiplication by

rewiring. However, we believe there will be further improvements if the architectures are designed specifically for the normal basis.

Our analyses ignore the communication overhead, which can be crucial in some cases. Also the design of the algorithms were not architecture oriented. Future research with these in mind might yield more efficient algorithms and totally different approaches.

# BIBLIOGRAPHY

[1] G. B. Agnew, R. C. Mullin, I. Onyszchuk, and S. A. Vanstone. An implementation for a fast public-key cryptosystem. *Journal of Cryptology*, 3(2):63–79, 1991.

[2] E. Bach and J. Shallit. *Algorithmic Number Theory: Efficient Algorithms*, volume 1. Cambridge, MA: MIT Press, 1996.

[3] M. Ben-Or. Probabilistic algorithms in finite fields. In *22nd Annual Symposium on Foundations of Computer Science*, pages 394–398, 1981.

[4] A. Bender and G. Castagnoli. On the implementation of elliptic curve cryptosystems. In G. Brassard, editor, *Advances in Cryptology — CRYPTO 89, Proceedings*, Lecture Notes in Computer Science, No. 435, pages 186–192. New York, NY: Springer-Verlag, 1989.

[5] J. W. S. Cassels. *Lectures on Elliptic Curves.* New York, NY: Cambridge University Press, 1991.

[6] D. Coppersmith. Fast evaluation of logarithms in fields of characteristic two. *IEEE Transactions on Information Theory*, 30:587–594, 1984.

[7] D. Coppersmith, a. Odlyzko, and R. Schroeppel. Discrete logarithms in $GF(2^t)$. *Algorithmica*, 1:1–15, 1986.

[8] R. Crandall. Method and apparatus for public key exchange in a cryptographic system. U.S. Patent Number 5,159,632, October 1992.

[9] M. Diab. Systolic architectures for multiplication over finite field $gf(2^m)$. In *Proceedings of AAECC-9*, Lecture Notes in Computer Science, No. 508, pages 329–340. New York, NY: Springer-Verlag, 1991.

[10] W. Diffie and M. E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22:644–654, November 1976.

[11] T. ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory*, 31(4):469–472, July 1985.

[12] M. Feng. A VLSI architecture for fast inversion in $GF(2^m)$. *IEEE Transactions on Computers*, 38:1383–1386, 1989.

[13] S. Gao and H. W. Lenstra. Optimal normal bases. *Design, Codes and Cryptography*, 2:315–323, 1992.

[14] W. Geiselmann and D. Gollmann. VLSI degthe implementation of elliptic curve cryptosystems. In J. Seberry and J. Pieprzyk, editors, *Advances in Cryptology — AUSCRYPT 90*, Lecture Notes in Computer Science, No. 453, pages 398–405. New York, NY: Springer-Verlag, 1990.

[15] S. Goldwasser and J. Kilian. Almost all primes can be quickly certified. In *Proceedings of the 18th Annual ACM Symposium on Theory of Computing*, pages 316–329, 1986.

[16] G. H. Golub and C. F. van Loan. *Matrix Computations*. Baltimore, MD: The Johns Hopkins University Press, 2nd edition, 1989.

[17] A. Halbutoğulları and Ç. K. Koç. A general mastrovito multiplier. Manuscript being prepared for publication, October 1998.

[18] A. Halbutoğulları and Ç. K. Koç. Parallel finite field multiplication using polynomial residue number systems. Manuscript submitted for publication, June 1998.

[19] A. Halbutoğulları and Ç. K. Koç. A reduction method for multiplication in finite fields. Manuscript submitted for publication, May 1998.

[20] G. Harper, A. Menezes, and S. Vanstone. Public-key cryptosystems with very small key lengths. In R. A. Rueppel, editor, *Advances in Cryptology — EURO-*

*CRYPT 92*, Lecture Notes in Computer Science, No. 658, pages 163–173. New York, NY: Springer-Verlag, 1992.

[21] M. A. Hasan, M. Z. Wang, and V. K. Bhargava. Modular construction of low complexity parallel multipliers for a class of finite fields $GF(2^m)$. *IEEE Transactions on Computers*, 41(8):962–971, August 1992.

[22] M. Hellman and M. Reyneri. Fast computation of discrete logarithms in $GF(q)$. In *Advances in Cryptology — CRYPTO 82, Proceedings*, pages 3–13. Plenum Press, 1983.

[23] IEEE P1363. Standard specifications for public-key cryptography. Draft version 7, September 1998.

[24] D. Jungnickel, editor. *Finite Fields: Structure and Arithmetics.* Bibliographisches Institut, Mannheim, 1993.

[25] D. E. Knuth. *The Art of Computer Programming: Seminumerical Algorithms*, volume 2. Reading, MA: Addison-Wesley, Third edition, 1998.

[26] N. Koblitz. *Introduction to Elliptic Curves and Modular Forms.* New York, NY: Springer-Verlag, 1984.

[27] N. Koblitz. Elliptic curve cryptosystems. *Mathematics of Computation*, 48(177):203–209, January 1987.

[28] N. Koblitz. Constructing elliptic curve cryptosystems in characteristic 2. In *Advances in Cryptology — CRYPTO 90*, Lecture Notes in Computer Science, No. 537, pages 156–167. New York, NY: Springer-Verlag, 1991.

[29] N. Koblitz. *A Course in Number Theory and Cryptography.* New York, NY: Springer-Verlag, Second edition, 1994.

[30] Ç. K. Koç and T. Acar. Montgomery multiplication in $GF(2^k)$. *Design, Codes and Cryptography*, 14(1):57–69, April 1998.

[31] Ç. K. Koç and B. Sunar. Mastrovito multiplier for all trinomials. *IEEE Transactions on Computers*, to appear, 1999.

[32] Ç. K. Koç and B. Sunar. Low-complexity bit-parallel canonical and normal basis multipliers for a class of finite fields. *IEEE Transactions on Computers*, 47(3):353–356, March 1998.

[33] K. Koyama, U. M. Maurer, T. Okamoto, and S. A. Vanstone. New public-key schemes based on elliptic curves over the ring $\ddagger_n$. In J. Feigenbaum, editor, *Advances in Cryptology — CRYPTO 91, Proceedings*, Lecture Notes in Computer Science, No. 576, pages 252–266. New York, NY: Springer-Verlag, 1991.

[34] A. Lenstra, H. W. Lenstra, M. Manasse, and J. Pollard. The number field sieve. In *Proceedings of the 22nd Annual ACM Symposium on Theory of Computing*, pages 564–572, 1990.

[35] A. K. Lenstra. Primality testing. In C. Pomerance, editor, *Cryptology and Computational Number Theory*, Proceedings of Symposia in Applied Mathematics, Volume 42, pages 13–25. Providence, RI: American Mathematical Society, 1990.

[36] H. W. Lenstra. Factoring integers with elliptic curves. *The Annals of Mathematics*, 126:649–673, 1987.

[37] R. Lidl and H. Niederreiter. *Finite Fields.* New York, NY: Cambridge University Press, 1987.

[38] R. Lidl and H. Niederreiter. *Introduction to Finite Fields and Their Applications.* New York, NY: Cambridge University Press, 1994.

[39] E. D. Mastrovito. VLSI architectures for multiplication over finite field $GF(2^n)$. In T. Mora, editor, *Applied Algebra, Algebraic Algorithms and Error-Correcting Codes*, Lecture Notes in Computer Science, No. 357, pages 297–309. New York, NY: Springer-Verlag, 1988.

[40] E. D. Mastrovito. *VLSI Architectures for Computation in Galois Fields.* PhD thesis, Linköping University, 1991.

[41] K. McCurley. Crypographic key distribution and computation in class groups. In R. Mollin, editor, *Number Theory and Applications.* Boston, MA: Kluwer Academic Publishers, 1989.

[42] R. J. McEliece. *Finite Fields for Computer Scientists and Engineers.* Boston, MA: Kluwer Academic Publishers, 1987.

[43] A. Menezes, T. Okamoto, and S. Vanstone. Reducing elliptic curve logarithms to logarithms in a finite field. *IEEE Transactions on Information Theory*, 39(5):1639–1646, September 1993.

[44] A. Menezes, S. Vanstone, and R. Zuccherato. Counting points on elliptic curves over $f_{2^m}$. *Mathematics of Computation*, 60:407–420, 1993.

[45] A. J. Menezes, editor. *Applications of Finite Fields.* Boston, MA: Kluwer Academic Publishers, 1993.

[46] A. J. Menezes. *Elliptic Curve Public Key Cryptosystems.* Boston, MA: Kluwer Academic Publishers, 1993.

[47] A. J. Menezes and S. A. Vanstone. Elliptic curve cryptosystems and their implementation. *Journal of Cryptology*, 6(4):209–224, 1993.

[48] V. Miller. Uses of elliptic curves in cryptography. In H. C. Williams, editor, *Advances in Cryptology — CRYPTO 85, Proceedings*, Lecture Notes in Computer Science, No. 218, pages 417–426. New York, NY: Springer-Verlag, 1985.

[49] V. Miller. Short programs for functions on curves. unpublished manuscript, 1986.

[50] F. Morain and J. Olivos. Speeding up the computations on an elliptic curve using addition-subtraction chains. Rapport de Recherche 983, INRIA, March 1989.

[51] R. Mullin, I. Onyszchuk, S. Vanstone, and R. Wilson. Optimal normal bases in $GF(p^n)$. *Discrete Applied Mathematics*, 22:149–161, 1988.

[52] T. Okamoto, A. Fujioka, and E. Fujisaki. An efficient digital signature scheme based on an elliptic curve over the ring $z_n$. In E. F. Brickel, editor, *Advances in Cryptology — CRYPTO 92, Proceedings*, Lecture Notes in Computer Science, No. 740, pages 54–65. New York, NY: Springer-Verlag, 1993.

[53] J. Omura and J. Massey. Computational method and apparatus for finite field arithmetic. U.S. Patent Number 4,587,627, May 1986.

[54] C. Paar. *Efficient VLSI Architectures for Bit Parallel Computation in Galois Fields.* PhD thesis, Universität GH Essen, VDI Verlag, 1994.

[55] C. Paar. A new architecture for a parallel finite field multiplier with low complexity based on composite fields. *IEEE Transactions on Computers*, 45(7):856–861, July 1996.

[56] S. Pohlig and M. Hellman. An improved algorithm for computing logarithms over $GF(p)$ and its cryptographic significance. *IEEE Transactions on Information Theory*, 24:106–110, 1978.

[57] J. Pollard. Monte carlo methods for index computation (mod $p$). *Mathematics of Computation*, 32:918–924, 1971.

[58] C. Pomerance. Fast, rigorous factorization and discrete logarithms algorithms. *Discrete algorithms and complexity*, pages 119–143, 1987.

[59] C. Pomerance. Factoring. In C. Pomerance, editor, *Cryptology and Computational Number Theory*, Proceedings of Symposia in Applied Mathematics, Volume 42, pages 27–47. Providence, RI: American Mathematical Society, 1990.

[60] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, February 1978.

[61] H. Rück. A note on elliptic curves over finite fields. *Mathematics of Computation*, 49:301–304, January 1987.

[62] R. Schoof. Elliptic curves over finite fields and the computation of square roots mod $p$. *Mathematics of Computation*, 44(170):483–494, April 1985.

[63] R. Schoof. Nonsingular plane cubic curves over finite fields. *Journal of Combinatorial Theory*, A 46:183–211, 1987.

[64] R. Schroeppel, S. O'Malley H. Orman, and O. Spatscheck. Fast key exchange with elliptic curve systems. In D. Coppersmith, editor, *Advances in Cryptology — CRYPTO 95*, Lecture Notes in Computer Science, No. 973, pages 43–56. New York, NY: Springer-Verlag, 1995.

[65] J. Silverman. *The Arithmetic of Elliptic Curves.* New York, NY: Springer-Verlag, 1986.

[66] J. Silverman and J. Tate. *Rational Points on Elliptic Curves.* New York, NY: Springer-Verlag, 1992.

[67] N. M. Stephens. Lenstra's factorization method based on elliptic curves. In H. C. Williams, editor, *Advances in Cryptology — CRYPTO 85, Proceedings*, Lecture Notes in Computer Science, No. 218, pages 409–416. New York, NY: Springer-Verlag, 1985.

[68] T.Itoh, O. Teechai, and S. Tsujii. A fast algorithm for computing multiplicative inverses in $GF(2^t)$ using normal bases. *J. Society for Electronic Communications*, 44:31–36, 1986.

[69] T.Itoh and S. Tsujii. Structure of parallel multipliers for a class of finite fields $GF(2^m)$. *Information and Computation*, 83:21–40, 1989.

[70] J. Voloch. A note on elliptic curves over finite fields. *Bull.Soc.Math.France*, 116:455–458, 1988.

[71] C. Wang and D. Pei. A VLSI design for computing exponentiations in $GF(2^m)$ and its application to generate pseudorandom number sequences. *IEEE Transactions on Computers*, 39:258–262, 1990.

[72] E. De Win, A. Bosselaers, S. Vandenberghe, P. De Gersem, and J. Vandewalle. A fast software implementation for arithmetic operations in $GF(2^n)$. In K. Kim and T. Matsumoto, editors, *Advances in Cryptology — ASIACRYPT 96*, Lecture Notes in Computer Science, No. 1163, pages 65–76. New York, NY: Springer-Verlag, 1996.

# APPENDICES

# Appendix A
# NOTATIONS & DEFINITIONS

| | | |
|---|---|---|
| $\mathcal{Z}_n$ | *cyclic group* | cyclic group with n elements |
| $G \oplus H$ | *direct sum* | direct sum of additive groups |
| $F_q$ | *finite field* | The *unique* finite field with $q = p^m$ elements, where p is a prime |
| $\overline{H}$ | *algebraic closure* | if $H = F_q$ then $\overline{H} = \bigcup_{m \geq 1} F_{q^m}$ |
| $F_q^*$ | $= F_q - \{0\}$ | the Abelian group formed by the nonzero elements of the finite field $F_q$ |
| $GF(q)$ | *Galois field* | Galois field equivalent to $F_q$ |
| $\mathcal{S}^*$ | $= (\mathcal{S} - \min \mathcal{S})$ | the subset of $\mathcal{S}$ with minimum element removed |
| $\lvert\mathcal{S}\rvert$ | *order of* $\mathcal{S}$ | the number of elements in the set $\mathcal{S}$ |
| $\phi$ | *Euler phi function* | number of integers less than and relatively prime to $n$: $\phi(n) = n \prod_{p \mid n} \left(1 - \frac{1}{p}\right)$ |

| | | |
|---|---|---|
| $E(F)$ | *F-rational points* | the set of points on the elliptic curve both of whose coordinates lie in the field $F$ |
| $\#E(F)$ | *order of* $E(F)$ | the number of elements in the set $E(F_p)$ |
| $Te$ | *Te function* | the function $Te : F_{2^m} \longrightarrow F_4$ defined by: $Te(\alpha) = \alpha + \alpha^{2^2} + \alpha^{2^4} + \cdots + \alpha^{2^{m-2}}$ |
| $Tr$ | *trace function* | the linear function $Tr : F_{2^m} \longrightarrow F_2$ defined by: $Tr(\alpha) = \alpha + \alpha^{2^1} + \alpha^{2^2} + \cdots + \alpha^{2^{m-1}}$ |
| $N$ | *norm function* | the function $N : F_{2^m} \longrightarrow F_2$ defined by: $Tr(\alpha) = \alpha \cdot \alpha^{2^1} \cdot \alpha^{2^2} \cdots \alpha^{2^{m-1}}$ |
| $\left(\frac{a}{p}\right)$ | *Legendre symbol* | Function that maps $a$ to $\pm 1$ depending on whether $x$ is a quadratic residue modulo $p$ or not, i.e., $= \begin{cases} 0, & \text{if } p \mid a \\ +1, & \text{if } a \text{ is a quadratic residue } \pmod p \\ -1, & \text{if } a \text{ is not a quadratic residue } \pmod p \end{cases}$ |
| $\chi$ | *quadratic character* | Function that maps $n \in F$ to $\pm 1$ depending on whether or not$n$ has a square root in the field $F$, i.e., $\chi(n) = \begin{cases} 0, & \text{if } x = 0, \\ +1, & \text{if } n \text{ has a square root in } F, \\ -1, & \text{if } n \text{ has no square root in } F. \end{cases}$ |

| | | |
|---|---|---|
| $L[n, c, \alpha]$ | *subexponential time* | Time complexity of an algorithm that is asymptotically faster (resp. slower) than an algorithm whose running time is fully exponential (resp. polynomial) in the input size $n$. It is defined as: $O(\exp((c + o(1))(\ln n)^{\alpha}(\ln \ln n)^{1-\alpha}))$ |
| $\|$ | *concatenation* | used to build a new vector from others (*see* page 101 for an example) |
| $\rightarrow$ $\leftarrow$ | *right & left shift* | used to shift a vector to right or left (*see* page 101 for an example) |
| $\begin{bmatrix} \mathbf{U} \\ \vdots \\ \mathbf{W} \end{bmatrix}_{n \times m}$ | *matrix construction* | used to construct an $n \times m$ matrix using the given $n$ vectors (of various lengths) as rows (*see* page 102 for an example) |
| $T_A$ | *AND delay* | the delay of a single 2-input `AND` gate |
| $T_X$ | *XOR delay* | the delay of a single 2-input `XOR` gate |
| $N[n, m]$ | $= \left\lceil \frac{n-1}{m} \right\rceil$ | a notation used in Section 11 to count the number of steps in the analysis |

# Appendix B
# ALGORITHMS

## B.1 Repeated *square-and-multiply* method

Let $G$ be a multiplicatively written finite group of order $n$, and $\alpha, \beta \in G$. The number or group operations used is at most $2\lceil \log_2 l \rceil$. (Source: [46])

**Input**: $\alpha \in G$, $l \in \mathcal{Z}$

**Output**: $\alpha^l$

1. Let $(b_t b_{t-1} \ldots b_1 b_0)$, be the binary representation of $l$, where $b_i \in \{0, 1\}$ and $b_t = 1$.

2. Set $\beta \longleftarrow \alpha$

3. For $i$ from $t-1$ down to 0 do

   $\beta \longleftarrow \beta \cdot \beta$

   If $b_i = 1$ then

   $\beta \longleftarrow \beta \cdot \alpha$

4. Output $\beta$

## B.2 Repeated *double-and-add* method

Let $G$ be a additively written finite group of order $n$, and $\alpha, \beta \in G$. The number or group operations used is at most $2\lceil \log_2 l \rceil$. (Source: [46])

**Input**: $\alpha \in G, l \in \mathcal{Z}$

**Output**: $\alpha^l$

1. Let $(b_t b_{t-1} \ldots b_1 b_0)$, be the binary representation of $l$, where $b_i \in \{0,1\}$ and $b_t = 1$.

2. Set $\beta \longleftarrow \alpha$

3. For $i$ from $t-1$ down to 0 do

   $\beta \longleftarrow \beta + \beta$

   If $b_i = 1$ then

   $\beta \longleftarrow \beta + \alpha$

4. Output $\beta$

## B.3 Euclidean Algorithm

This algorithm computes the greatest common divisor of 2 numbers $a$ and $b$ by repeated divisions. The number of steps is at most $2\log_2(\max\{2a, 2b\})$. (Source: [66])

**Input**: Two integers $a$ and $b$ such that $a > b$

**Output**: integer $d$ which is the greatest common divisor of $a$ and $b$, i.e., $d = (a, b)$

1. Let $i = 0$, $r_i = a$ and $r_{i+1}$.

2. Divide $r_i$ by $r_{i+1}$ and get the quotient $q_{i+1}$ and remainder $r_{i+2}$, i.e.,

$$r_i = r_{i+1} q_{i+1} + r_{i+2} \qquad \text{with} \qquad 0 \leq r_{i+2} < r_{i+1} \ .$$

3. If $r_{i+2} = 0$ then set $d = r_{i+1}$ and go to next step. If $r_{i+2} \neq 0$ then increment $i$, i.e., $i = i + 1$, and go to step 2.

4. Output $d$.

## B.4 Pollard's $(p-1)$ Algorithm

If $n$ has a factor $p$, such that $(p-1)$ has no large prime divisor, then the following method is virtually certain to find $p$. (Sources: [29, 66])

**Input**: A composite number $n$

**Output**: A factor $p$ of $n$

1. Choose an integer $v$ that is a product of small primes to small powers. (For example take $k = \text{lcm}\,[1, 2, 3, \ldots, u]$, for some small number $u$)
2. Choose an arbitrary integer $a$ such that $1 < a < n$.
3. Compute $(a \pmod n)$.
   If it is greater then 1, then it is a non-trivial factor of $n$; set $p$ to that value and go to step 7.
   Otherwise go to step 4.
4. Compute $(a^v \pmod n)$ by the repeated square-and-multiply method.
5. Compute $d = \gcd(a^v - 1, n)$ using Euclidean algorithm and the residue of $a^v$ modulo $n$ from step 4.
6. If $d = 1$ then go to step 1 and choose a larger $v$.
   If $d = n$ then go to step 2 and choose another $a$.
   Otherwise $1 < d < n$ will hold and $d$ will be a non-trivial divisor. Set $p$ to $d$ and goto step 7.
7. Output $p$.

## B.5 Lenstra's Elliptic Curve Factoring Algorithm

This is a probabilistic algorithm that uses elliptic curves. (Sources: [29, 66, 36]) For a simpler version with complexity analysis see [5].

**Input**: A composite number $n$

**Output**: A factor $p$ of $n$

1. Check that $\gcd(n, 6) = 1$ and that $n$ doesn't have the form $m^r$ for some $r \geq 2$.

2. Choose random integers $b, x_1, y_1$ between 1 and $n$.

3. Let $c = y_1^2 - x_1^3 - bx_1 \pmod{n}$, let $C$ be the cubic curve

$$C : y^2 = x^3 + bx + c$$

   and let $P = (y_1, x_1) \in C$ .

4. Check that $\gcd(4b^3 + 27c^2, n) = 1$. (If it equals $n$, go to step 2 and choose a new $b$. If it is strictly between 1 and $n$ then it is a non-trivial divisor. Set $p$ to that value and goto step 8.

5. Choose an integer $v$ that is a product of small primes to small powers. (For example take $v = \text{ lcm } [1, 2, 3, \ldots, u]$, for some small number $u$)

6. Compute

$$vP = \left(\frac{a_v}{d_v^2}, \frac{b_v}{d_v^3}\right) .$$

7. Compute $D = \gcd(d_v, n)$.

   If $D = 1$, either go to Step 5 and increase $v$ or go to Step 2 and choose a new curve. If $D = n$, then go to Step 5 and increase $v$.

   Otherwise $1 < D < n$ will hold and $d$ will be a non-trivial divisor. Set $p$ to $D$ and go to Step 8.

8. Output $p$.

# Appendix C
# THEOREMS

**Theorem C.1 (Chinese Remainder Theorem,[25])** *The solution of the system*

$$\begin{aligned} p_1 &\equiv p \pmod{m_1} \\ p_2 &\equiv p \pmod{m_2} \\ &\vdots \\ p_L &\equiv p \pmod{m_L} \end{aligned}$$

*is unique with respect to modulus* $M = \prod_{i=1}^{L} m_i$ *for pairwise relatively prime moduli* $m_i$, $L \geq i \geq 1$. *It can be computed using*

$$p = \left[ \sum_{i=1}^{L} p_i \cdot \left( M_i^{-1} \mod m_i \right) \cdot M_i \right] \mod M \ ,$$

*where*

$$M_i = \frac{M}{m_i} = m_1 m_2 \cdots m_{i-1} m_{i+1} \cdots m_L \ ,$$

*and, the inverse* $M_i^{-1} \pmod{m_i}$ *is defined as*

$$M_i^{-1} \cdot M_i = 1 \pmod{m_i} \ ,$$

*which exists since* $\gcd(M_i, m_i) = 1$ *and all* $m_i$*s are pairwise relatively prime.*

**Theorem C.2 (Prime Number Theorem,[2])** *If* $\pi(n)$ *is defined as the number of primes less than* $n$, *then*

$$\pi(n) \sim \frac{n}{\log n} \ .$$

# Appendix D
# NOTES ON FINITE FIELDS

- $Z_n$ is a finite field if and only if $n$ is a prime number.
- If the characteristic $m$ of a field is not 0, then it is a prime number. And that field contains a subfield having $m$ elements. This subfield is called *ground field* of the original field.
- If $F$ is a finite field of characteristic $p$, then it contains $p^n$ elements for some positive integer $n$. Furthermore, such a field exists for every prime $p$ and positive integer $n$.
- If $Z_p[x]/f(x)$ is the set of all equivalence classes in $Z_p[x]$ under the congruence modulo the irreducible polynomial $f(x)$ of order $n$, then it forms a field with $p^n$ elements.
- For every non-zero element $\alpha \in GF(q)$, the identity $\alpha^{q-1} = 1$ holds. Furthermore, an element $\alpha \in GF(q^m)$ lies in $GF(q)$ itself if and only if $\alpha^q = \alpha$.
- In every field $F = GF(q)$ there exists a primitive element.
- The *minimal polynomial* of an element $\alpha \in F_{p^k}^*$ is unique and is an irreducible polynomial given by

$$m_\alpha(x) = \prod_{\beta \in C(\alpha)} (x - \beta) \ ,$$

where $C(\alpha)$ is the set of *conjugates* of $\alpha$ with respect to $GF(p)$, i.e.,

$$C(\alpha) = \{\alpha, \alpha^p, \alpha^{p^2}, \ldots, \alpha^{p^{k-1}}\} \ .$$

- To simplify multiplication of polynomials, one can precompute the $z(i)$ values such that $1 + \alpha^i = \alpha^{z(i)}$. The table formed in that way is called a *Zech's log table.*

# Index